

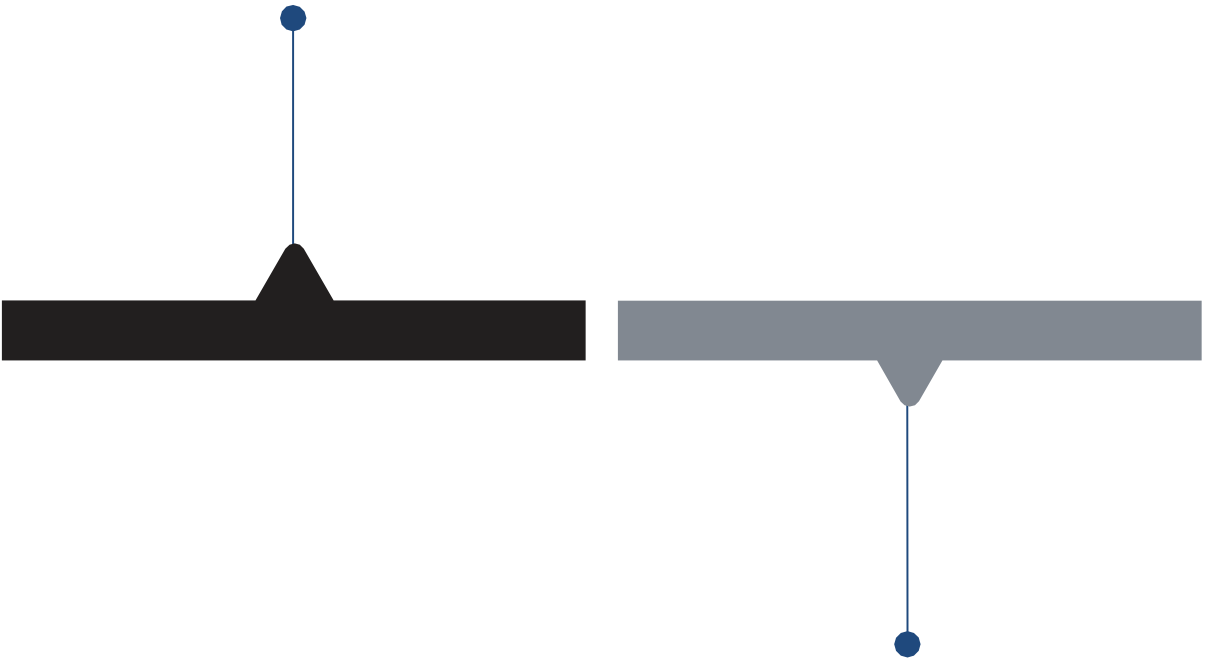
OBJECT ORIENTED PROGRAMMING IN C# AND .NET CORE

Object Creation: From Classical Design Patterns to Dependency Injection

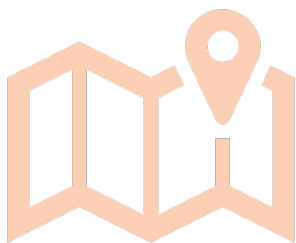
Design Patterns

Design Patterns

The What



The Why



What to expect from this course?

Introduction to Design Patterns idea

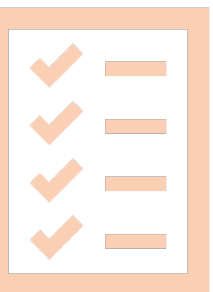
Understand the importance of polymorphism

Why constructing objects can be a problem

Main Creational Design Patterns

Understand Inversion of Dependencies and Dependency Injection, with examples in MVC Core

Understand Repository and Unit Of Work patterns with examples in MVC Core.



Design Patterns



Idea borrowed from “real architecture” concept by Christopher Alexander (1977/79).



Reusable design solution to recurrent problems



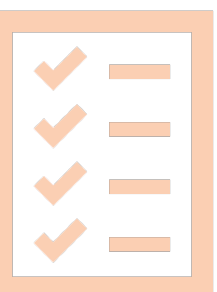
Motivation, Structure, Implementation, Sample Code



High level vocabulary







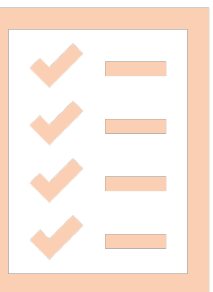
Watch out for “patternitis”







Design Principles

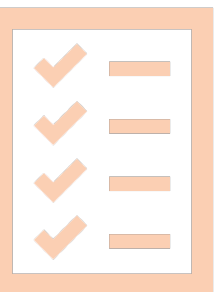
Design Principles

-  **Single Responsibility:** a class should have only a single responsibility (cohesion)
-  **Open Closed:** code should be open for extension, closed for modification
-  **Liskov Substitution:** objects should be replaceable with instances of their subtypes without altering the correctness of the program. (no, it's not theoretical physics..)
-  **Interface Segregation:** prefer many client-specific interfaces over one general-purpose interface




Design Principles

-  **Dependency Inversion:** High-level modules should not depend on low-level modules. Both should depend on abstractions.
-  **DRY:** Avoid duplicate code, abstract out common things to a single location.
-  **Loose Coupling:** Strive for loosely coupled designs of autonomous, interacting objects
-  **Law of Demeter (principle of least knowledge) :** Only talk to your immediate friends.



