Understanding Dependencies, Covariance, and Contravariance

Zoran Horvat

OWNER AT CODING HELMET CONSULTANCY

@zoranh75 www.codinghelmet.com

We May Encounter Problems While...

Instantiating an object through the constructor

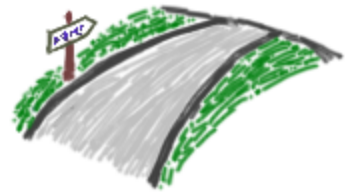
Assigning references to variables

Passing objects as arguments

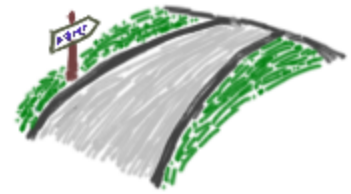
In This Module...

Constructor preconditions
With special attention to class dependencies

Covariance and contravariance of generic types



```
class RoadSegment
{
    private double distanceKm;
    private TimeSpan timeSpent;

    public RoadSegment(double distanceKm, TimeSpan timeSpent)
    {
        this.DistanceKm = distanceKm;
        this.TimeSpent = timeSpent;
    }

    public double DistanceKm...

    public TimeSpan TimeSpent...

    public double VelocityKph => this.DistanceKm / this.TimeSpent.TotalHours;

    public ISupplies DemandSupplies()
    {
        return new Fuel(this.DistanceKm * 8.0 / 100.0);
    }
}
```

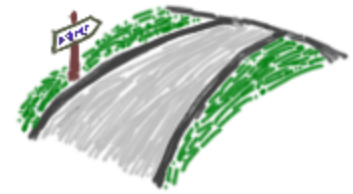


```
class RoadSegment
{
    private double distanceKm;
    private TimeSpan timeSpent;

    public RoadSegment(double distanceKm, TimeSpan timeSpent)
    {
        this.DistanceKm = distanceKm;
        this.TimeSpent = timeSpent;
    }

    public double DistanceKm...

    public TimeSpan TimeSpent...

    public double VelocityKph => this.DistanceKm / this.TimeSpent.TotalHours;

    public ISupplies DemandSupplies()
    {
        //return new Fuel(this.DistanceKm * 8.0 / 100.0);
        return new BottleOfWater((int)Math.Ceiling(this.distanceKm / 5.0));
    }

}
```

How do we decide?

There must be some switching logic

Better yet: Move logic to a separate object!

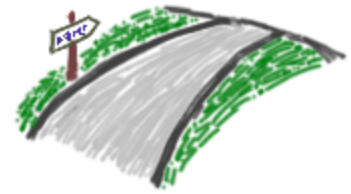


```
class RoadSegment
{
    private ISuppliesCalculator SuppliesCalculator { get; }
    private double distanceKm;
    private TimeSpan timeSpent;

    public RoadSegment(ISuppliesCalculator suppliesCalculator,
                       double distanceKm, TimeSpan timeSpent)
    {
        if (suppliesCalculator == null) throw new ArgumentNullException();
        this.SuppliesCalculator = suppliesCalculator;
        this.DistanceKm = distanceKm;
        this.TimeSpent = timeSpent;
    }

    public double DistanceKm...

    public TimeSpan TimeSpent...

    public double VelocityKph => this.DistanceKm / this.TimeSpent.TotalHours;

    public ISupplies DemandSupplies() =>
        this.SuppliesCalculator.EstimateFromDistance(this.DistanceKm);
}
```

Introduce a new object

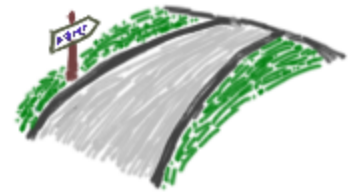
Supply it through the constructor

Delegate the call to the dependency



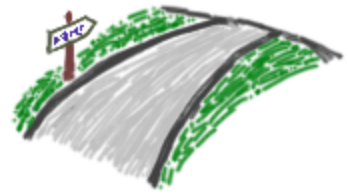
```
class RoadSegment
{
    private ISuppliesCalculator SuppliesCalculator { get; }
    private double distanceKm;
    private TimeSpan timeSpent;

    public RoadSegment(ISuppliesCalculator suppliesCalculator,
                       double distanceKm, TimeSpan timeSpent)
    {
        if (suppliesCalculator == null) throw new ArgumentNullException();
        this.SuppliesCalculator = suppliesCalculator;
        this.DistanceKm = distanceKm;
        this.TimeSpent = timeSpent;
    }

    public double DistanceKm...

    public TimeSpan TimeSpent...

    public double VelocityKph => this.DistanceKm / this.TimeSpent.TotalHours;

    public ISupplies DemandSupplies() =>
        this.SuppliesCalculator.EstimateFromDistance(this.DistanceKm);

}
```

ISupplies looks like an example of the Composite pattern



```
class RoadSegment
{
    private ISuppliesCalculator SuppliesCalculator { get; }
    private double distanceKm;
    private TimeSpan timeSpent;

    public RoadSegment(ISuppliesCalculator suppliesCalculator,
                       double distanceKm, TimeSpan timeSpent)
    {
        if (suppliesCalculator == null) throw new ArgumentNullException();
        this.SuppliesCalculator = suppliesCalculator;
        this.DistanceKm = distanceKm;
        this.TimeSpent = timeSpent;
    }

    public double DistanceKm...

    public TimeSpan TimeSpent...

    public double VelocityKph => this.DistanceKm / this.TimeSpent.TotalHours;

    public ISupplies DemandSupplies() =>
        this.SuppliesCalculator.EstimateFromDistance(this.DistanceKm);

}
```

Dependency is mandatory



Dependency

```
class RoadSegment
{
    private ISuppliesCalculator SuppliesCalculator { get; }
    private double distanceKm;
    private TimeSpan timeSpent;

    public RoadSegment(ISuppliesCalculator suppliesCalculator,
                       double distanceKm, TimeSpan timeSpent)
    {
        if (suppliesCalculator == null) throw new ArgumentNullException();
        this.SuppliesCalculator = suppliesCalculator;
        this.DistanceKm = distanceKm;
        this.TimeSpent = timeSpent;
    }

    public double DistanceKm...

    public TimeSpan TimeSpent...

    public double VelocityKph => this.DistanceKm / this.TimeSpent.TotalHours;

    public ISupplies DemandSupplies() =>
        this.SuppliesCalculator.EstimateFromDistance(this.DistanceKm);

}
```

State

Constructor Preconditions

Most of the classes

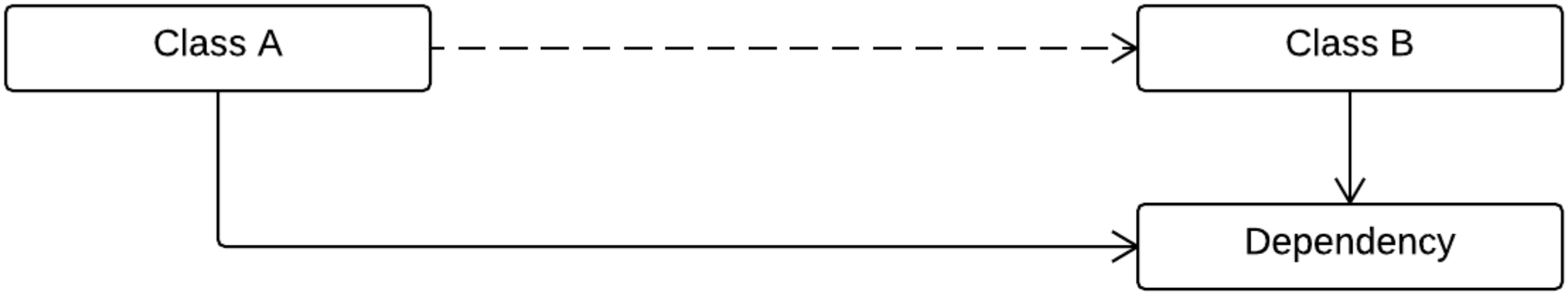
ctor is called after all prerequisites hold

