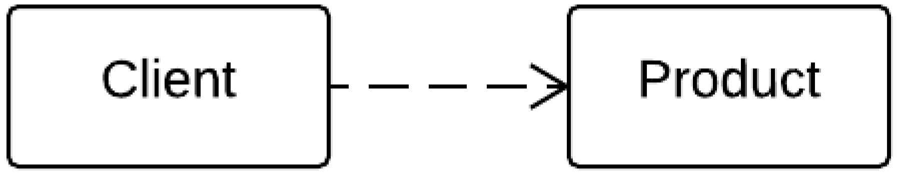
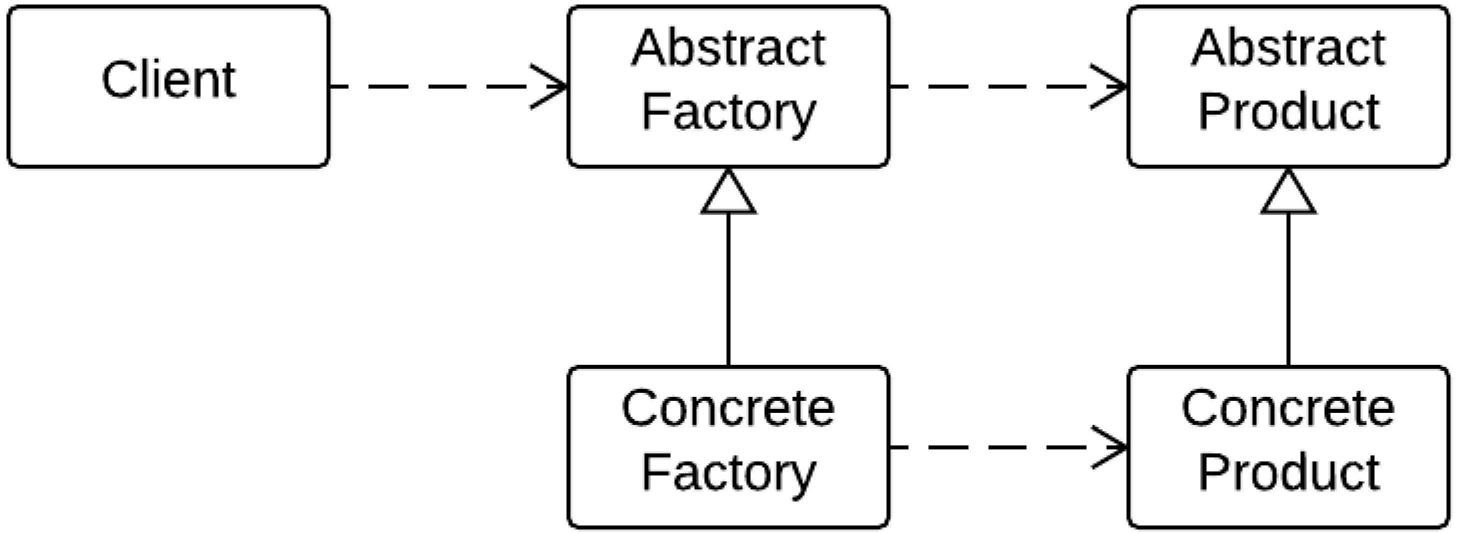
# Advancing from Constructor to Abstract Factory