Design Principles

 **Encapsulate what varies:** find our code section that are likely to change, and move them in their own method or class. For more extensibility, make those method virtual and those classes members of a family of classes. You will see this in action in the factory idiom and patterns



Design Principles



Code to interfaces, not implementations: rely on abstractions for your references,

as much as you can. Es. make functions like

```
public IEnumerable<Employee> GetAll()
```

```
public void Handle(ICollection<Account> accounts),
```

not

```
public Employee[] GetAll()
```

```
void Handle (List<Account> accounts).
```



Design

Principles



Prefer composition over inheritance: “Has a” can often be better than “Is a”:

inheritance is a powerful tool, but also a dangerous one, as noted also by James Gosling. Implementation inheritance is useful but it is also the strongest type of coupling you can have between two classes. Deep or large implementation inheritance families can cause huge “ripple effects” in your code and make it fragile and hard to extend and modify. You can reuse code by composition, with less coupling and the added benefit of run time flexibility if you compose with an interface reference.



Extensibility And Maintenance

Extensibility And Maintenance



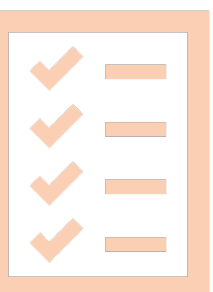
OOP main goal: easy maintenance and extensibility



Obtained through polymorphism: encapsulate what varies + code to an interface + composition + Liskov



That 's how you get Open Closed





The Problem With Constructors

The Problem With Constructors



Coding to an interface is all well and good until you have to do:

```
Abstraction x = new Implementation();
```

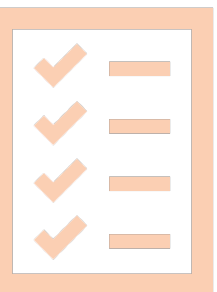


With direct instantiation through constructor you are kissing goodbye to:

Coding to an interface. Your references might be general and immutable but your code is littered with tons of implementation choices.



Single Responsibility: Your class is now also responsible for creating other objects



The Problem With Constructors



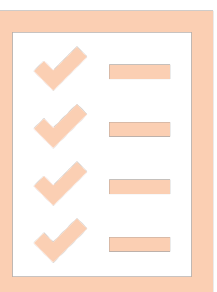
DRY: You repeat over and over the same pattern of creation for objects and their dependencies.



Dependency Inversion: Higher level modules end up depending on low level modules

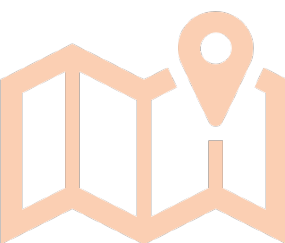
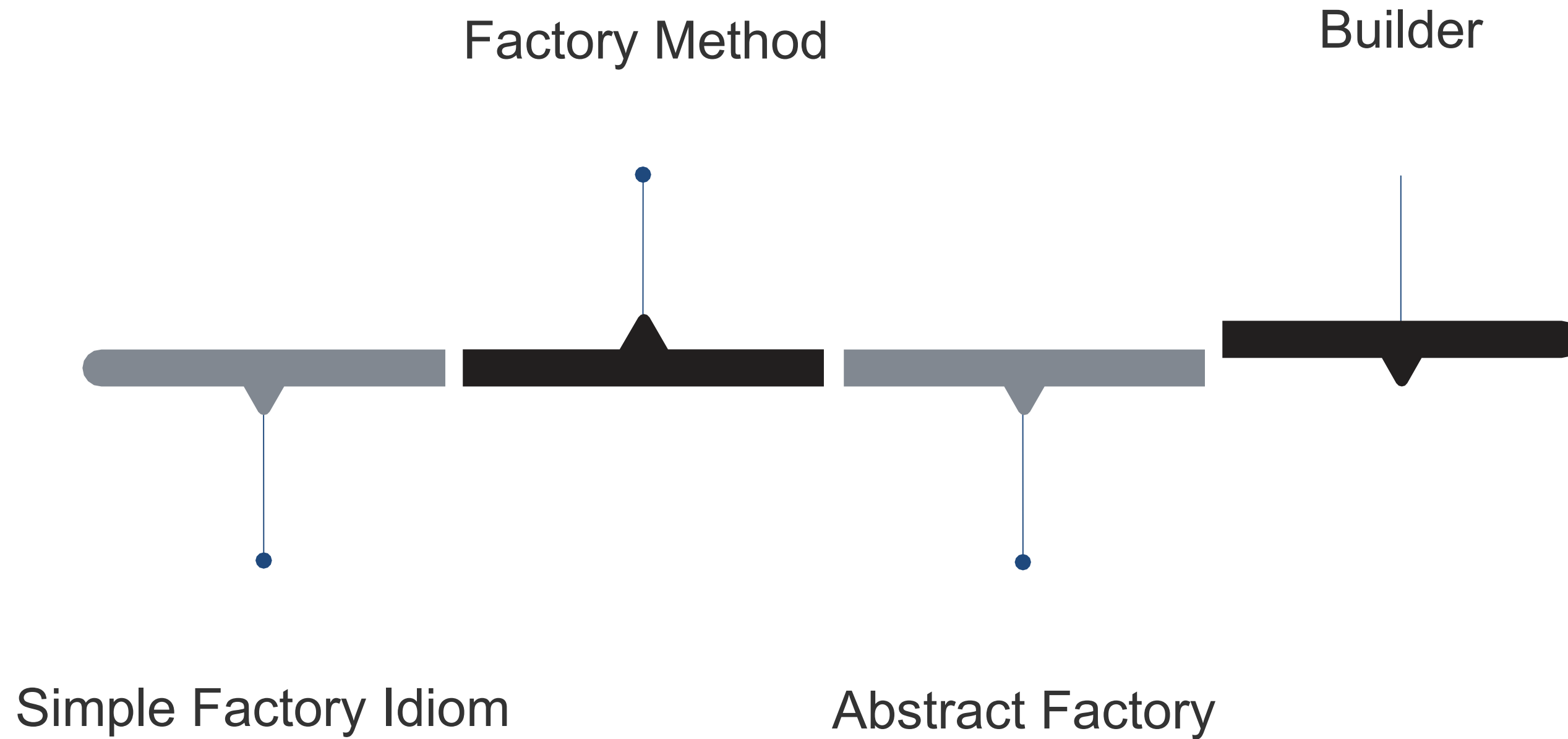


To sum it up, direct instantiation forces you to abandon “Riccardo Matrix Principle”: Ignorance (of the concrete type) is bliss.