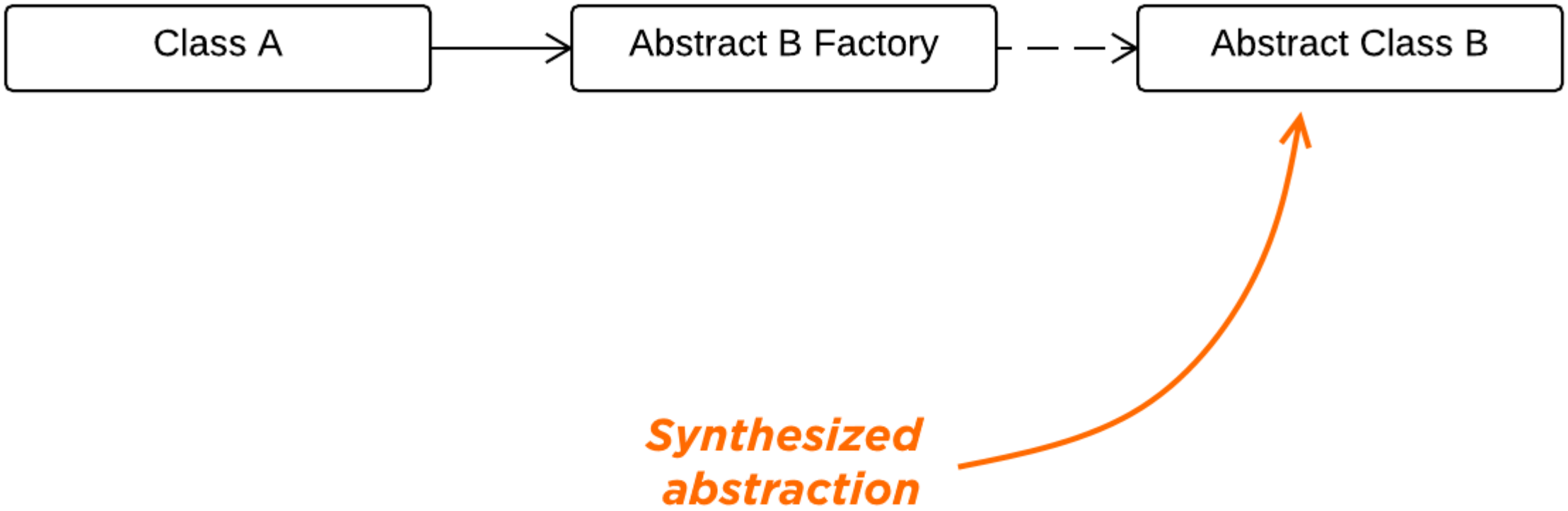
uctor then guarantees to create a new object

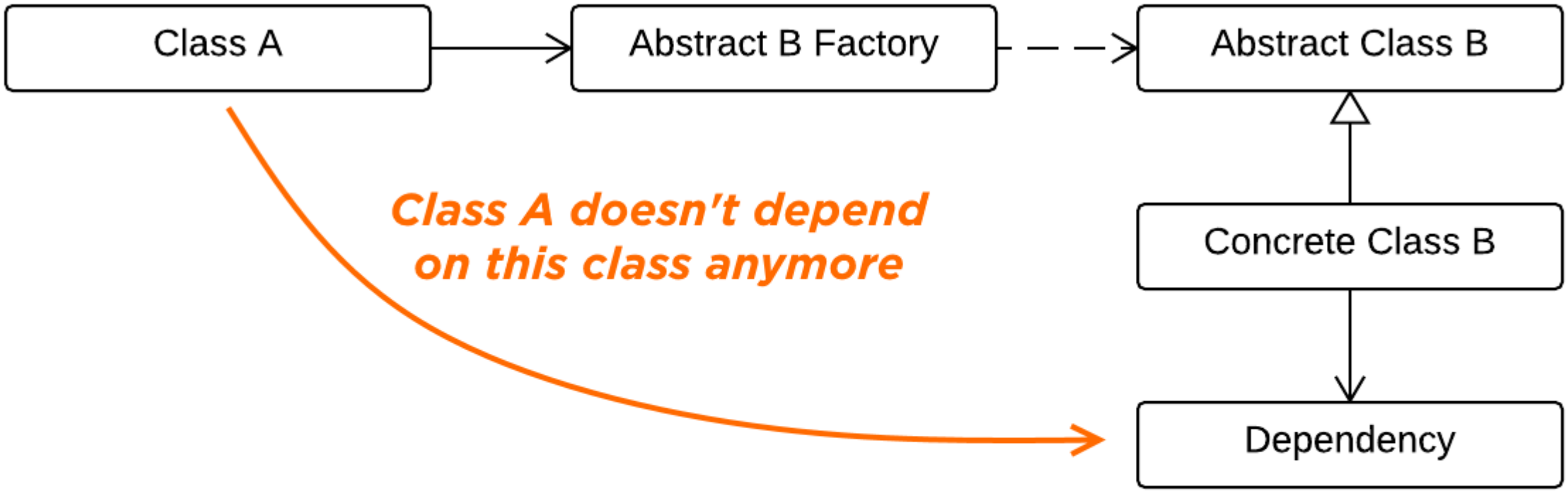
Client will be able to use the new object without fear of a failure