Zoran Horvat
OWNER AT CODING HELMET CONSULTANCY
@zoranh75 www.codinghelmet.com

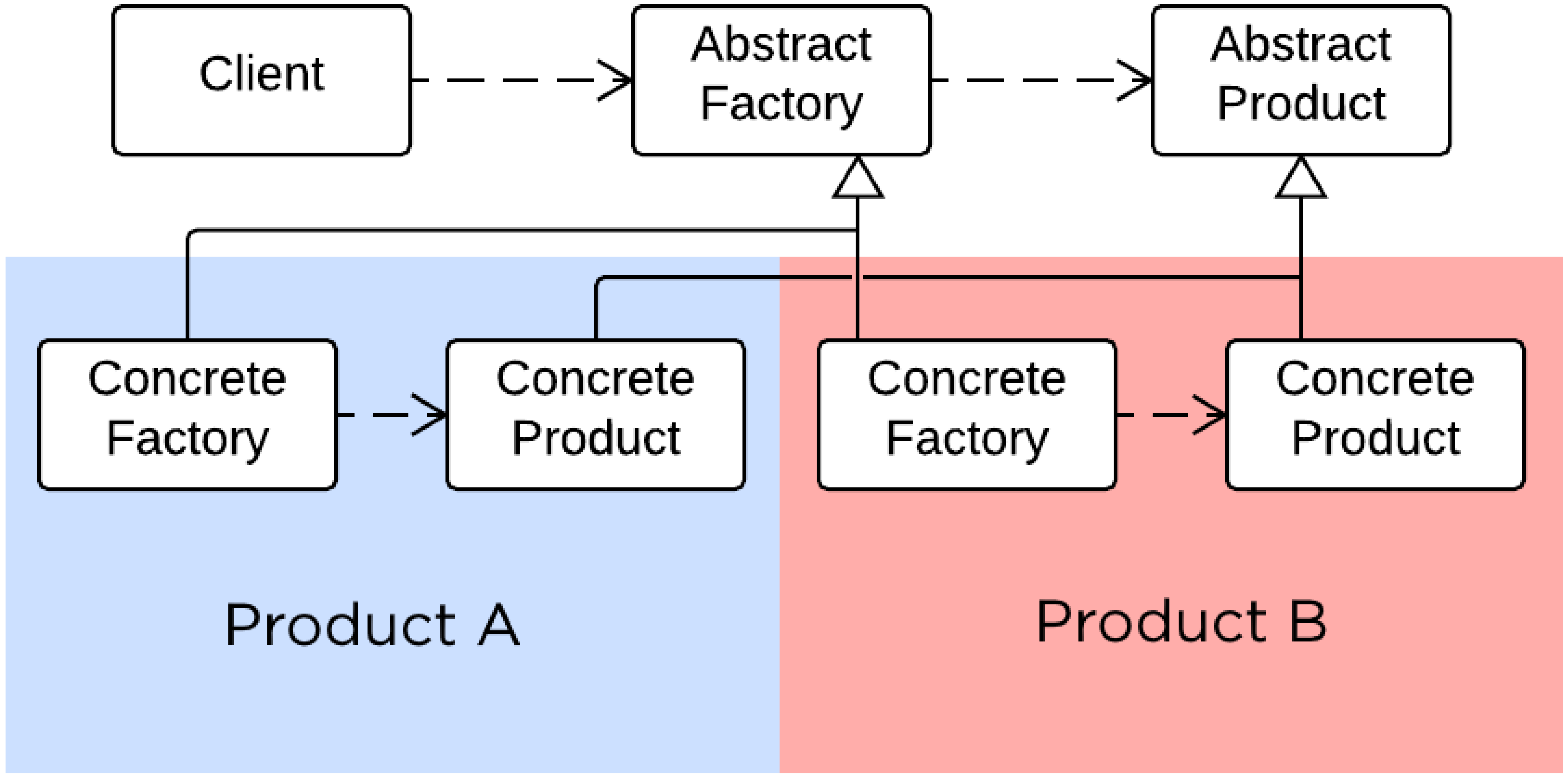
# Abstract Factory Principle

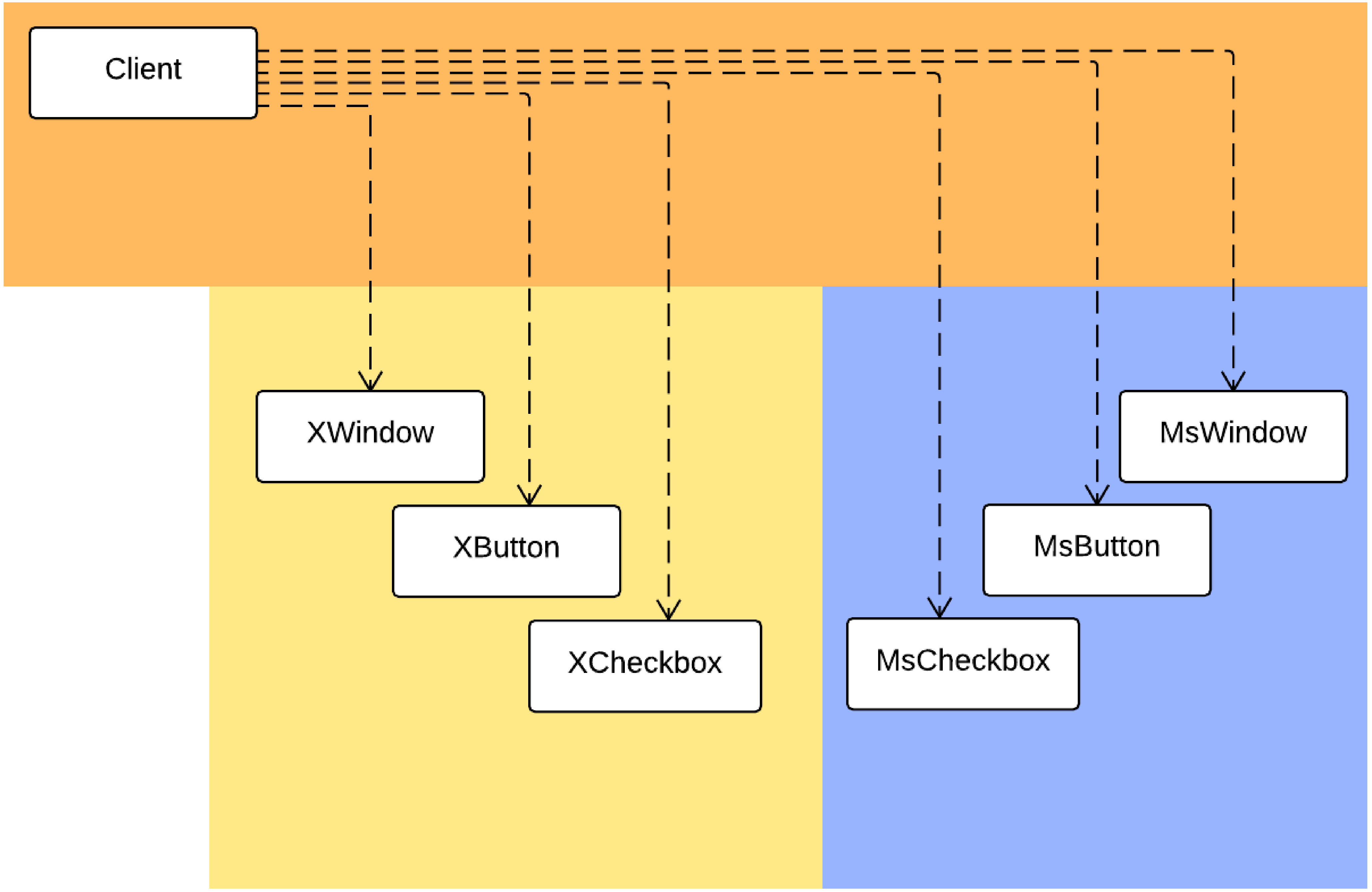


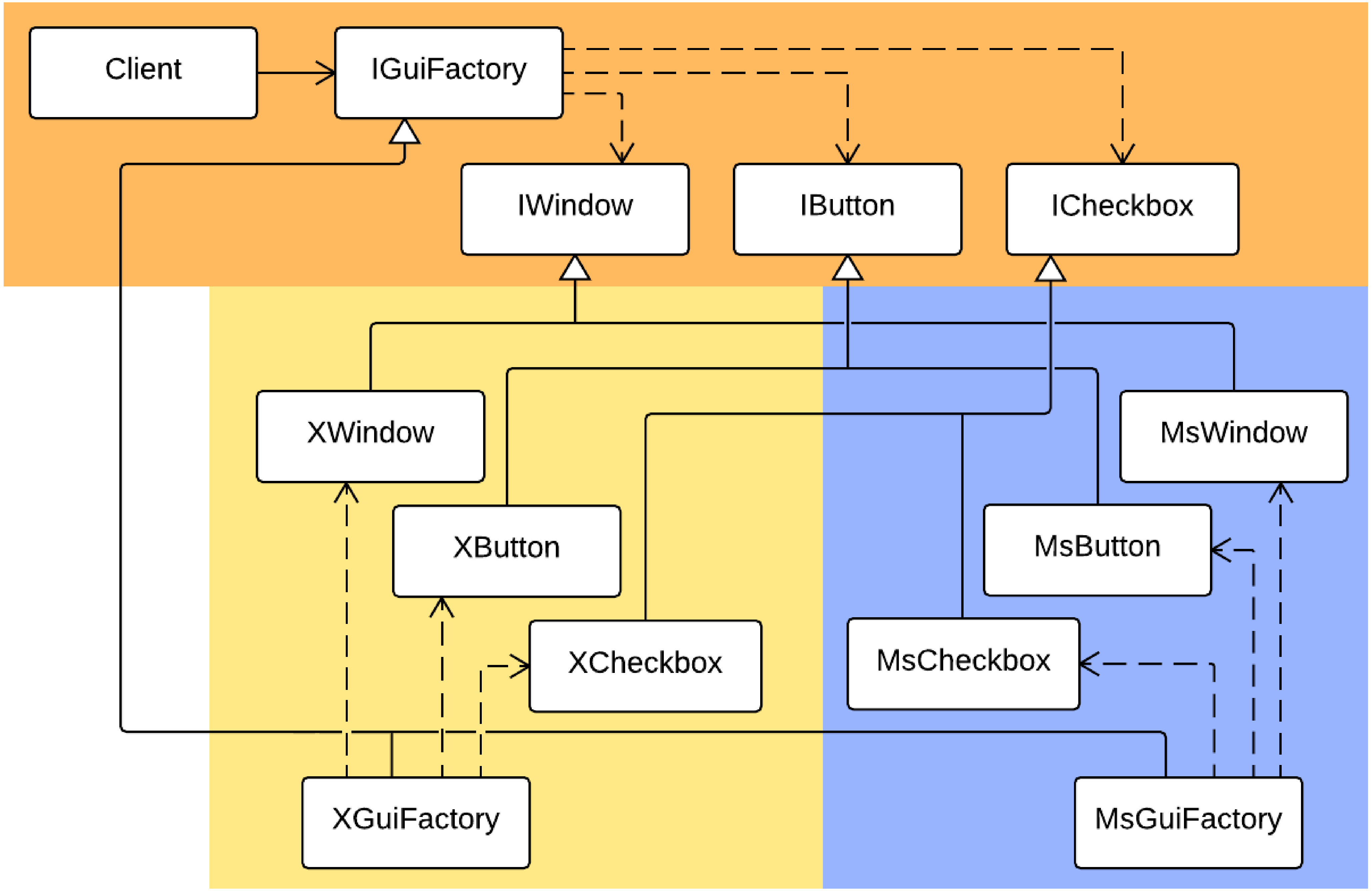
# Abstract Factory Principle

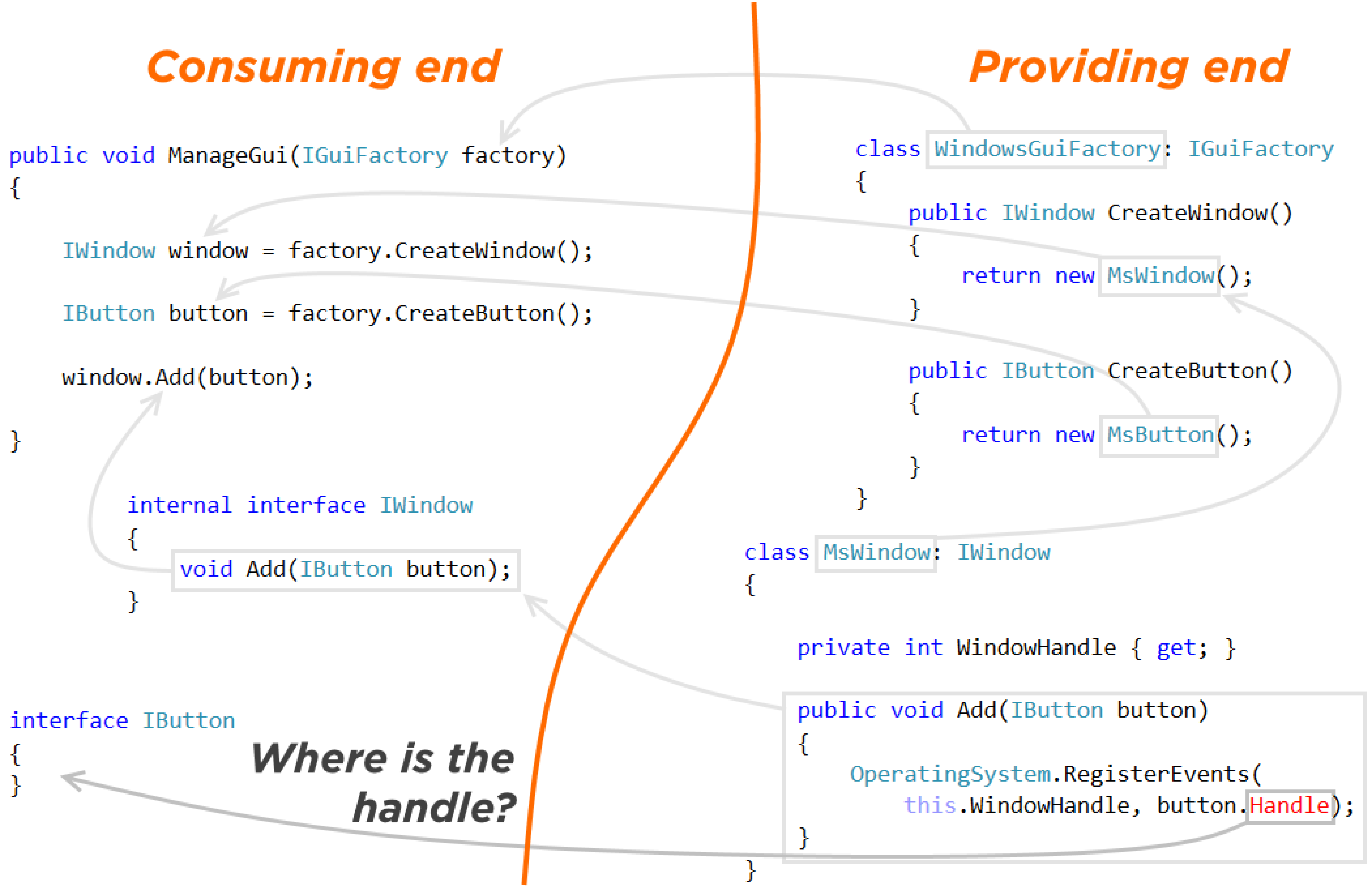


# Abstract Factory Principle




Client
XWindow
XButton
XCheckbox
MsWindow
MsButton
MsCheckbox


Client
IGuiFactory
IWindow
IButton
ICheckbox
XWindow
XButton
XCheckbox
XGuiFactory
MsWindow
MsButton
MsCheckbox
MsGuiFactory



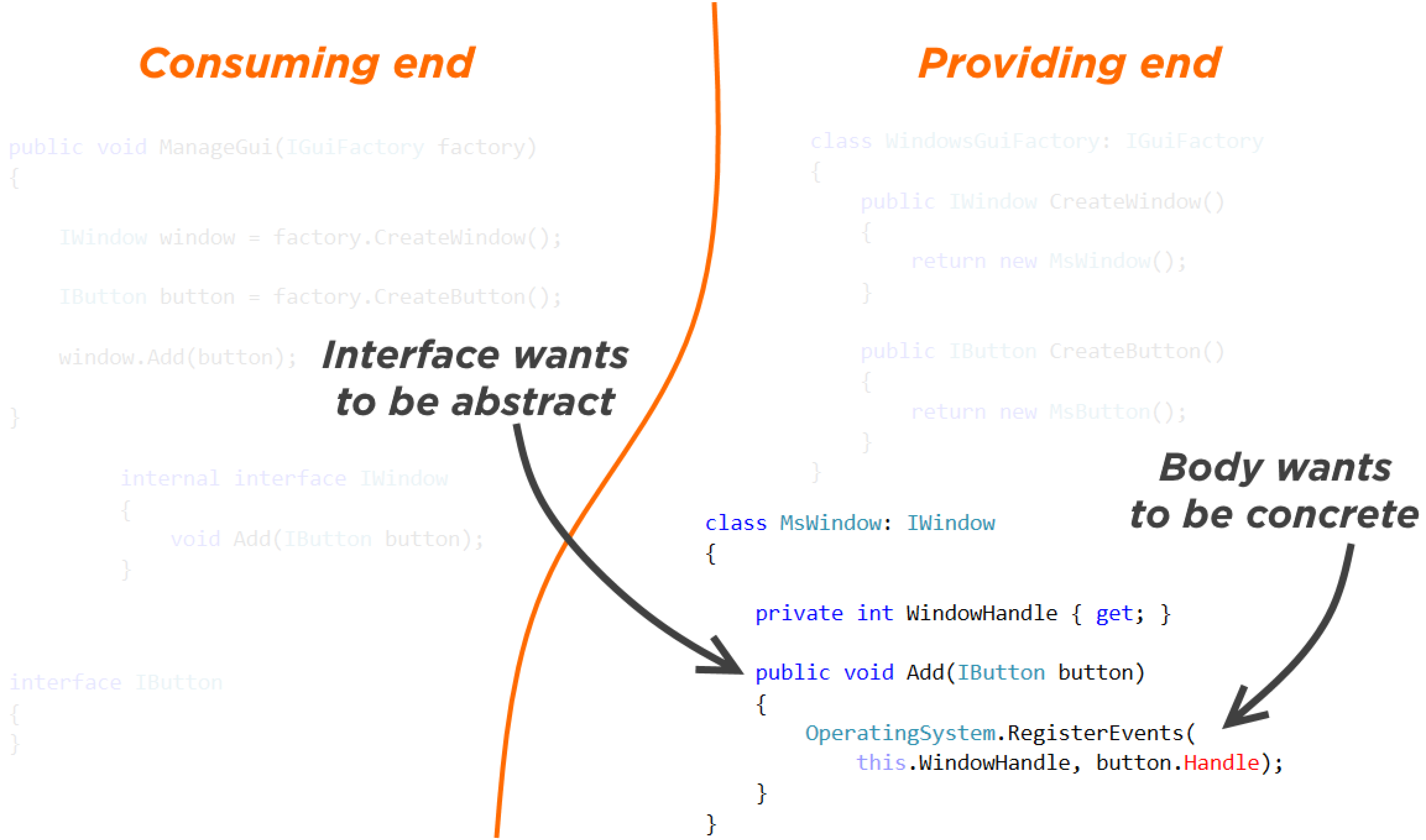
## Consuming end

```
public void ManageGui(IGuiFactory factory)
{

    IWindow window = factory.CreateWindow();

    IButton button = factory.CreateButton();

    window.Add(button);

}

    internal interface IWindow
    {
        void Add(IButton button);
    }


interface IButton
{
}
```

**Where is the handle?**

## Providing end

```
        class WindowsGuiFactory: IGuiFactory
        {
            public IWindow CreateWindow()
            {
                return new MsWindow();
            }

            public IButton CreateButton()
            {
                return new MsButton();
            }
        }

class MsWindow: IWindow
{

    private int WindowHandle { get; }

    public void Add(IButton button)
    {
        OperatingSystem.RegisterEvents(
            this.WindowHandle, button.Handle);
    }
}
```