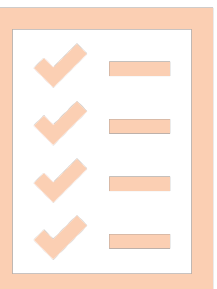
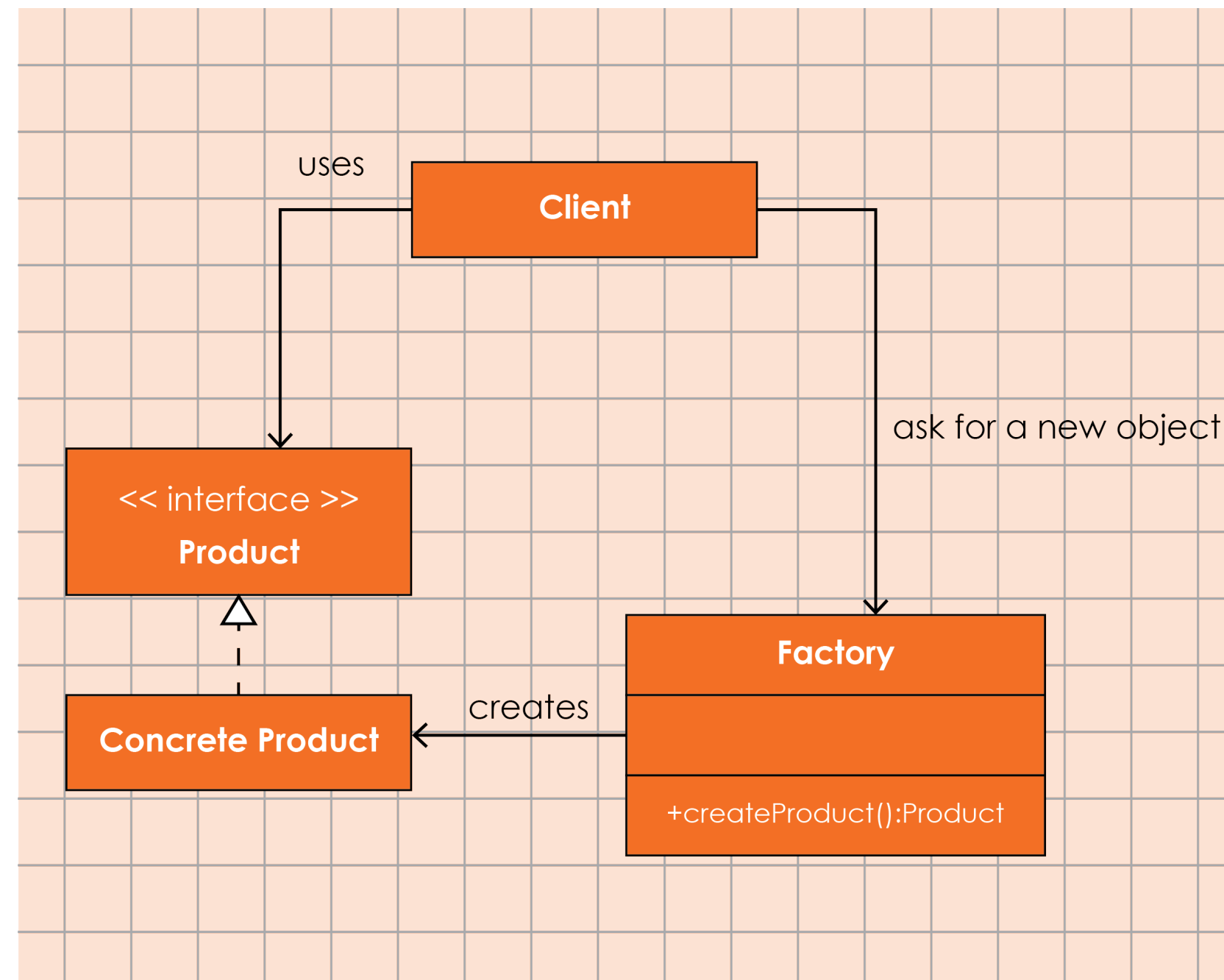


Design Patterns To The Rescue:

Design Patterns To The Rescue:



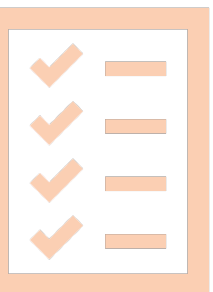
Simple Factory Idiom: UML



Simple Factory Idiom : Code Sample

```
public class SimplePizzaFactory
{
    //encapsulate what changes
    //single responsibility principle

    public static Pizza CreatePizza(PizzaType type)
    {
        // not really open closed principle!
        switch(type)
        {
            case PizzaType.margherita:
                return new Margherita();
            case PizzaType.capricciosa:
                return new Capricciosa();
            default:
                throw new ArgumentException("pizza type not permitted");
        }
    }
}
```

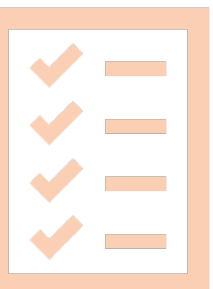


Simple Factory Idiom: Test

Usage

```
[Fact]
public void CreatePizza()
{
    Pizza pizza = SimplePizzaFactory.CreatePizza(PizzaType.capricciosa);
    Assert.IsType<Capricciosa>(pizza);

    pizza = SimplePizzaFactory.CreatePizza(PizzaType.margherita);
    Assert.IsType<Margherita>(pizza);
}
```



