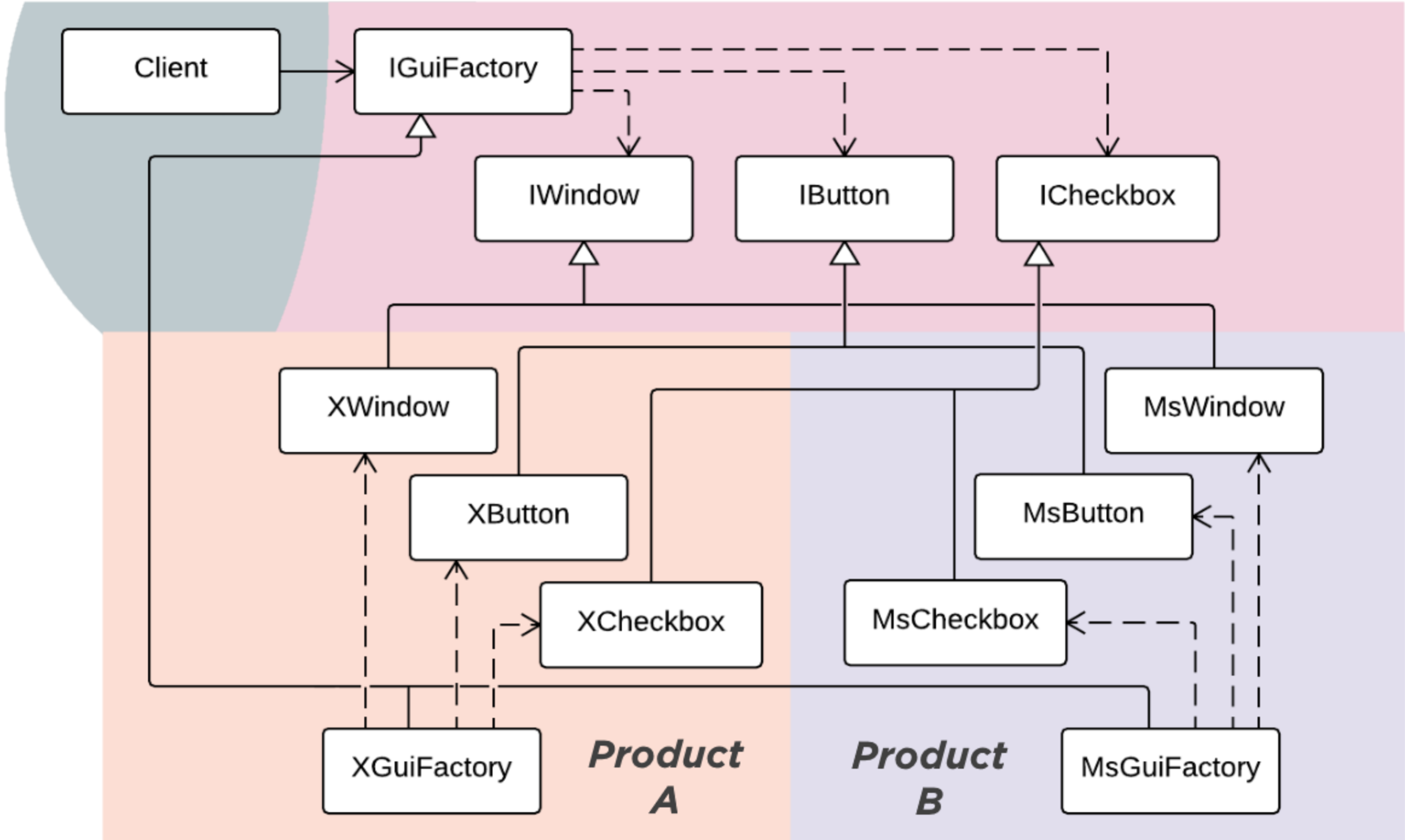