## Consuming end

```
public void ManageGui(IGuiFactory factory)
{
    IWindow window = factory.CreateWindow();
    IButton button = factory.CreateButton();
    window.Add(button);
}

        internal interface IWindow
        {
            void Add(IButton button);
        }

interface IButton
{
}
```

## Providing end

```
class WindowsGuiFactory: IGuiFactory
{
    public IWindow CreateWindow()
    {
        return new MsWindow();
    }

    public IButton CreateButton()
    {
        return new MsButton();
    }
}
```

```
class MsWindow: IWindow
{
    private int WindowHandle { get; }

    public void Add(IButton button)
    {
        OperatingSystem.RegisterEvents(
            this.WindowHandle, button.Handle);
    }
}
```

## Consuming end

## Providing end

```
class MsWindow: IWindow
{

    private int WindowHandle { get; }

    public void Add(IButton button)
    {
        OperatingSystem.RegisterEvents(
            this.WindowHandle, button.Handle);
    }
}
```

**No implementation-specific features allowed**

## Consuming end

## Providing end

```
class MsWindow: IWindow
{

    private int WindowHandle { get; }

    public void Add(IButton button)
    {

        MsButton msButton = button as MsButton;

        if (msButton == null)
            throw new ArgumentException();

        OperatingSystem.RegisterEvents(
            this.WindowHandle, msButton.Handle);

    }
}
```

**Cast before using?**

## Consuming end

```
public void ManageGui(IGuiFactory factory)
{

    IWindow window = factory.CreateWindow();

    IButton button = new XButton();

    window.Add(button);

}
```

**Unexpected object type causes an exception**

## Providing end

```
class MsWindow: IWindow
{

    private int WindowHandle { get; }

    public void Add(IButton button)
    {

        MsButton msButton = button as MsButton;

        if (msButton == null)
            throw new ArgumentException();

        OperatingSystem.RegisterEvents(
            this.WindowHandle, msButton.Handle);

    }
}
```

## How do We Progress with Abstract Factory?

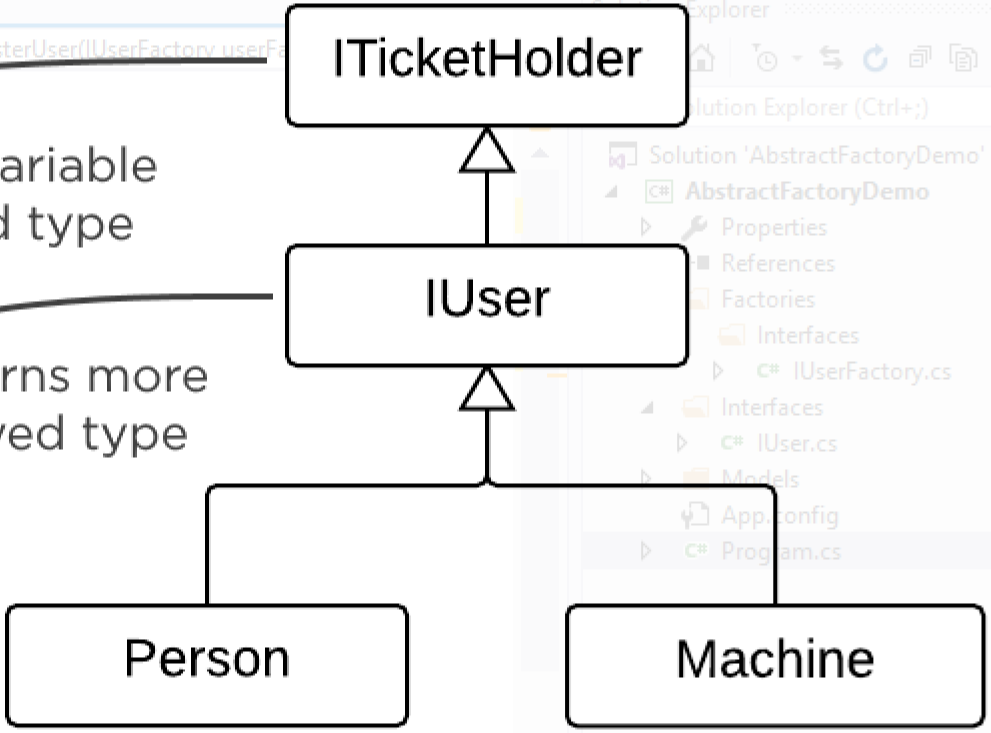
**ɔes causes issues**

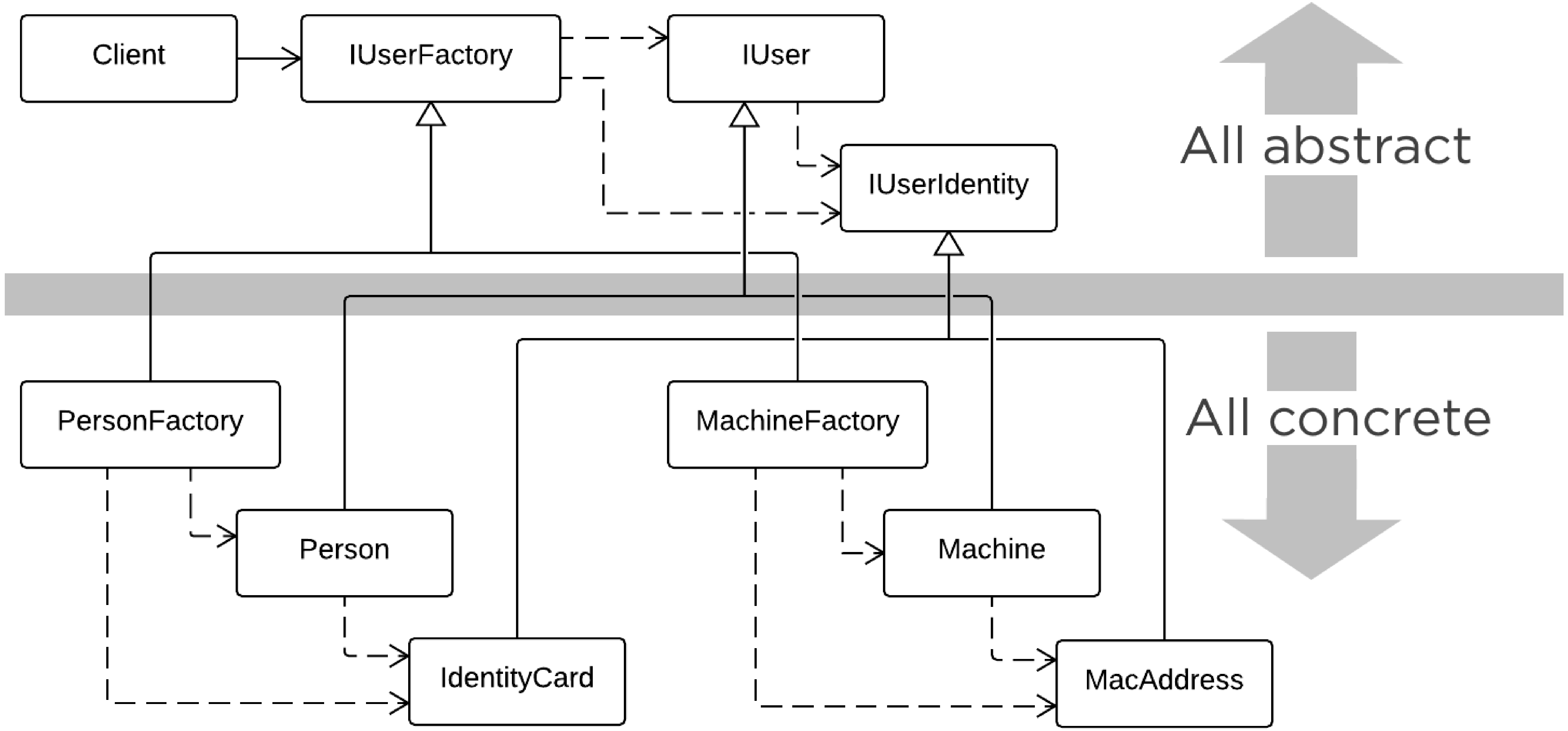
```
using AbstractFactoryDemo.Factories.Interfaces;
using AbstractFactoryDemo.Interfaces;

namespace AbstractFactoryDemo
{
    class Program
    {

        static void RegisterUser(IUserFactory userFactory)
        {
            ITicketHolder holder = userFactory.CreateUser();
        }

        static void Main(string[] args)
        {
        }
    }
}
```