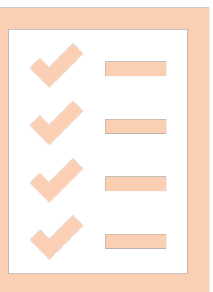
Factory Method: Definition

Factory Method: Definition

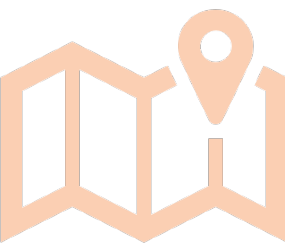
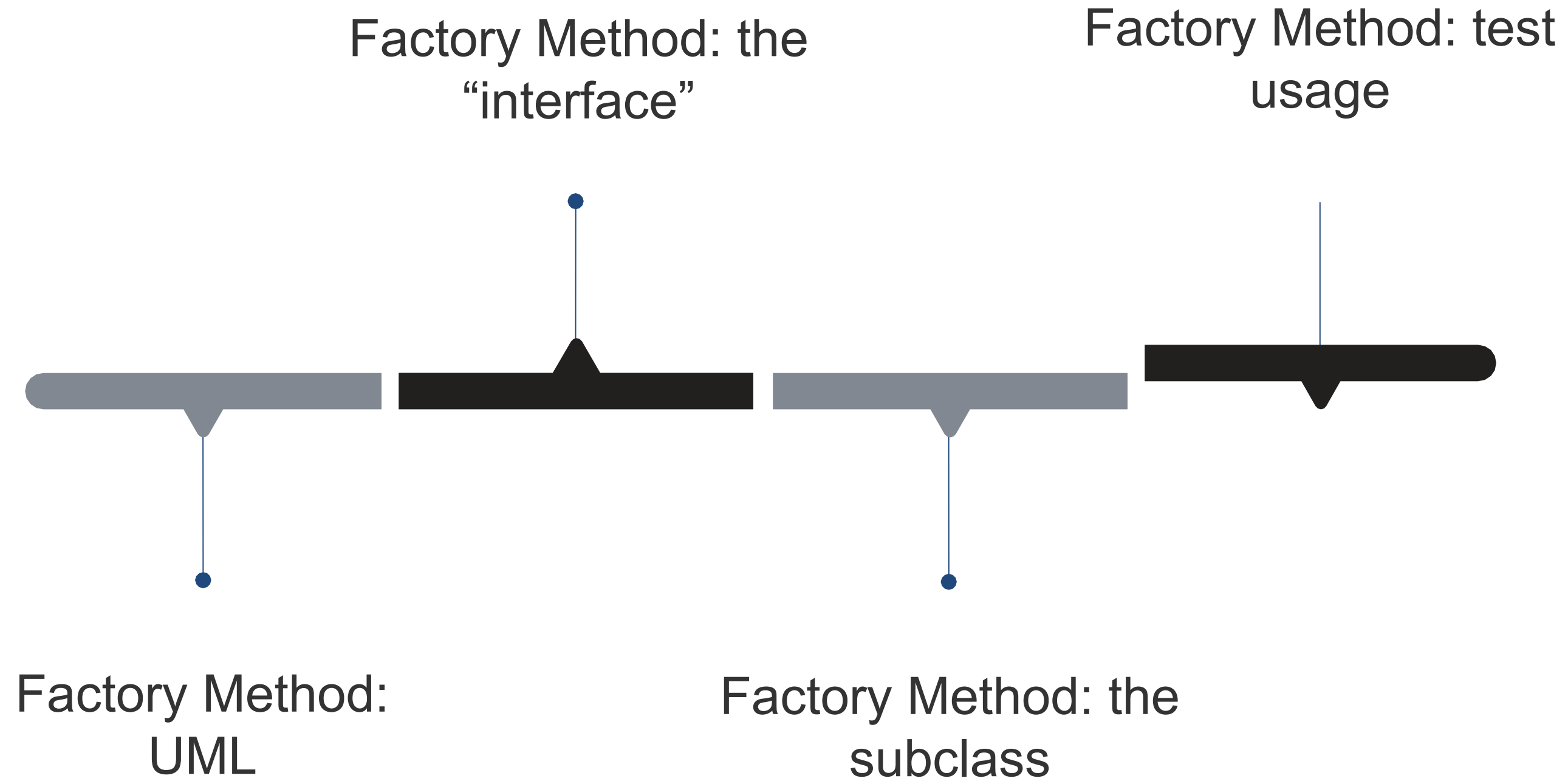


Define an interface for creating objects , but lets subclasses decide which class to instantiate. Factory method lets a class defer instantiation to subclasses.