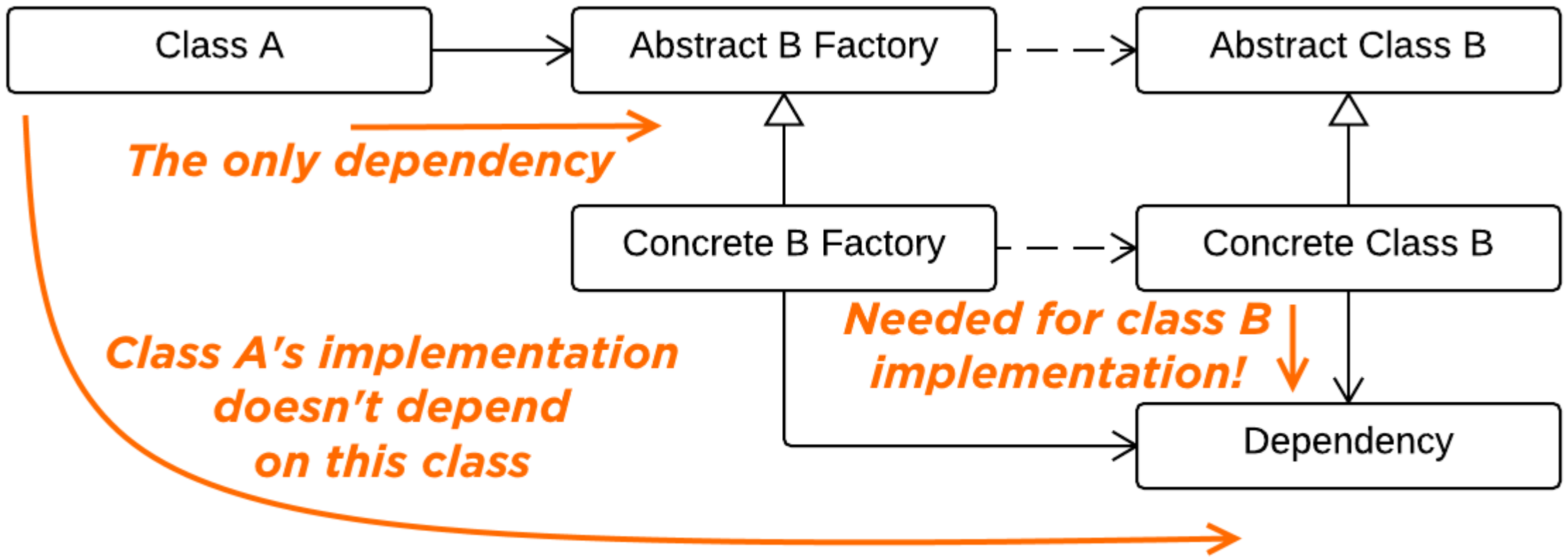
Tactical Design Patterns in .NET Series

Factory had a specific responsibility
other than just creating objects

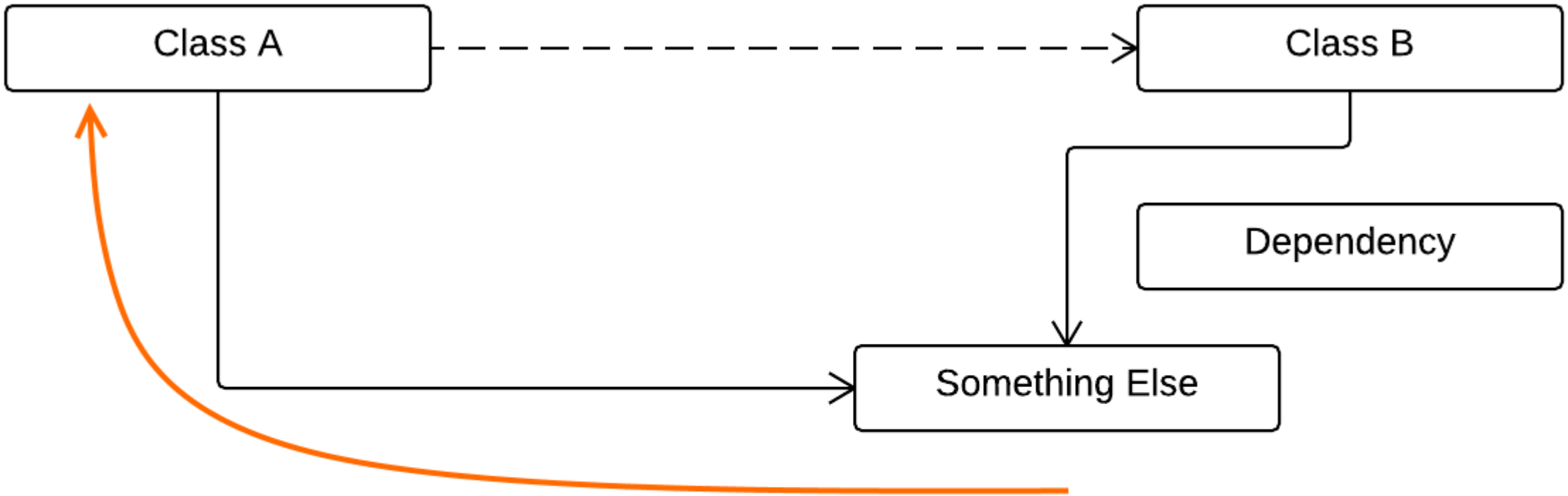

Class A
Class B
Dependency


Class A
Abstract B Factory
Abstract Class B
Synthesized abstraction

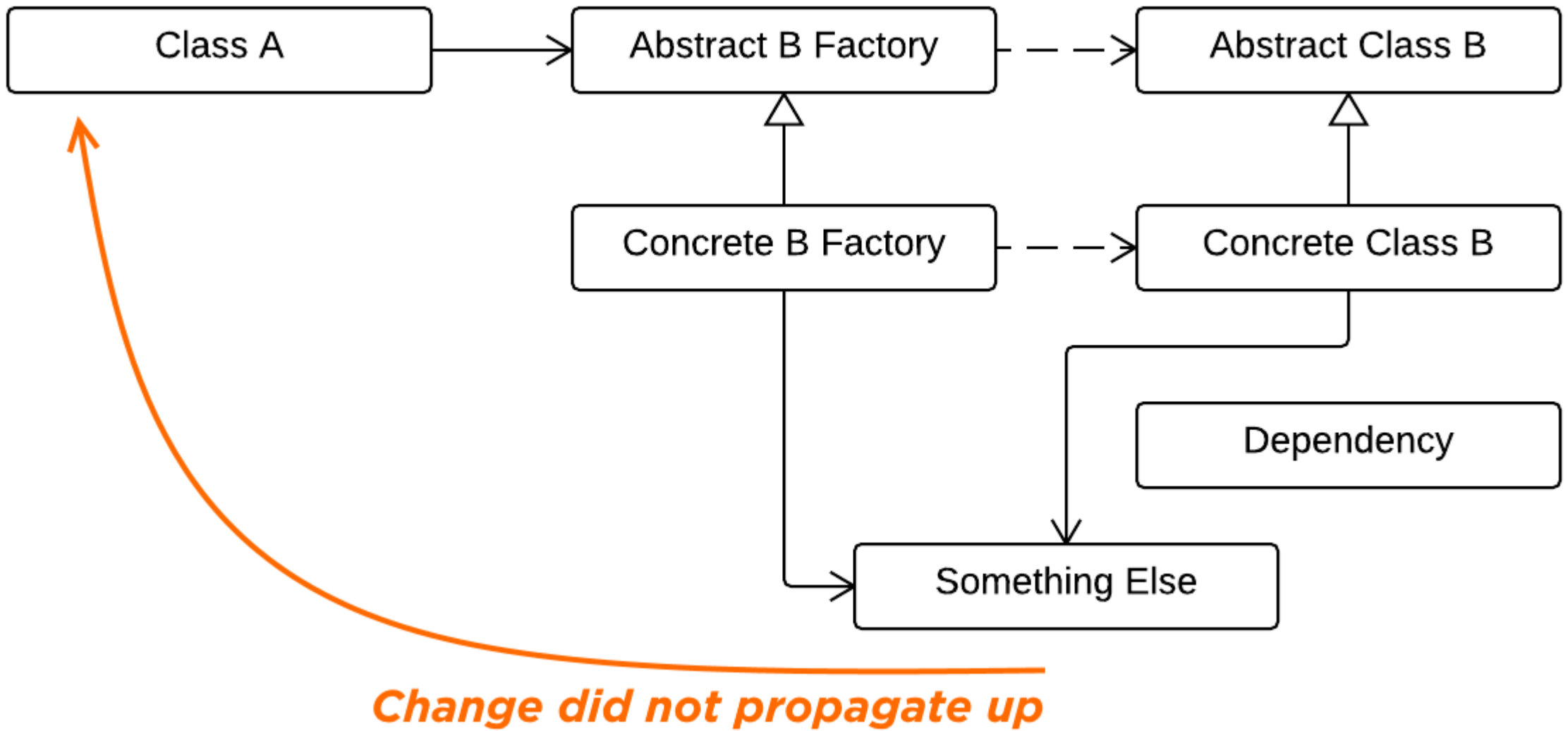

Class A
Abstract B Factory
Abstract Class B
Class A doesn't depend
on this class anymore
Concrete Class B
Dependency