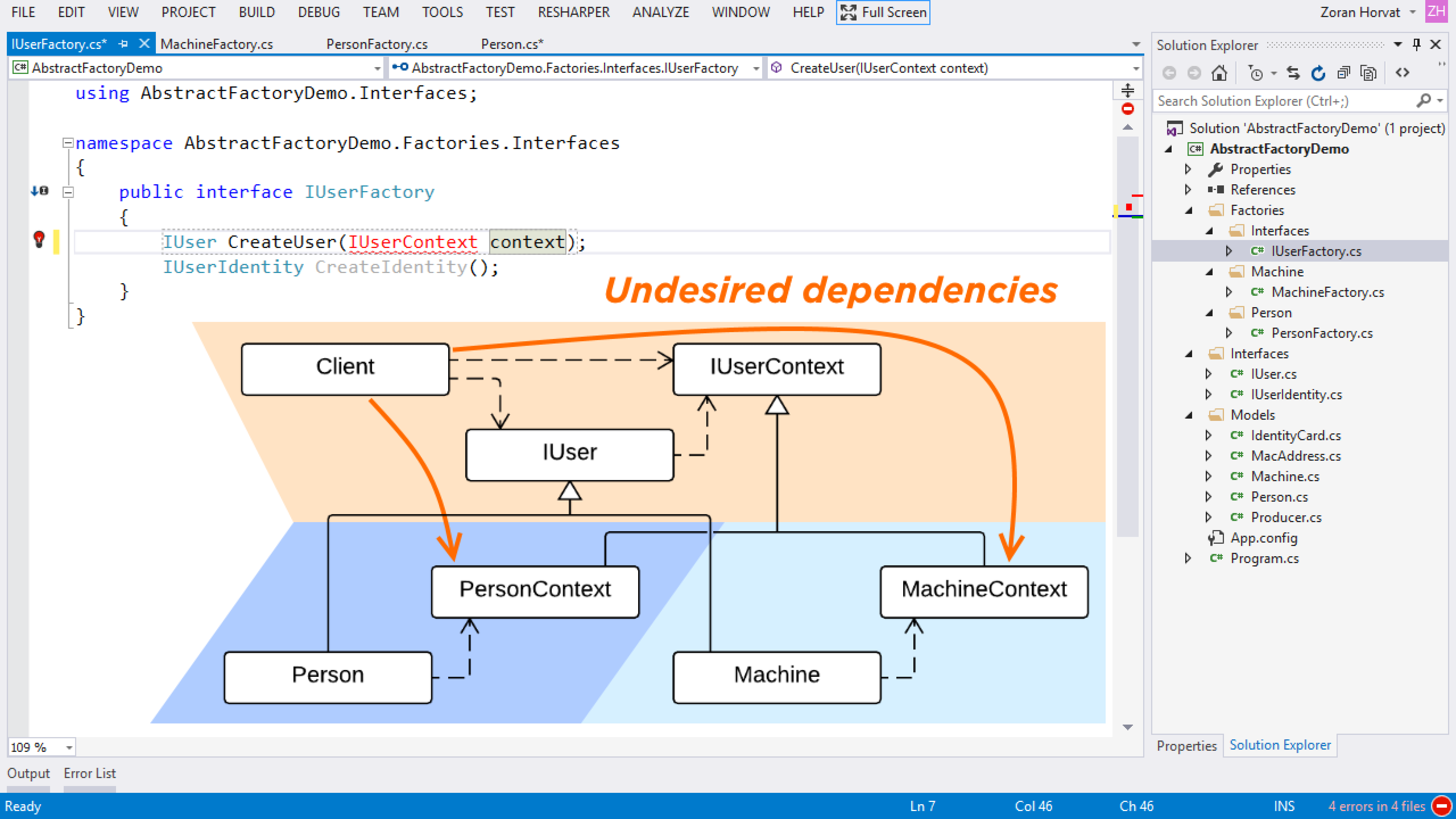
Assigns to a variable of less derived type