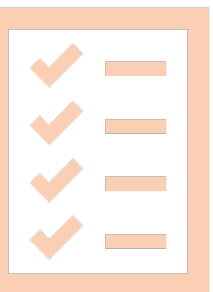
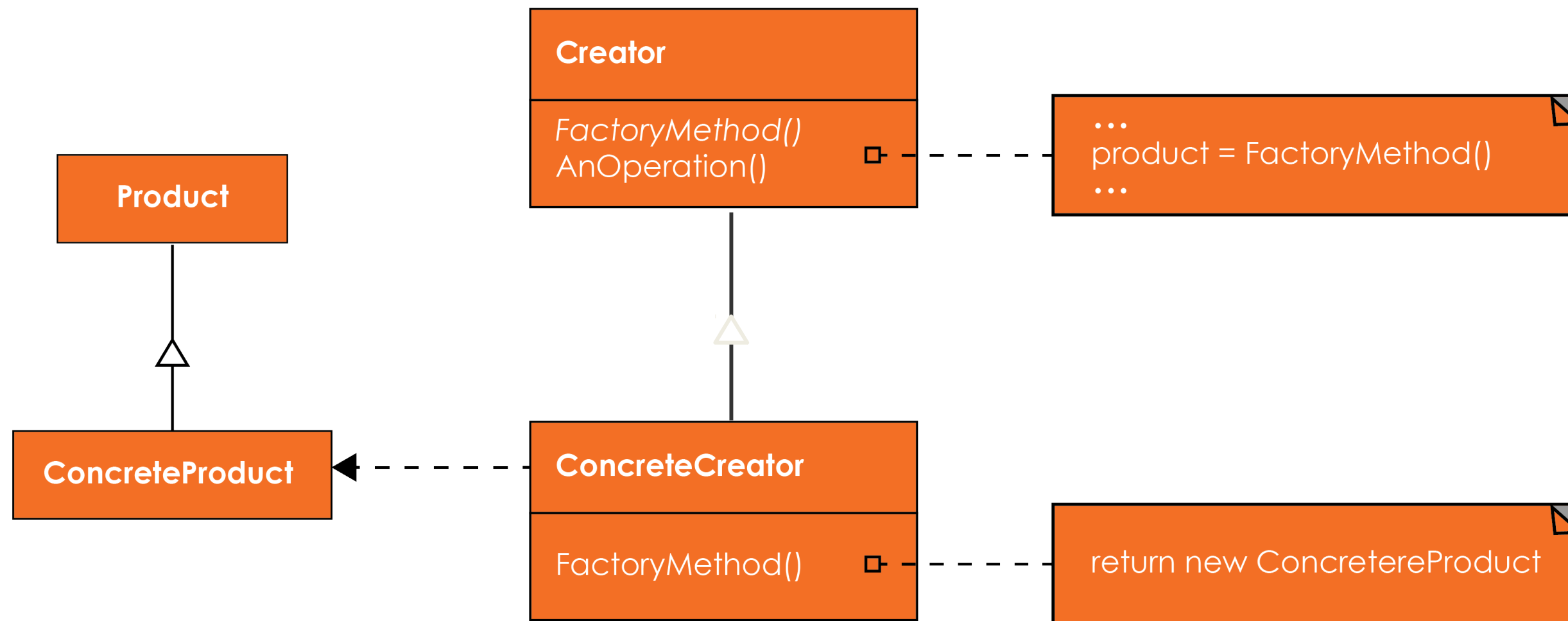
Uh? English please...



Factory Method



Factory Method: UML

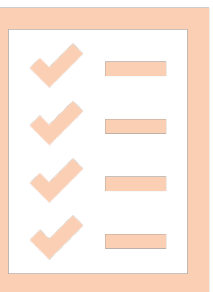


Factory Method: the “interface”

```
public abstract class FoodStore
{
    public void ReceiveOrder(int howMany) { /*logic to receive order */ }

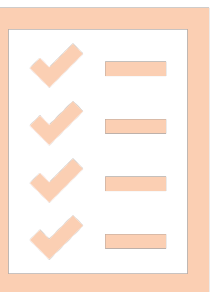
    public void ShipOrder() { /*logic to ship order */ }

    //note how this is abstract
    public abstract Arancino CookArancino();
}
```



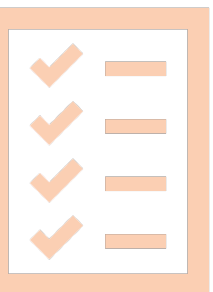
Factory Method: The Subclass

```
public class PalermoFoodStore : FoodStore
{
    public override Arancino CookArancino() => new ArancinoPalermitano();
}
```



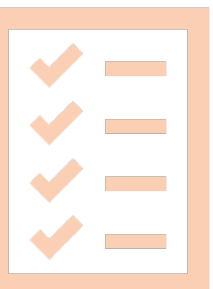
Factory Method : Test Usage

```
[Fact]
public void CookArancinoPalermitano()
{
    FoodStore store = new PalermoFoodStore();
    store.ReceiveOrder(1);
    Arancino cino = store.CookArancino();
    store.ShipOrder();
    Assert.IsType<ArancinoPalermitano>(cino);
}
```



Factory Method : Test Usage

```
[Fact]
public void CookArancinoCatanesese()
{
    FoodStore store = new CataniaFoodStore();
    store.ReceiveOrder(1);
    Arancino cino = store.CookArancino();
    store.ShipOrder();
    Assert.IsType<ArancinoCatanesese>(cino);
}
```