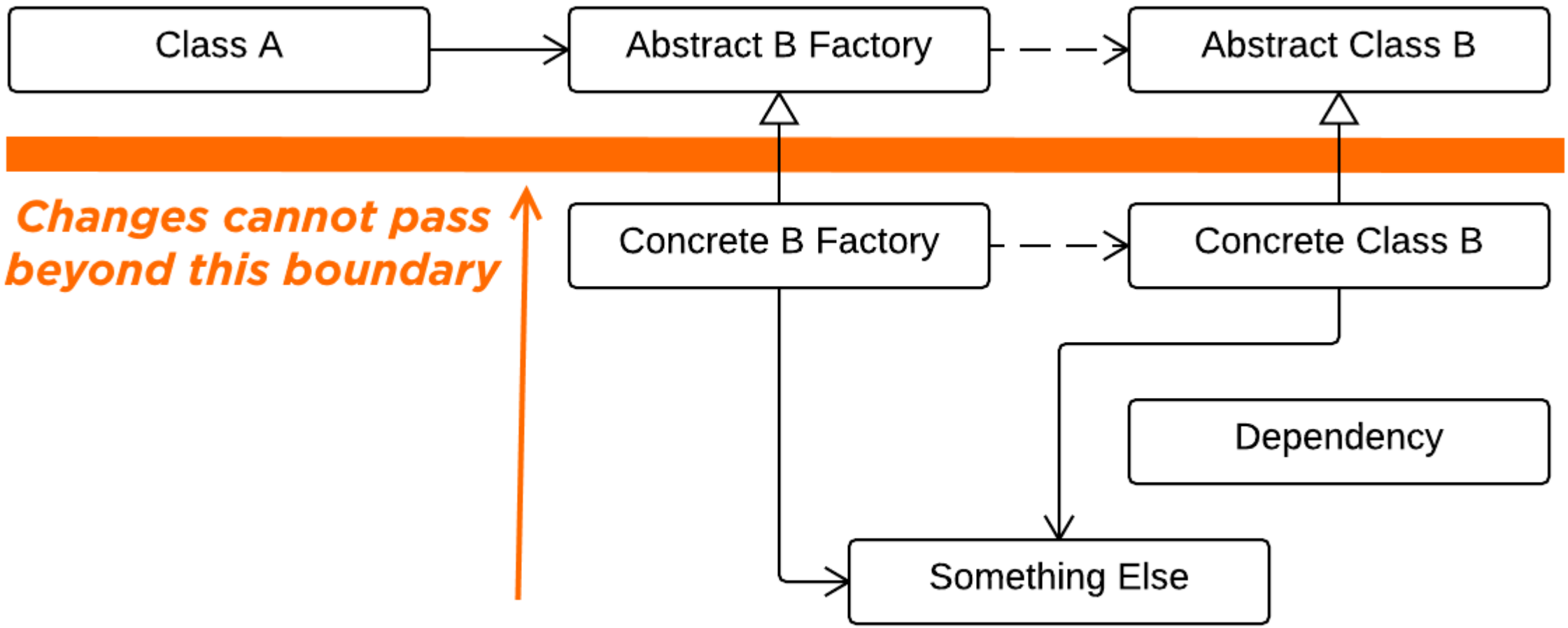

Class A
Abstract B Factory
Abstract Class B
The only dependency
Concrete B Factory
Concrete Class B
Needed for class B implementation!
Class A's implementation doesn't depend on this class
Dependency

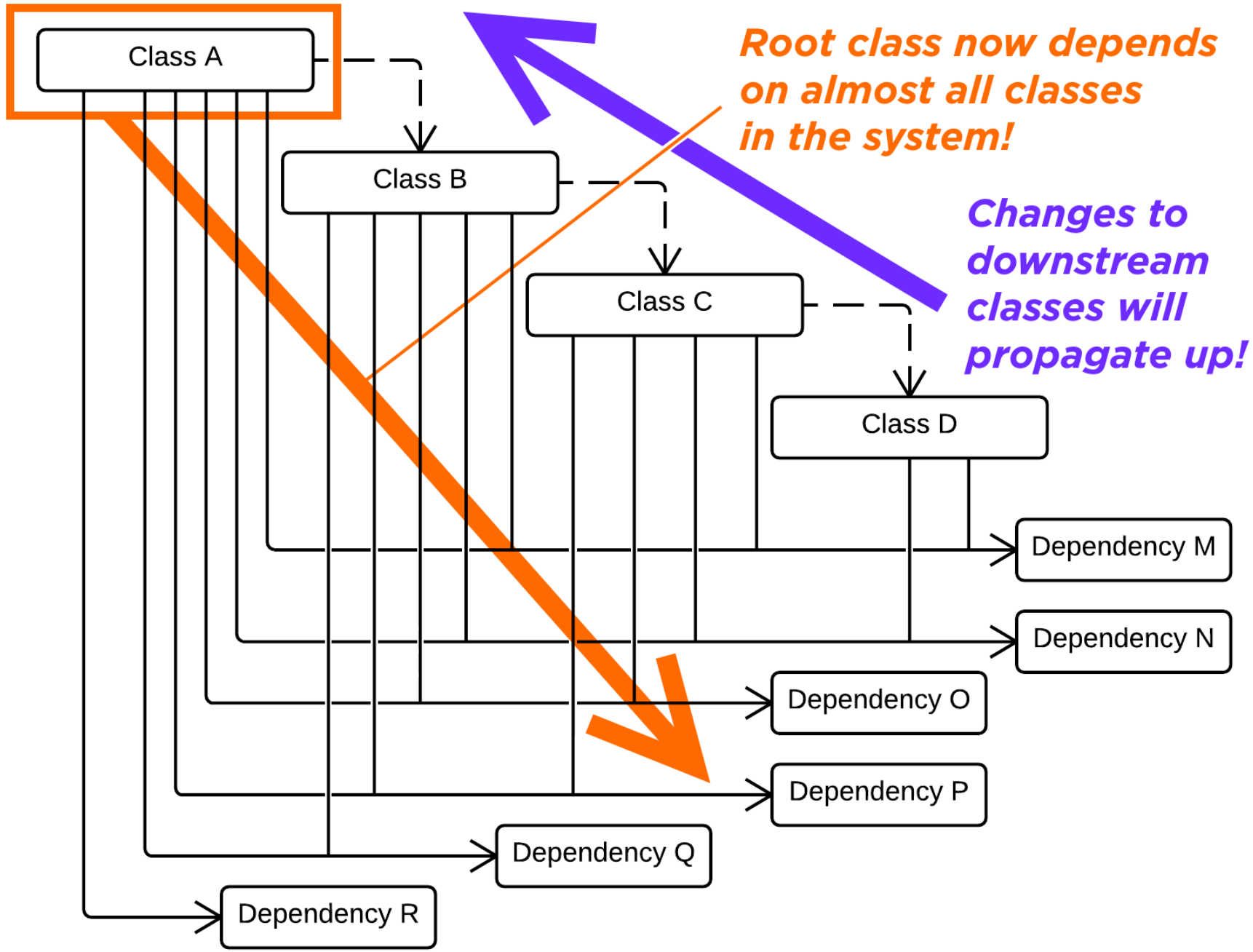
Implementation of class B
has leaked back into its creator