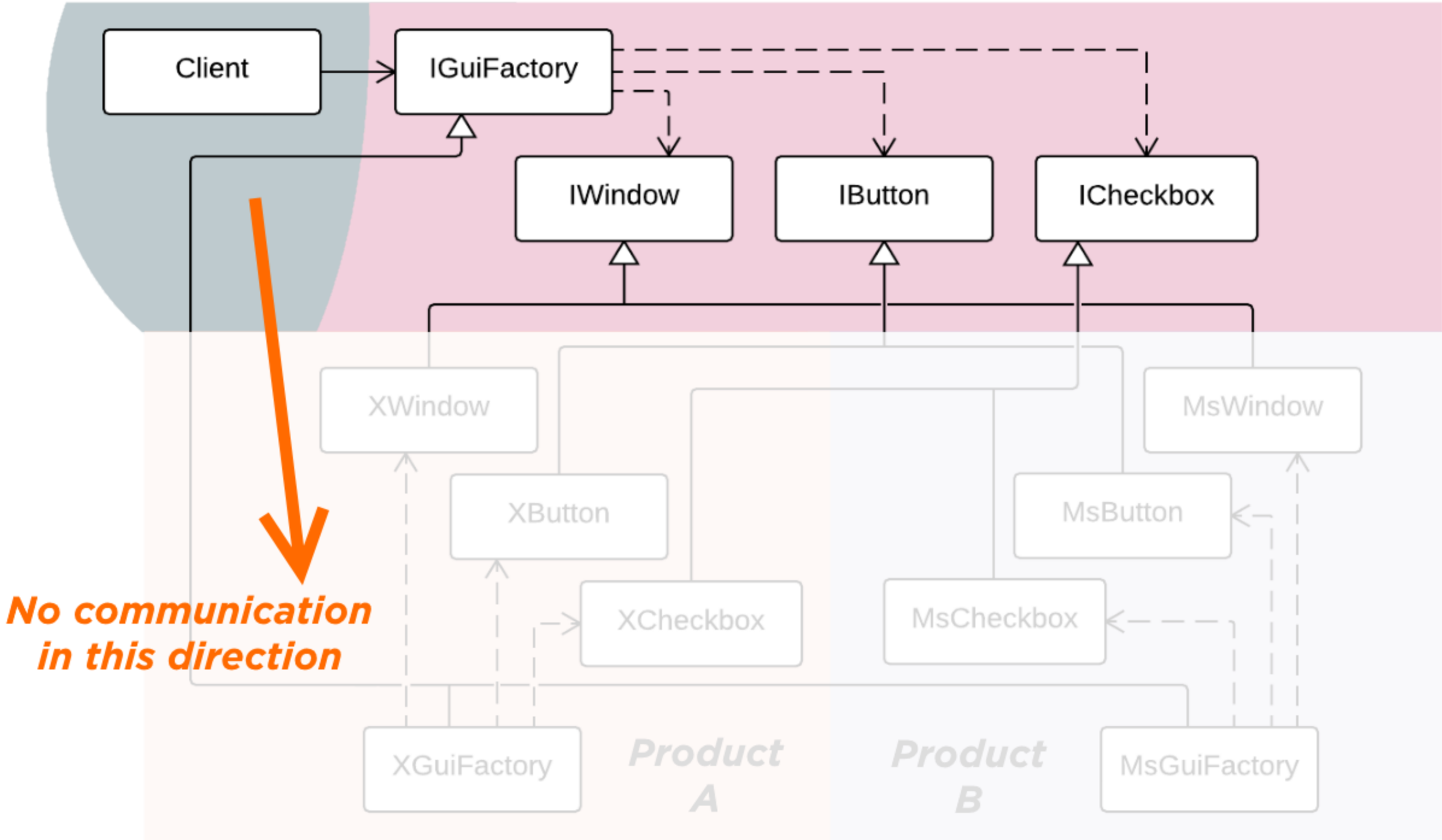
Avoiding Excess Factory Abstractness