Abstract Factory: Definition

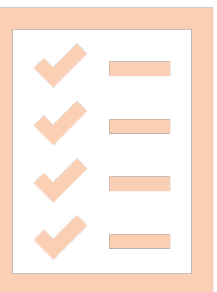
Abstract Factory: Definition



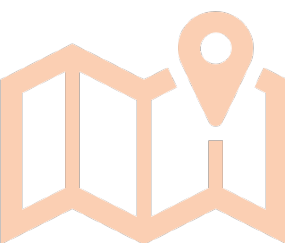
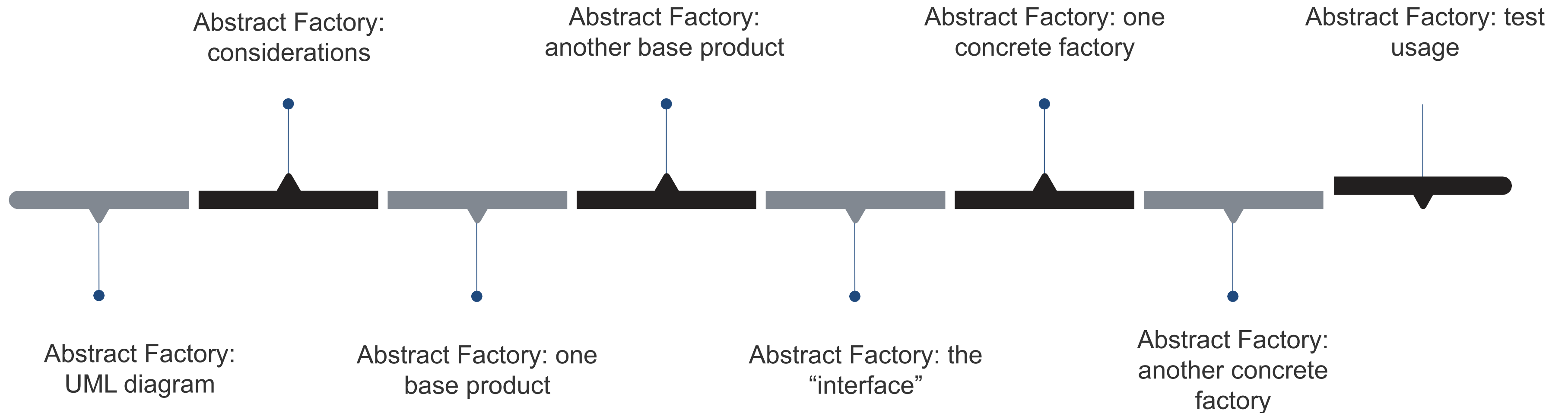
Provides an interface for creating families of related or dependent objects

without specifying their concrete class

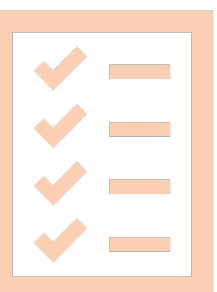
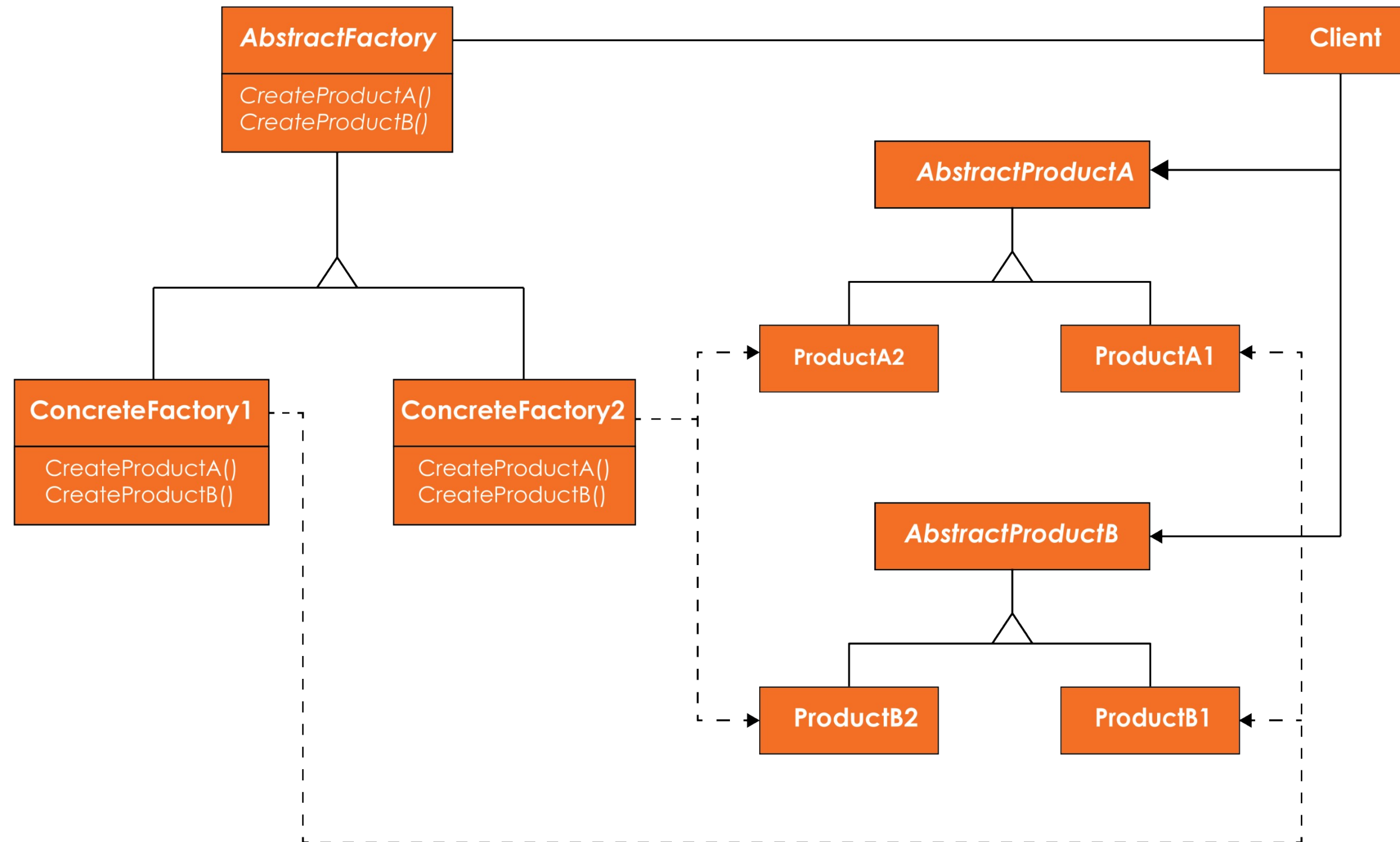
Uh? English please...