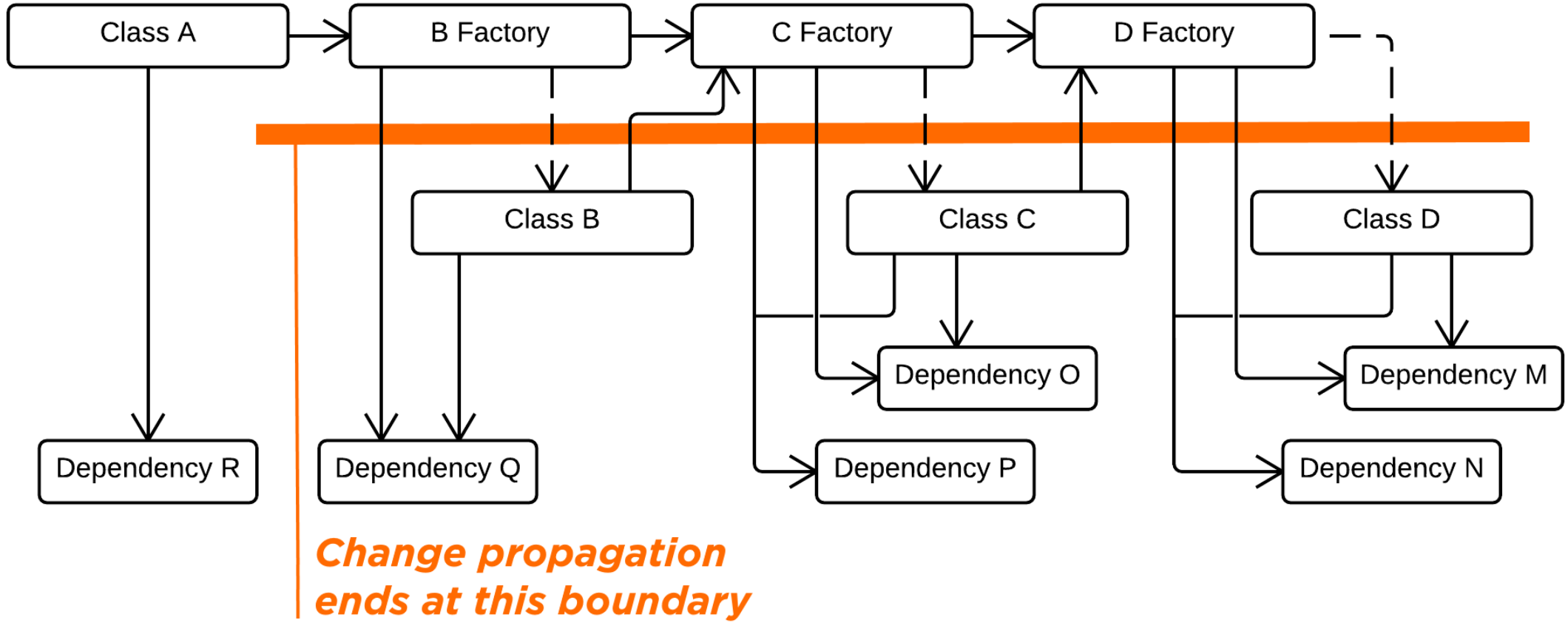


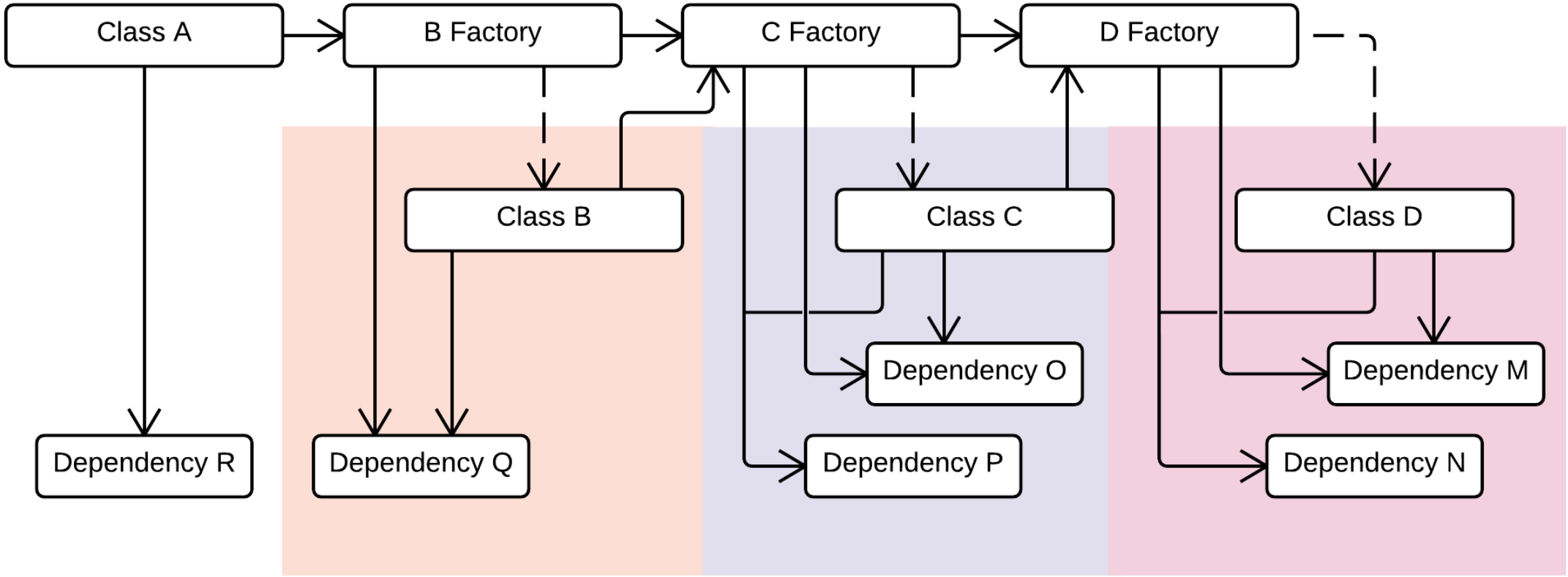
Change in implementation of class B immediately leaked up into class A


Class A
Abstract B Factory
Abstract Class B
Concrete B Factory
Concrete Class B
Dependency
Something Else
Change did not propagate up


Class A
Abstract B Factory
Abstract Class B
Changes cannot pass
beyond this boundary
Concrete B Factory
Concrete Class B
Dependency
Something Else


Class A
Class B
Class C
Class D
Dependency M
Dependency N
Dependency O
Dependency P
Dependency Q
Dependency R
Root class now depends on almost all classes in the system!
Changes to downstream classes will propagate up!


Class A
B Factory
C Factory
D Factory
Class B
Class C
Class D
Dependency R
Dependency Q
Dependency P
Dependency O
Dependency N
Dependency M
Change propagation
ends at this boundary



Classes are boxed together with their dependencies

Abstract Factory as a Dependency Breaker

Dependencies tend to leak upstream

More constructor arguments are added

They only support dependencies of dependencies

This is the symptom of implementation leakage

Introduce Abstract Factory

Close the class for future modifications caused by changing dependencies

Open for future extensions by supporting different products

Abstract Factory is reinforcing the Open/Closed Principle

Variance

Covariance

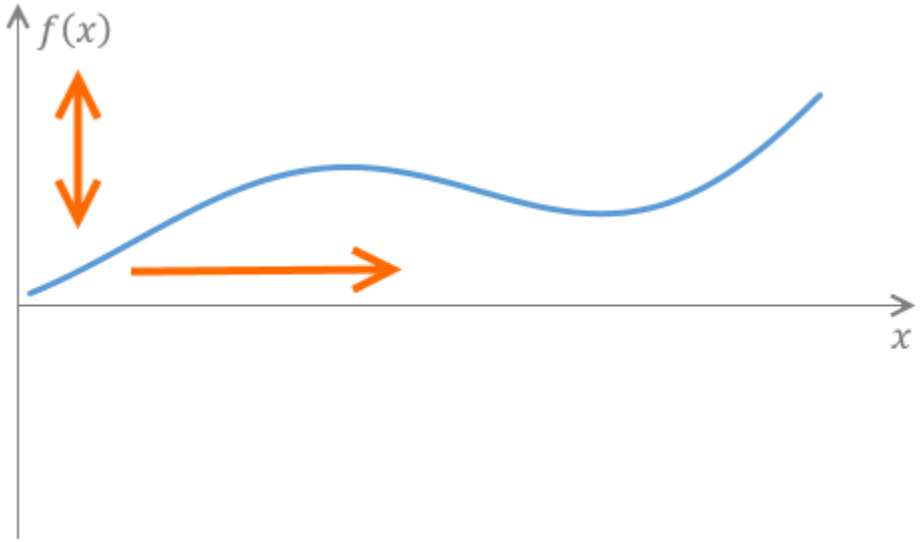
Contravariance

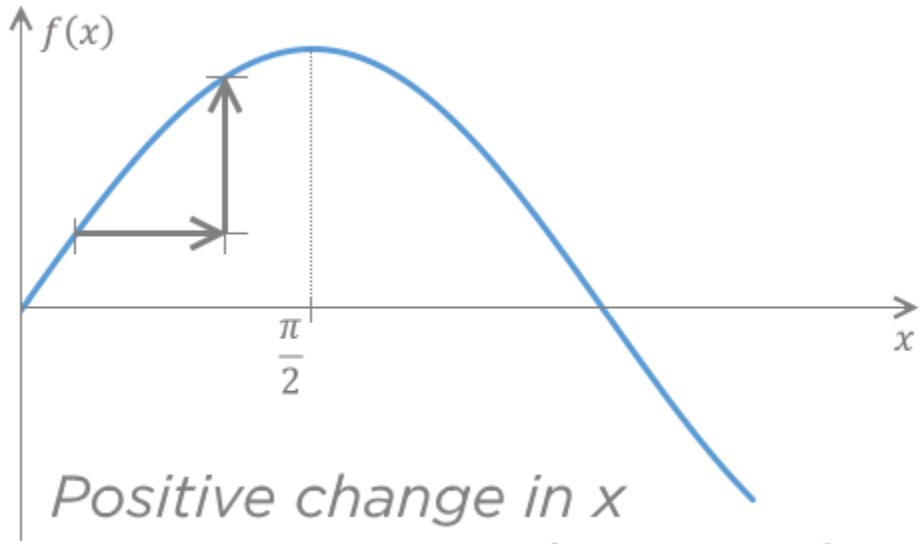
Used in:

Category theory,

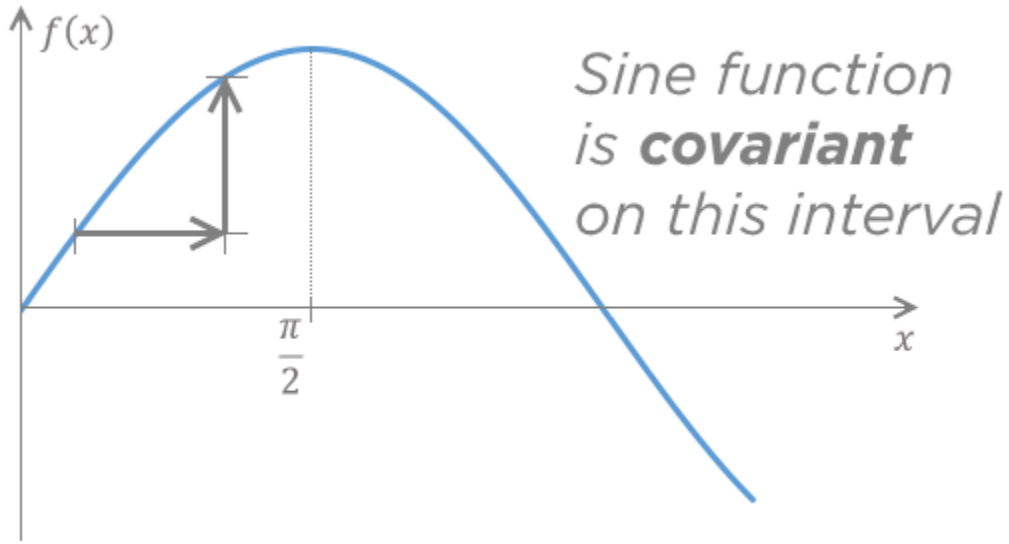
Topology,

Vectors, etc.

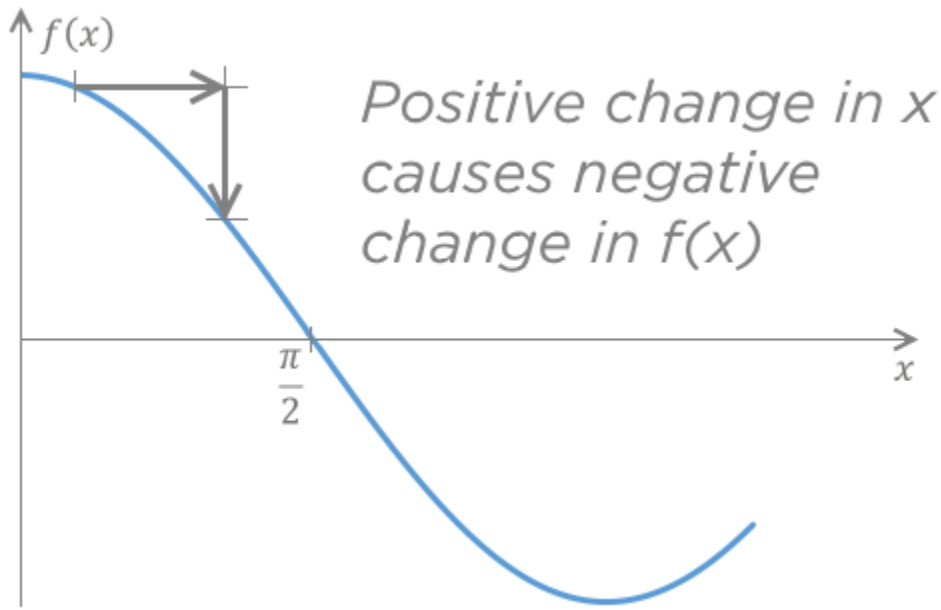

$f(x)$
x



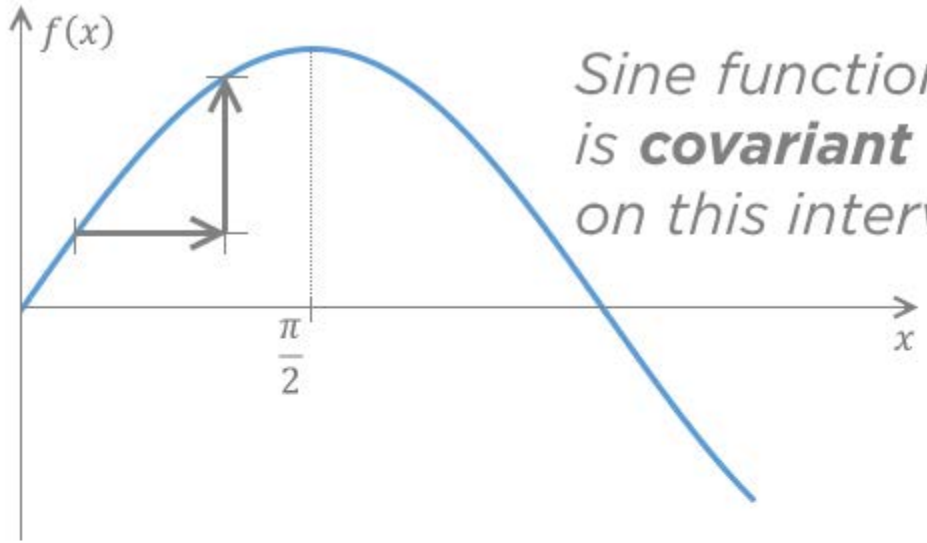
Positive change in x
causes positive change in f(x)



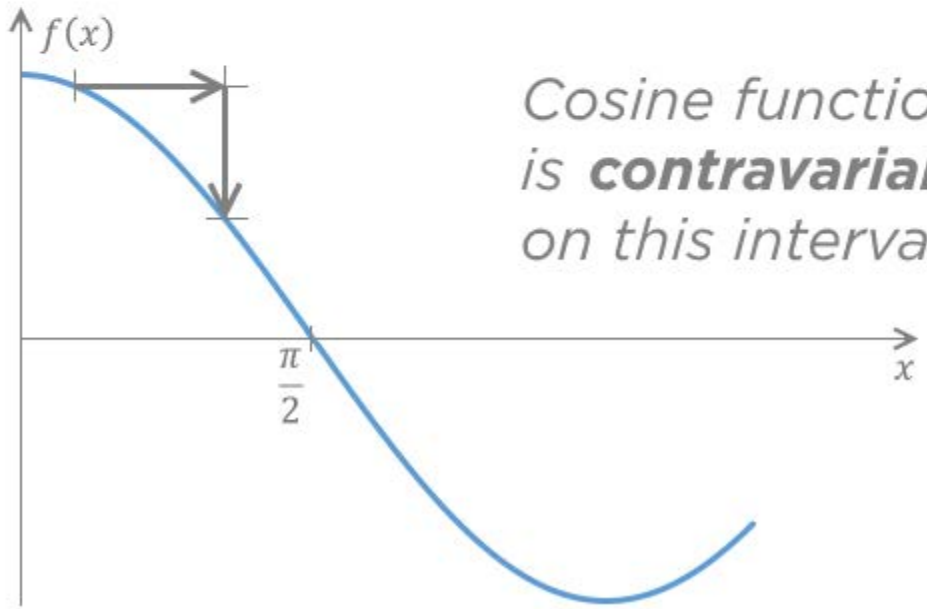
*Sine function is **covariant** on this interval*