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


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


Client
IGuiFactory
IWindow
IButton
ICheckbox
XWindow
MsWindow
XButton
MsButton
XCheckbox
MsCheckbox
XGuiFactory
MsGuiFactory
Product A
Product B
No communication in this direction

Complications Inherent to Abstract Factory

Abstract layer only contains features common to all concrete products

Some concrete products are yet to be discovered in the future

Arguments to Create method must satisfy all concrete products

Concrete Products May Depend on Each Other

```
public interface IUser
{
    void SetIdentity(IUserIdentity identity);
}
```

Abstract view

```
public class Machine : IUser
{
    public void SetIdentity(MacAddress identity)
    {
        // ...
    }
}
```

```
public class Person : IUser
{
    public void SetIdentity(IdentityCard identity)
    {
        // ...
    }
}
```

Concrete view

Abstract Factory Trade-offs

Conformist approach

Make method arguments more abstract

Accept strings instead of concrete types

Stringly-typed solution may fail at runtime!

Escapist approach

Support only one family of concrete products

Don’t try to find what different families have in common

Move some concrete features into abstract part of the diagram

Practical Problems with Abstract Factory

Fixing one complication makes another grow bigger

Conclusion is to seek for the middle solution in each practical case

Trade-off

strained naturally in some case?

trade type safety for it!

```
public interface IControllerFactory
{
    IController CreateController(RequestContext requestContext, string controllerName);
    void ReleaseController(IController controller);
}
```

```
public interface IControllerFactory
{
    IController CreateController(RequestContext requestContext, string controllerName);
    void ReleaseController(IController controller);
}
```

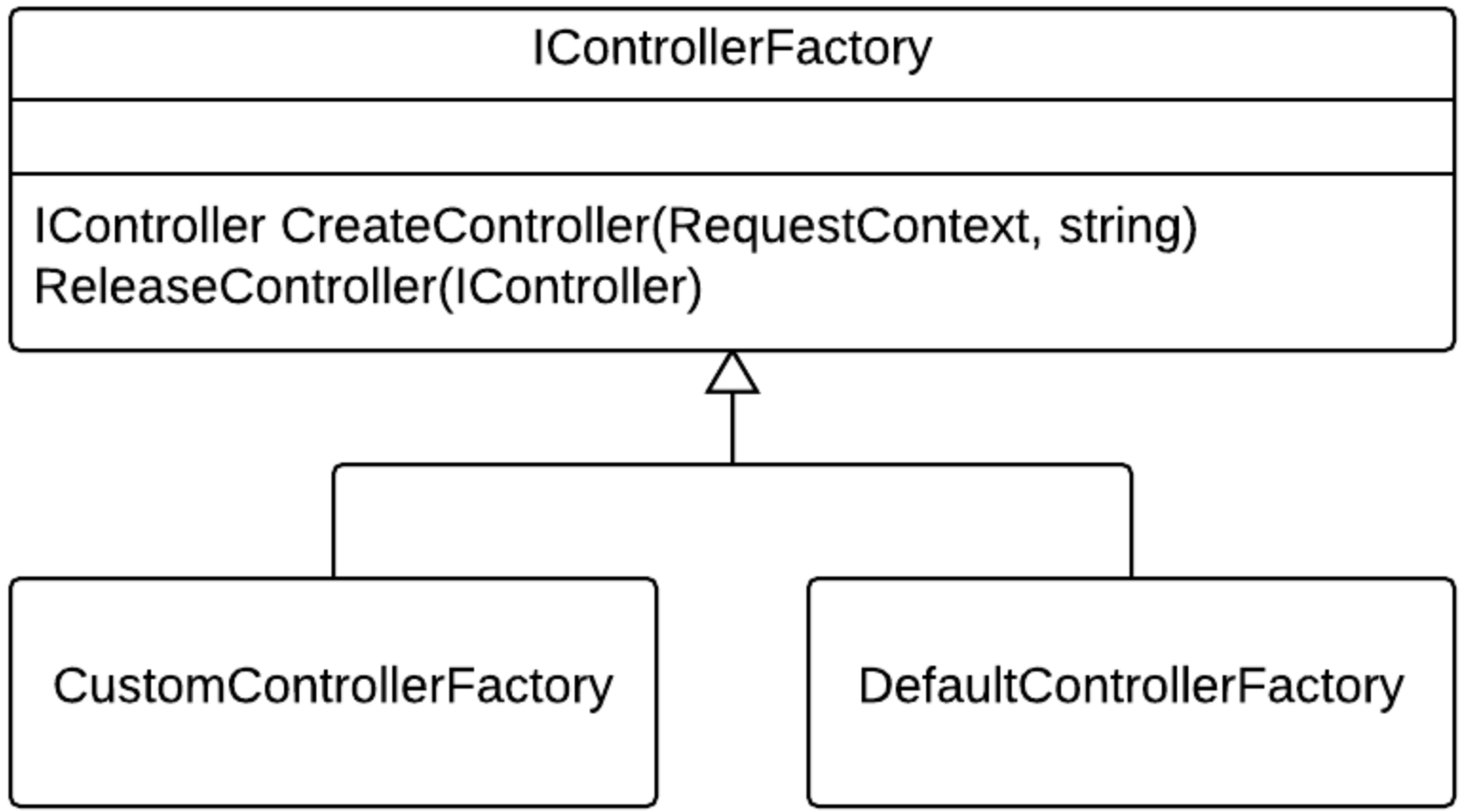
Context is tightly bound to Web request

String used to support future classes

This abstract factory now supports only one family of concrete products

Multiplatform implementation is almost impossible even for this one family of products

```
public interface IControllerFactory
{
    IController CreateController(RequestContext requestContext, string controllerName);
    void ReleaseController(IController controller);
}
```



When we rely on strings...

We cannot stop execution early when string is incorrect

String is passed to deeper layers for further analysis

The invalid string can even reach the Domain Model

Domain Model finally fails, but that is way too late

When we rely on strings...

We loose compile-time checks

We loose run-time checks at early stages

We can just fail deep inside the application

HTTP Status Code in the Controller Factory?