Abstract Factory Method

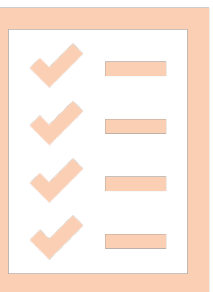


Abstract Factory: UML Diagram



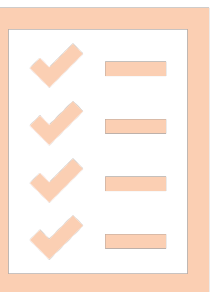
Abstract Factory: Considerations

- ✔ Sort of a combination of the Simple Factory Idiom and the Factory Method:
encapsulate what varies + use polymorphism to change behaviour.
- ✔ Complex solution worth using when multiple families of products are present.



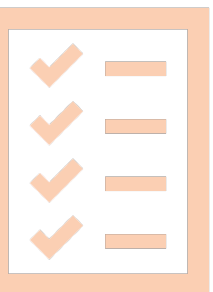
Abstract Factory: One Base Product

```
public abstract class Pizza
{
    private List<string> toppings;
    public Pizza(params string[] toppings)
    {
        this.toppings = new List<string>();
        this.toppings.AddRange(toppings);
    }
    public void AddTopping(string topping)
    {
        toppings.Add(topping);
    }
    public IEnumerable<string> Toppings
    {
        get
        {
```



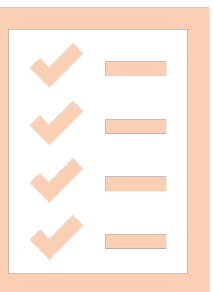
Abstract Factory: Another Base Product

```
public abstract class Pasta
{
    public Pasta(string condiment, bool isFresh)
    {
        Condiment = condiment;
        Fresh = isFresh;
    }
    public string Condiment { get; }
    public bool Fresh { get; }
    public abstract void Cook();
}
```



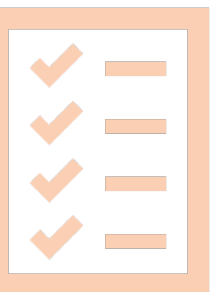
Abstract Factory: the “interface”

```
public abstract class Restaurant
{
    public abstract Pizza CookPizza();
    public abstract Pasta CookPasta();
    public static Restaurant Instance
    {
        get
        {
            foreach(Type t in typeof(Restaurant).Assembly.GetTypes())
            {
                if( !t.IsInterface && typeof(Restaurant).IsAssignableFrom(t))
                {
                    return (Restaurant) Activator.CreateInstance(t);
                }
            }
            return null;
        }
    }
}
```



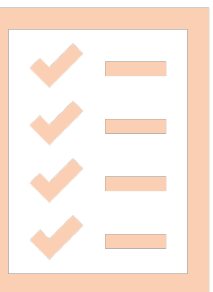
Abstract Factory: One Concrete Factory

```
public class USRestaurant : Restaurant
{
    public override Pasta CookPasta() => new FettuccineAlfredo();
    public override Pizza CookPizza() => new PepperoniPizza();
}
```



Abstract Factory: Another Concrete Factory

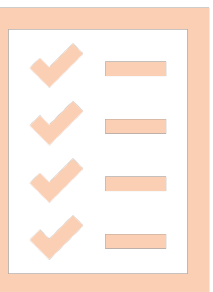
```
public class ItalianRestaurant : Restaurant
{
    public override Pasta CookPasta() => new Ravioli();
    public override Pizza CookPizza() => new Capricciosa();
}
```



Abstract Factory: Test Usage

```
[Fact]
public void CookPizza()
{
    Restaurant resto = Restaurant.Instance;
    Pizza pizza = resto.CookPizza();
    Assert.IsType<Capricciosa>(pizza);
}

[Fact]
public void CookPasta()
{
    Restaurant resto = Restaurant.Instance;
    Pasta pasta = resto.CookPasta();
    Assert.IsType<Ravioli>(pasta);
}
```