Returns more derived type

***Similar to covariance!***

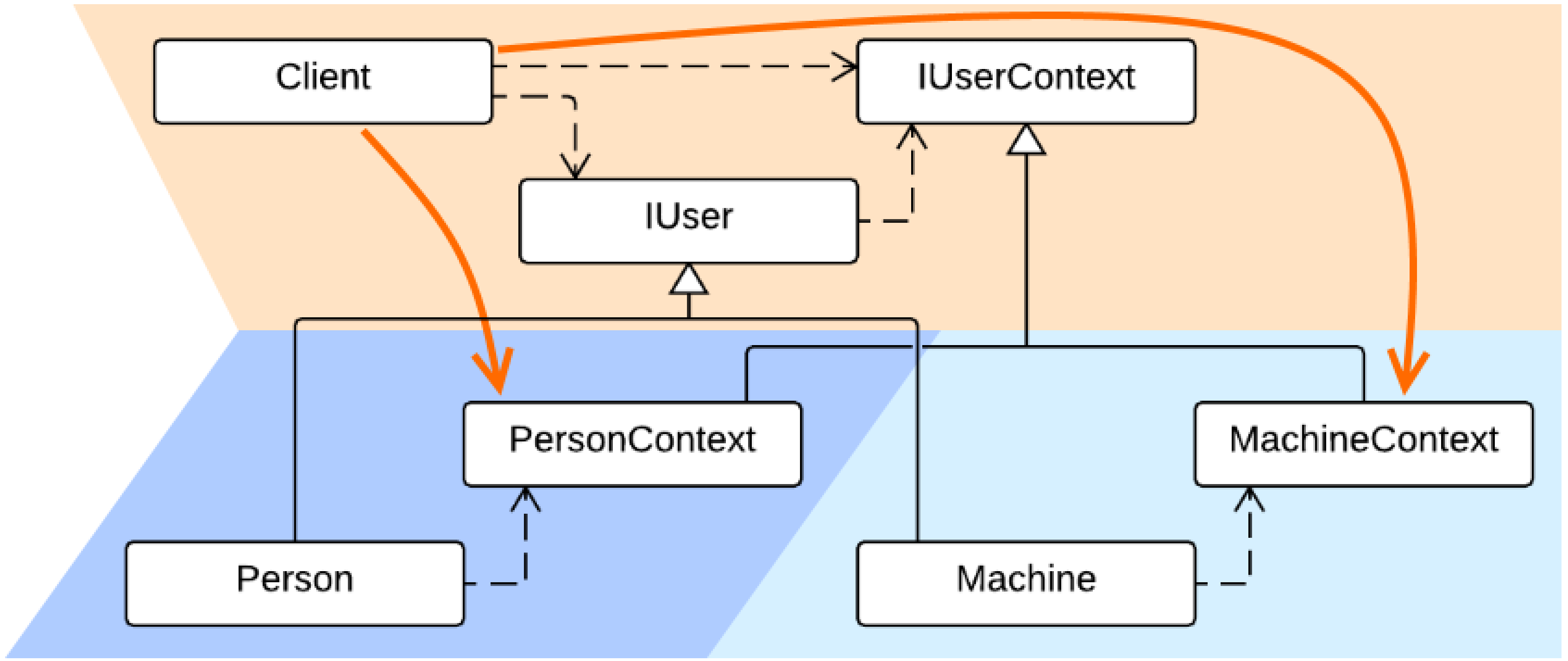


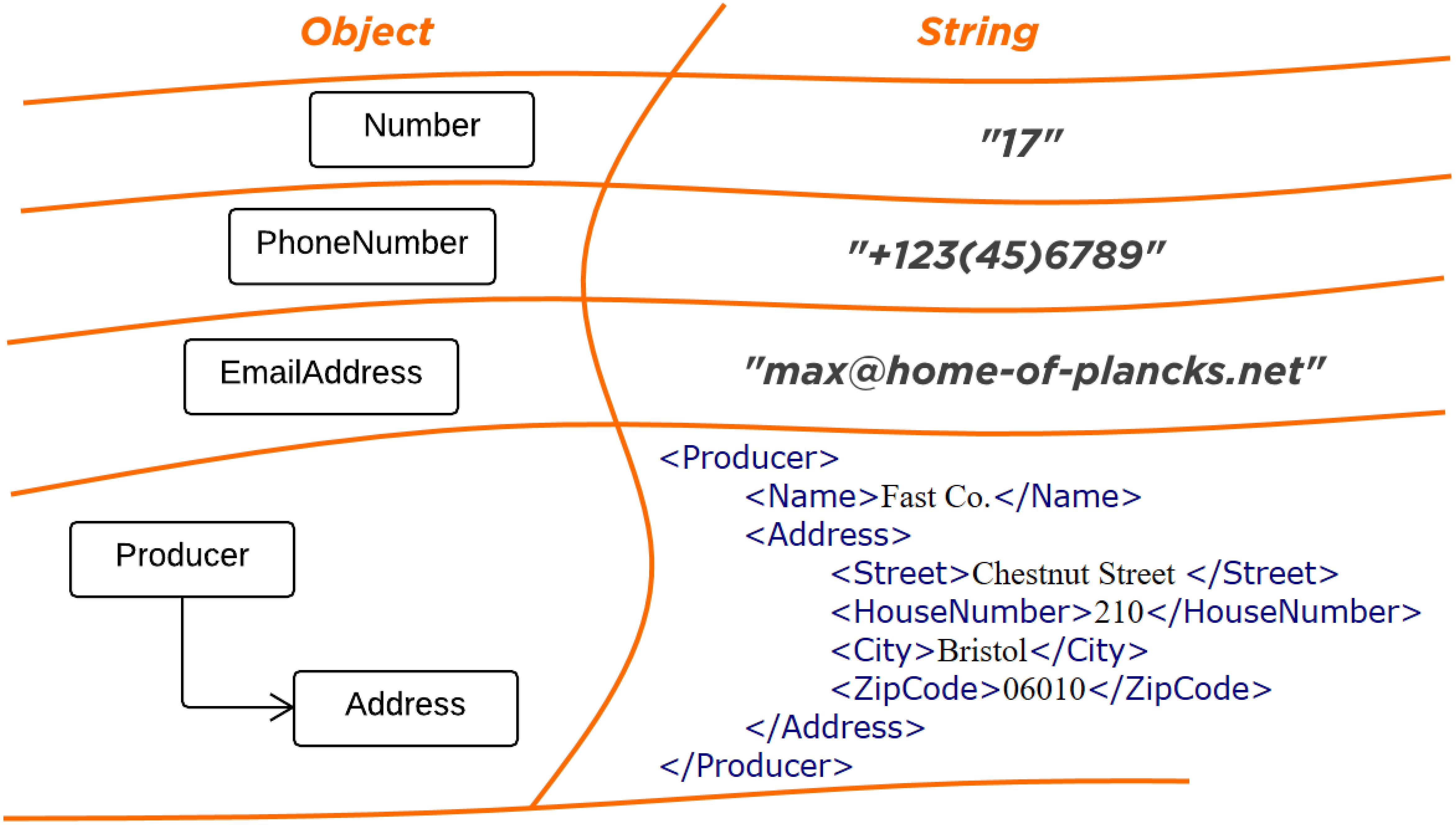

Client
IUserFactory
IUser
IUserIdentity
All abstract
PersonFactory
MachineFactory
All concrete
Person
Machine
IdentityCard
MacAddress


FILE EDIT VIEW PROJECT BUILD DEBUG TEAM TOOLS TEST RESHARPER ANALYZE WINDOW HELP Full Screen
Zoran Horvat ZH
IUserFactory.cs* MachineFactory.cs PersonFactory.cs Person.cs*
AbstractFactoryDemo
AbstractFactoryDemo.Factories.Interfaces.IUserFactory
CreateUser(IUserContext context)
Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'AbstractFactoryDemo' (1 project)
AbstractFactoryDemo
Properties
References
Factories
Interfaces
IUserFactory.cs
Machine
MachineFactory.cs
Person
PersonFactory.cs
Interfaces
IUser.cs
IUserIdentity.cs
Models
IdentityCard.cs
MacAddress.cs
Machine.cs
Person.cs
Producer.cs
App.config
Program.cs
109 %
Properties Solution Explorer
Output Error List
Ready Ln 7 Col 46 Ch 46 INS 4 errors in 4 files

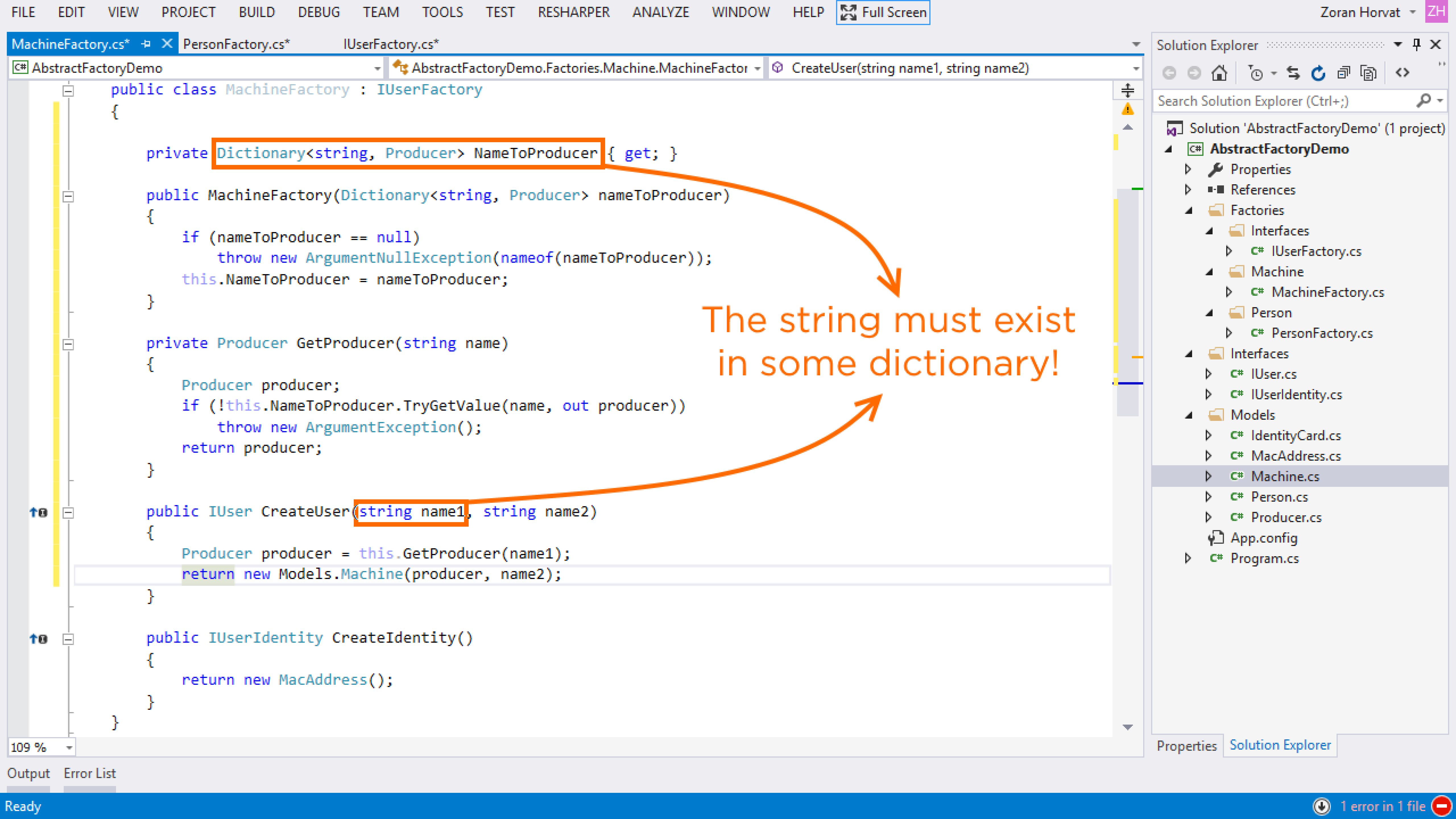
```
using AbstractFactoryDemo.Interfaces;

namespace AbstractFactoryDemo.Factories.Interfaces
{
    public interface IUserFactory
    {
        IUser CreateUser(IUserContext context);
        IUserIdentity CreateIdentity();
    }
}
```