Factory is doing it's caller's job

Factory attempts to give clues about how to recover from an error

Factory is sending potentially confusing message

It has analyzed the request

It has found the error

And it advices the caller how to recover

Error message sent from the Factory is too specific

What if it shouldn't be status 404 in this case?

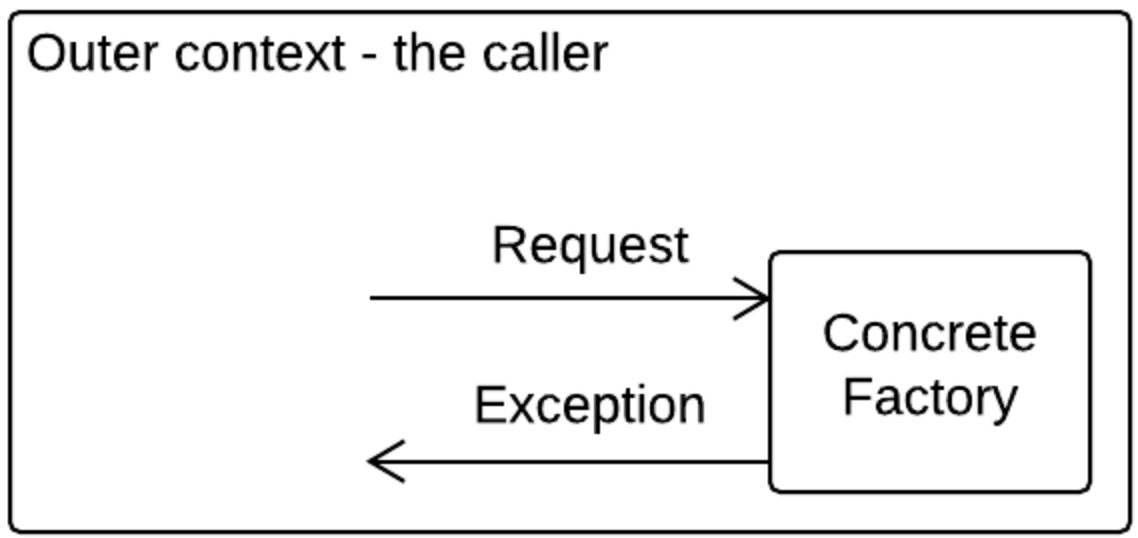
What if it was the Internal Server Error, status 500 in the end?

```
public IController CreateController(RequestContext requestContext, string controllerName)
{
    if (controllerName != "Home")
        throw new HttpException((int)HttpStatusCode.NotFound, "Controller not found.");

    return new HomeController(
        new AppSignature()
        {
            ApplicationName = "Concrete Factory Demo",
            AuthorName = "Zoran Horvat"
        });
}
```

```
public IController CreateController(RequestContext requestContext, string controllerName)
{
    if (controllerName != "Home")
        throw new HttpException((int)HttpStatusCode.NotFound, "Controller not found.");

    return new HomeController(
        new AppSignature()
        {
            ApplicationName = "Concrete Factory Demo",
            AuthorName = "Zoran Horvat"
        });
}
```

HTTP status 404

Is this just a lucky guess?



Division of Responsibilities

Caller

Callee

```
if (controllerName != "Home")
    throw new HttpException((int)HttpStatusCode.NotFound, "Controller not found.");
```

Throw ArgumentException instead?

Consequences of Using Strings Instead of Concrete Objects

Stringly-typed coding

Practice of passing a string when we don't have a more appropriate type

Abstract Factory often forces us to pass strings

There is no common type system for different product families

Potential solution

Use Builder or Specification instead

These will hide concrete objects from the caller through encapsulation

No more strings will be required

Summary

Abstract Factory applied to a single family of concrete products

- This is opposed to trying to unite different families of products

Positive consequences

- Some concrete features can be moved up the hierarchy into abstract types
- No fear that some derived product will suffer
  - There are no other derived products!

Summary

Other consequences

- There will be only one constructor
- No need to look for common constructor arguments
- The design is simpler and easier to use

Summary

An example

- Controller factory in ASP.NET MVC
- The framework supports only one family of controllers
- Create method arguments were meaningful to this family of products

Stringly-typed code

- Controller was identified by a string
- This has caused instability

Summary

Error recovery from a concrete factory

- It was not a good idea to recover from feature supplier
- Recovery may require more information
- Called object has no such information
- The only one who may have it is the caller

Summary

Then how to use Abstract Factory well?

- By understanding the root cause of the issues first!

Root causes:

- Class dependencies
- Substitution principle
- Liskov substitution principle
- Method preconditions

Next module –
Understanding Dependencies, Covariance and Contravariance