Positive change in x causes negative change in f(x)



*Sine function is **covariant** on this interval*



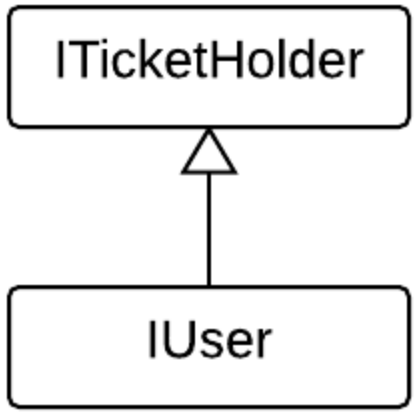
*Cosine function is **contravariant** on this interval*

FILE EDIT VIEW PROJECT BUILD DEBUG TEAM TOOLS TEST RESHARPER ANALYZE WINDOW HELP Full Screen Zoran Horvat ZH

IUserFactory.cs* Program.cs*

VarianceDemo AbstractFactoryDemo.Program RegisterUser(IUserFactory<IUser> userFactory)

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

namespace AbstractFactoryDemo
{
    class Program
    {

        static void RegisterUser(IUserFactory<IUser> userFactory)
        {
            IUserFactory<ITicketHolder> lessDerivedFactory = userFactory;
        }

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



109 %

Solution Explorer

Search Solution Explorer (Ctrl+;)

- Solution 'VarianceDemo' (1 project)
  - VarianceDemo
    - Properties
    - References
    - Factories
      - Interfaces
        - IUserFactory.cs
      - Machine
      - Person
    - Interfaces
      - ITicketHolder.cs
      - IUser.cs
      - IUserIdentity.cs
    - Models
    - App.config
    - Program.cs

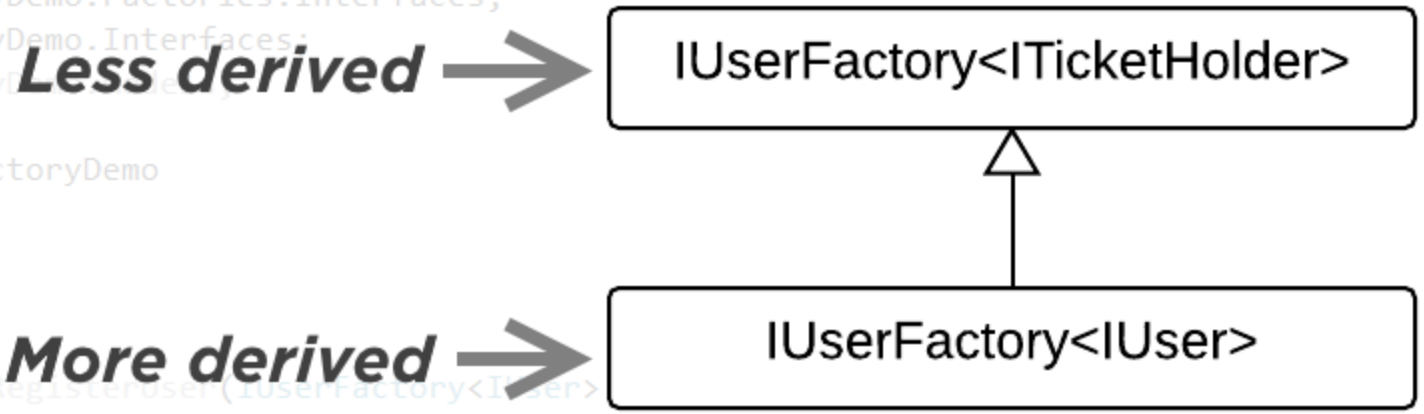
Properties Solution Explorer

Output

Show output from: General

Error List

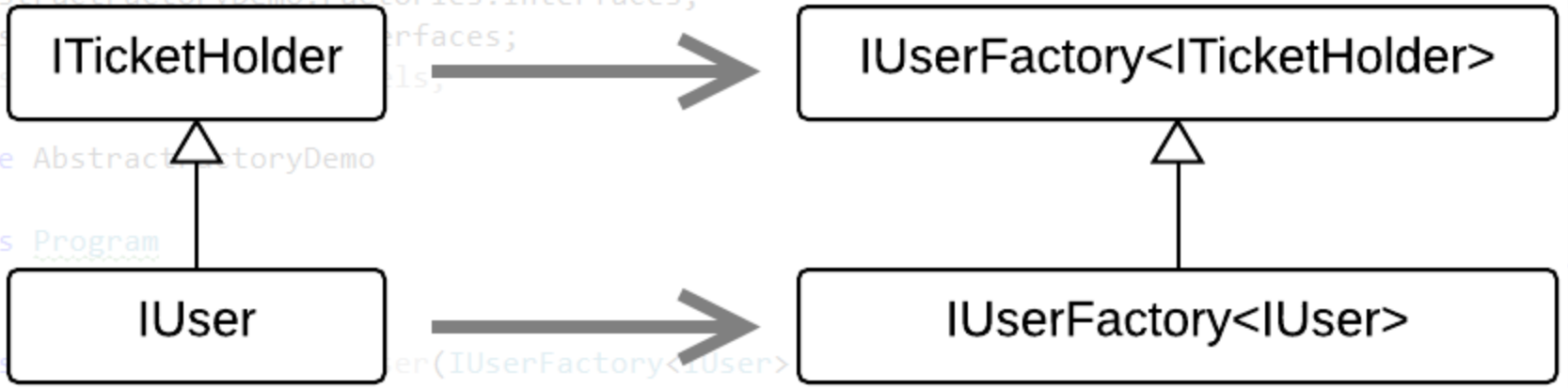
Ready 5 errors in 3 files




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

namespace AbstractFactoryDemo
{
    class Program
    {
        static void RegisterUser(IUserFactory<IUser> userFactory)
        {
            IUserFactory<ITicketHolder> lessDerivedFactory = userFactory;
        }

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

Substitution Principle:

Variable to the left
of the assignment operator
is equally or less derived
than the object to the right



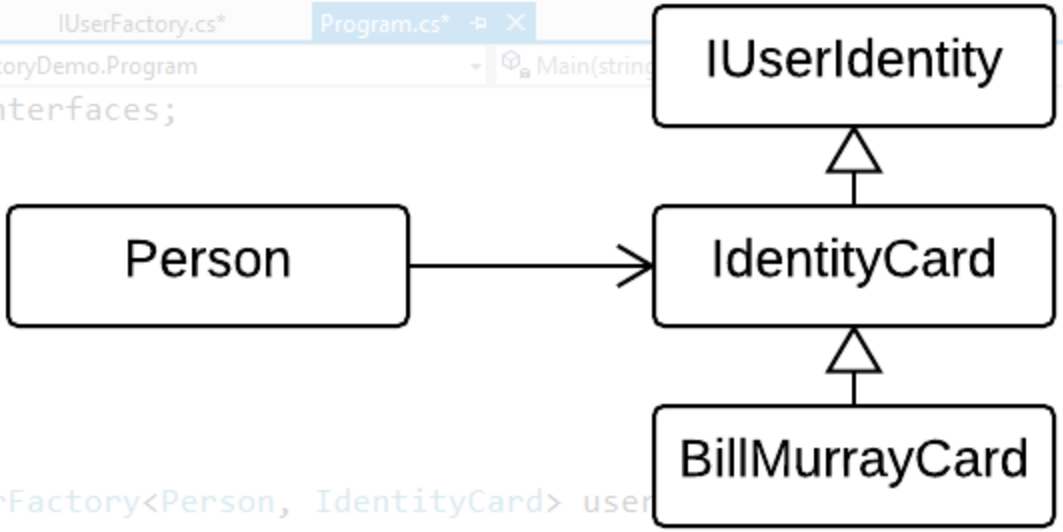

```
IUserFactory<ITicketHolder> lessDerivedFactory = userFactory;
```

Covariance in generic types:

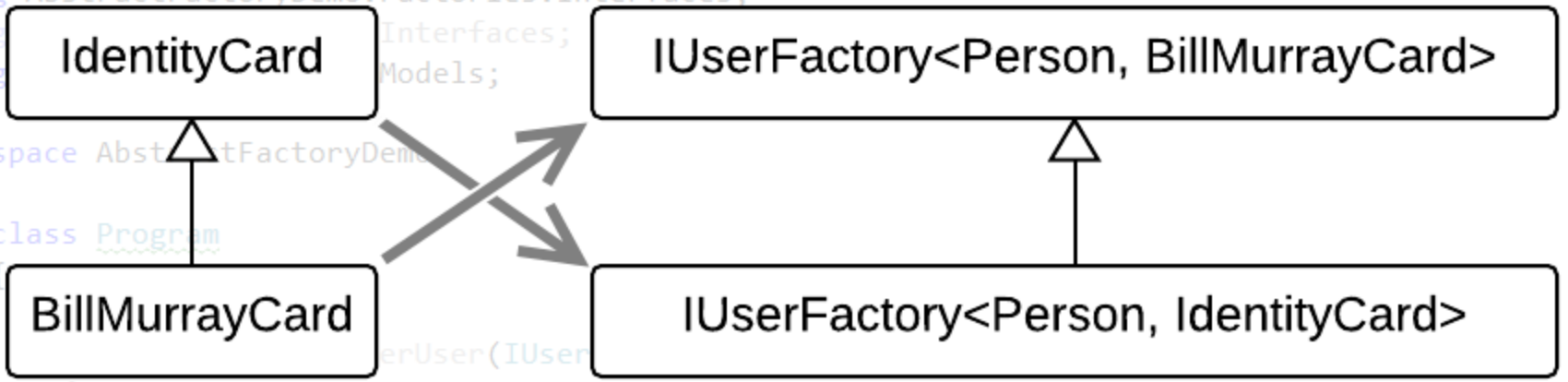
Less derived generic type argument produces less derived generic type

More derived generic type argument produces more derived generic type

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

namespace AbstractFactoryDemo
{
    class Program
    {

        static void RegisterUser(IUser
        {
            IUserFactory<Person, BillMurrayCard> lessDerivedFactory = userFactory;
        }

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




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

namespace AbstractFactoryDemo
{
    class Program
    {

        static void RegisterUser(IUserFactory<Person, IdentityCard> user
        {
            IUserFactory<Person, BillMurrayCard> lessDerivedFactory = userFactory;
        }

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



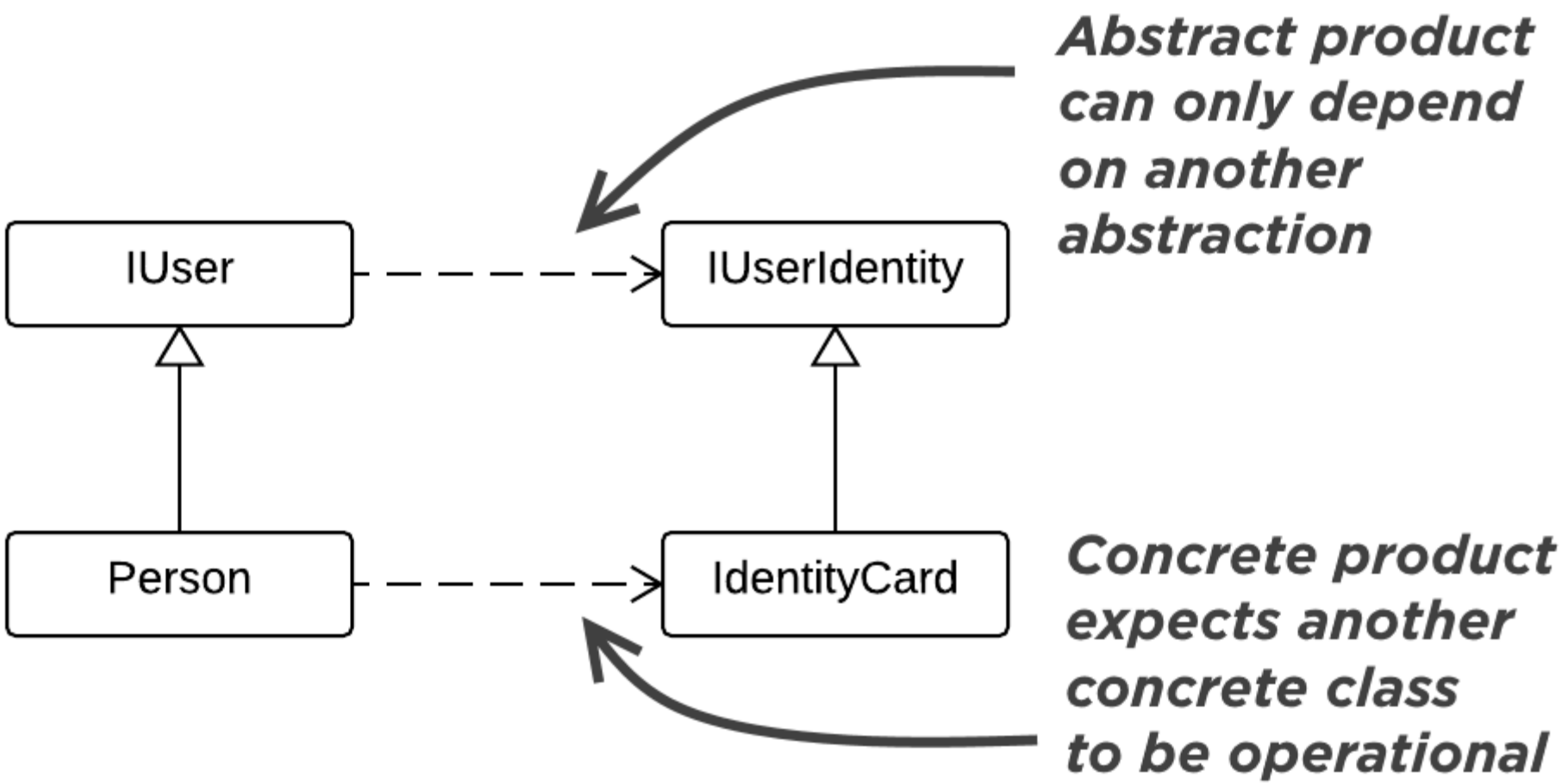

```
using AbstractFactoryDemo.Factories.Interfaces;
using                                  Interfaces;
using                                  Models;

namespace AbstractFactoryDem
{
    class Program
    {
                                  erUser(IUser
        {
            IUserFactory<Person, BillMurrayCard> lessDerivedFactory = userFactory;
        }

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

Contravariance in generic types:

Less derived generic type argument produces more derived generic type

More derived generic type argument produces less derived generic type

Covariance-Contravariance Clash



Covariance-Contravariance Clash

Abstract Factory

Cannot resolve the issue

Builder

Addresses the issue by hiding abstract types

Specification

Same as the Builder

Summary

Class dependencies

- Abstract Factory helps isolate dependencies

Covariance and contravariance

- Apply to generic types in object-oriented languages
- Objects look like being *covariant* on types their methods return
- Objects look like being *contravariant* on types of method arguments

Summary

The problem of Abstract Factory

- It is covariant on types of its concrete products
- Concrete products are contravariant on types of method arguments

This can become a serious problem

- It appears in cases when one product depends on another

Covariance-contravariance mismatch

- It will always be there
- Significant part of designing goes into working around this problem

Summary

More theory on the horizon...

Substitution Principle

- It tells which types can stand in place of which other types
- Ensures *syntactical* correctness

Liskov Substitution Principle (LSP)

- Extends the Substitution Principle
- Defines constraints on substitutes
- Ensures *semantical* correctness

Summary

Connection to covariance/contravariance

- Substitution Principle resembles covariance
- Liskov Substitution Principle resembles contravariance

Next module –

Applying the Substitution and Liskov Substitution Principles