## *Undesired dependencies*



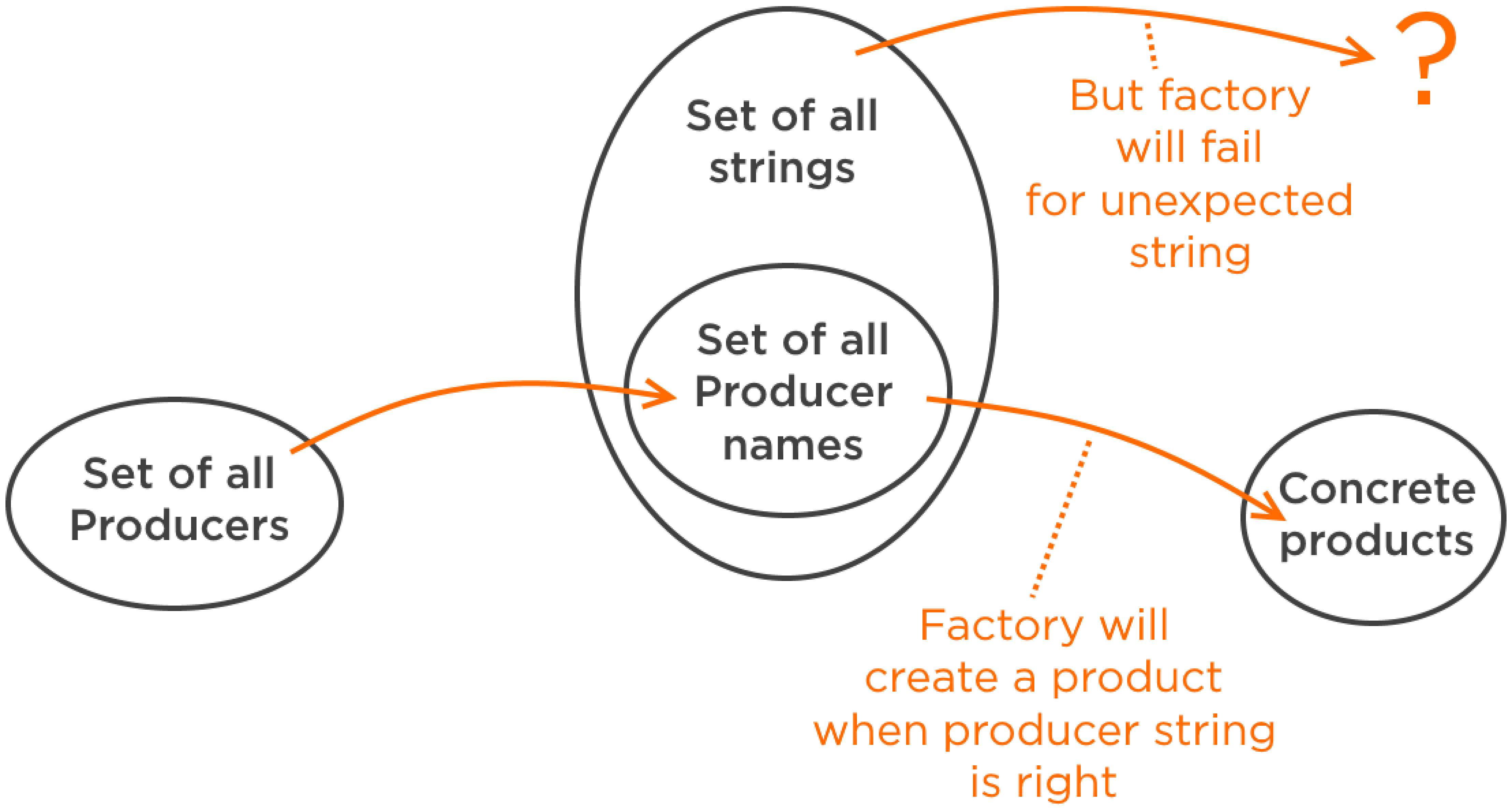


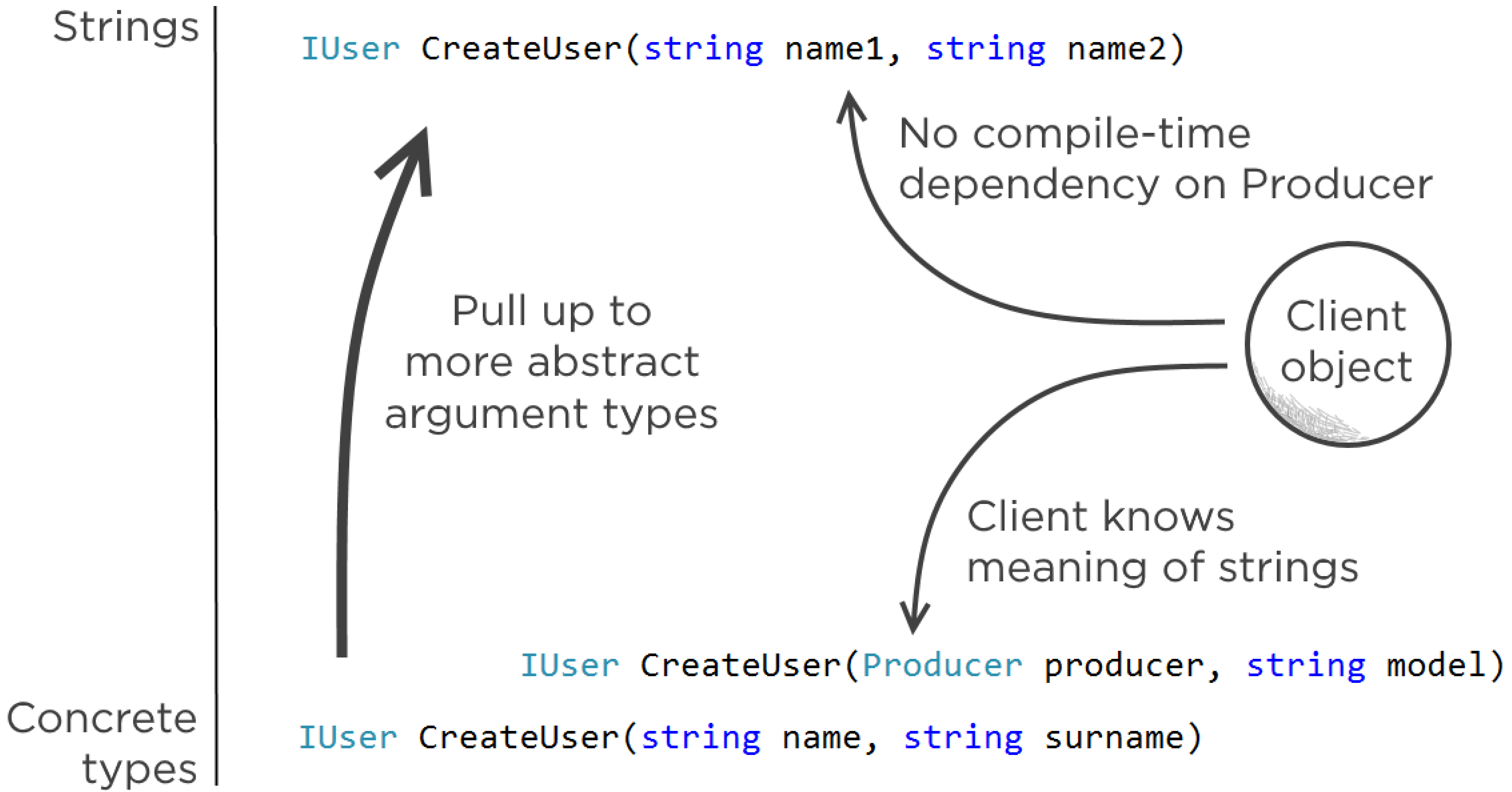
```
<Producer>
    <Name>Fast Co.</Name>
    <Address>
        <Street>Chestnut Street </Street>
        <HouseNumber>210</HouseNumber>
        <City>Bristol</City>
        <ZipCode>06010</ZipCode>
    </Address>
</Producer>
```


FILE EDIT VIEW PROJECT BUILD DEBUG TEAM TOOLS TEST RESHARPER ANALYZE WINDOW HELP Full Screen
Zoran Horvat
MachineFactory.cs* PersonFactory.cs* IUserFactory.cs*
AbstractFactoryDemo
AbstractFactoryDemo.Factories.Machine.MachineFactor
CreateUser(string name1, string name2)
Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'AbstractFactoryDemo' (1 project)
AbstractFactoryDemo
Properties
References
Factories
Interfaces
IUserFactory.cs
Machine
MachineFactory.cs
Person
PersonFactory.cs
Interfaces
IUser.cs
IUserIdentity.cs
Models
IdentityCard.cs
MacAddress.cs
Machine.cs
Person.cs
Producer.cs
App.config
Program.cs
109 %
Properties Solution Explorer
Output Error List
Ready
1 error in 1 file

```csharp
    public class MachineFactory : IUserFactory
    {

        private Dictionary<string, Producer> NameToProducer { get; }

        public MachineFactory(Dictionary<string, Producer> nameToProducer)
        {
            if (nameToProducer == null)
                throw new ArgumentNullException(nameof(nameToProducer));
            this.NameToProducer = nameToProducer;
        }

        private Producer GetProducer(string name)
        {
            Producer producer;
            if (!this.NameToProducer.TryGetValue(name, out producer))
                throw new ArgumentException();
            return producer;
        }

        public IUser CreateUser(string name1, string name2)
        {
            Producer producer = this.GetProducer(name1);
            return new Models.Machine(producer, name2);
        }

        public IUserIdentity CreateIdentity()
        {
            return new MacAddress();
        }
    }
```

The string must exist in some dictionary!


Set of all strings
But factory will fail for unexpected string
?
Set of all Producers
Set of all Producer names
Concrete products
Factory will create a product when producer string is right


Strings
No compile-time dependency on Producer
Pull up to more abstract argument types
Client object
Client knows meaning of strings
Concrete types

```
IUser CreateUser(string name1, string name2)
```

```
IUser CreateUser(Producer producer, string model)
```

```
IUser CreateUser(string name, string surname)
```

Strings

```
IUser CreateUser(string name1, string name2)
```

**What is good**

Factory is still abstract
Nobody depends on concrete products
There is some extendibility (Open-Closed Principle)

**What could be better**

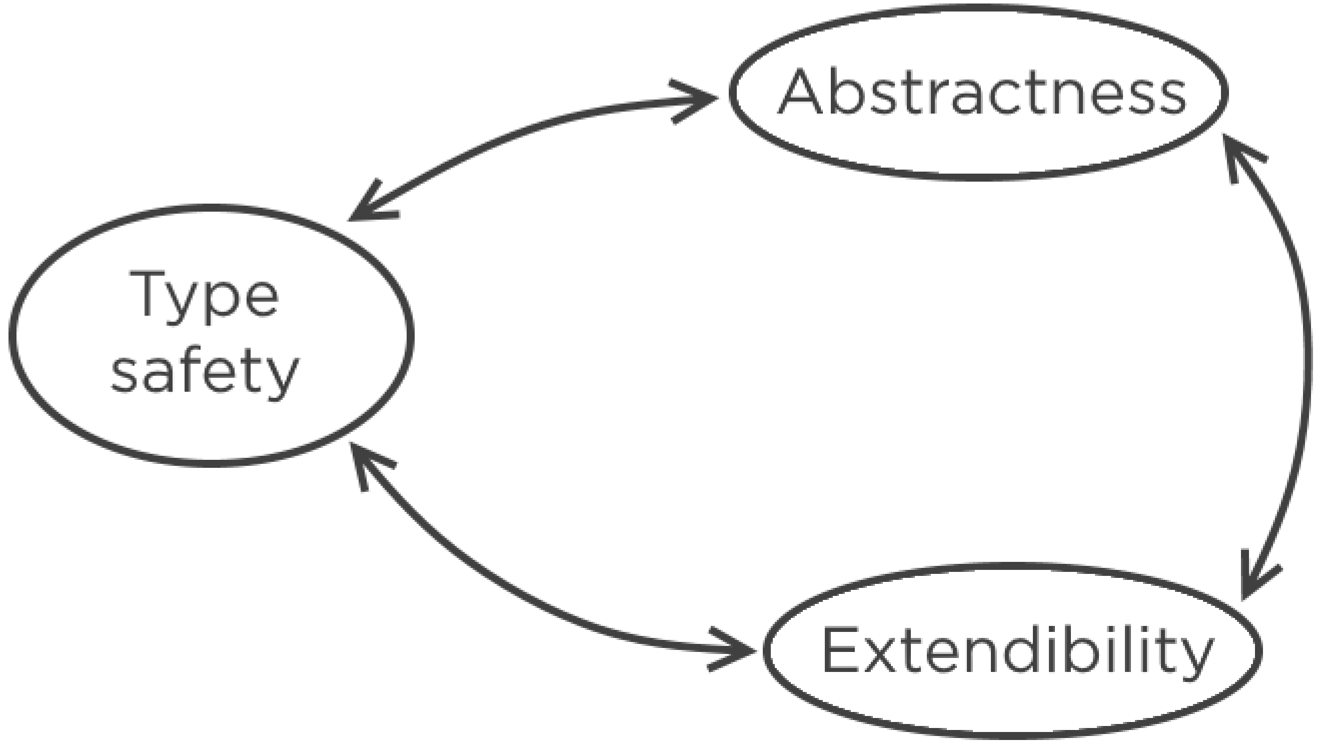
No compile-time type checking
Client must know about concrete products
Short of extendibility (Open-Closed Principle)

```
IUser CreateUser(Producer producer, string model)
```

```
IUser CreateUser(string name, string surname)
```

Concrete types

## Playing the Game



**Trade-off #1**
Less type safety
More abstractness
Some extendibility

**Trade-off #2**
Less abstractness
More type safety
Less extendibility

Support only one family of concrete products!

## Summary

**Overall impression:**

- Abstract Factory pattern has many limitations
- But other creational patterns will rely on it

**Classical example – GUI elements**

- There was an issue with casting abstract product into concrete class
- It puts emphasis on abstractness

## Summary

**Constructor arguments**

- Any object construction involves calling the constructor
- Different concrete products come with different constructors
- Abstract factory must reconcile these differences

**Stringly-typed factory**

- Lets us unify signatures of otherwise unrelated methods that create objects

***Next module –***
*Avoiding Excess Factory Abstractness*