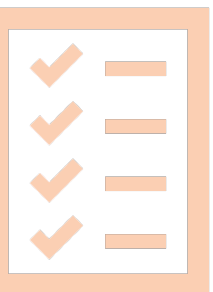
Builder: Definition

Builder: Definition

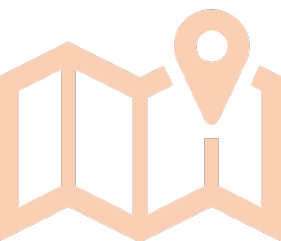
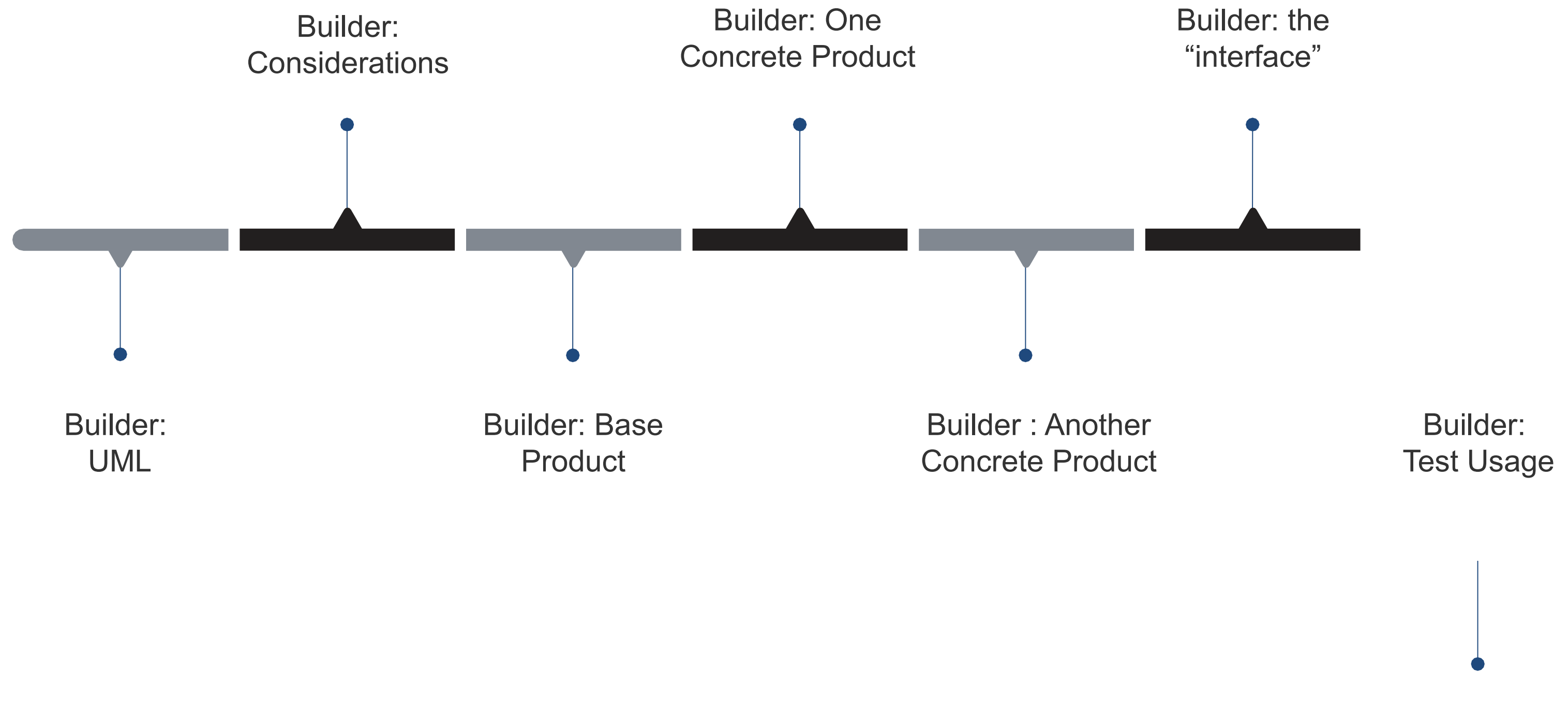


Separate the construction of a complex object from its representation so that the same construction process can create different representations.

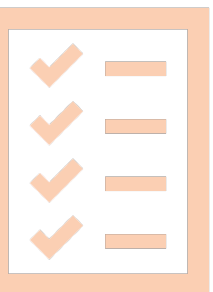
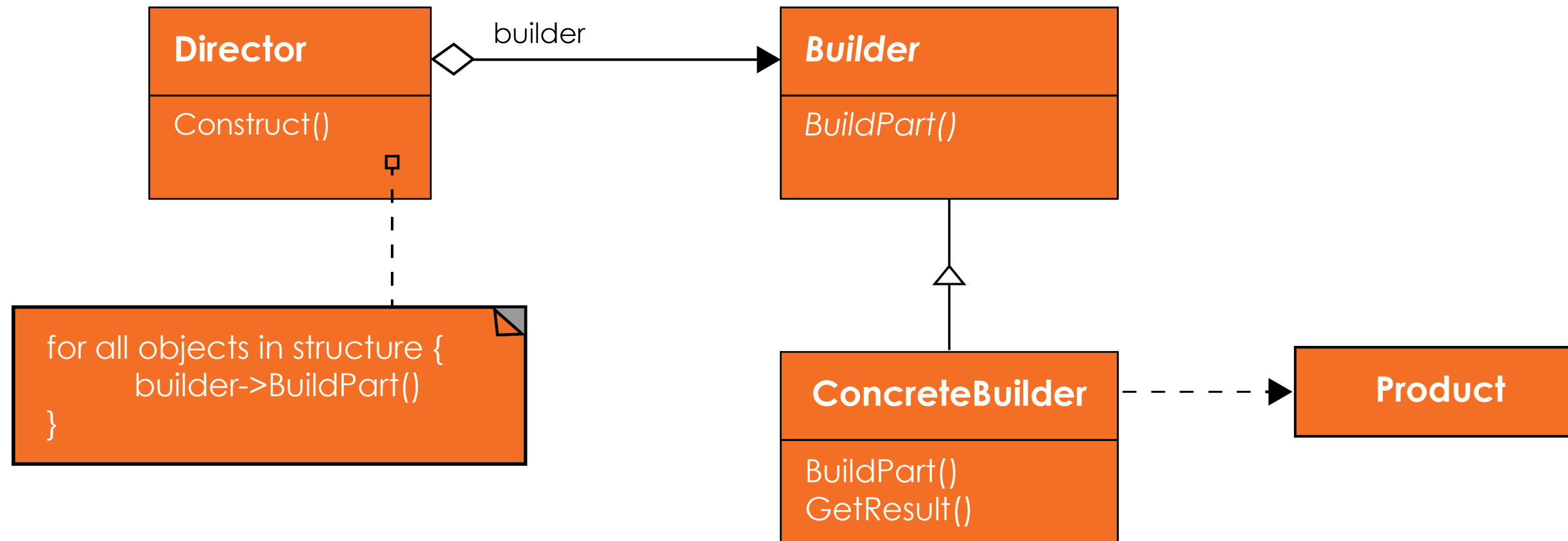
Uh? English, please....



Builder Method

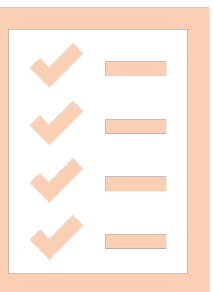


Builder: UML



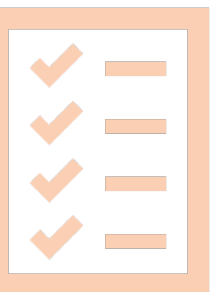
Builder: Considerations

- ✓ The Builder pattern is more concerned about abstracting away the complexities of flexible object construction rather than producing a family of objects in a flexible extensible manner, but it can help also with that.



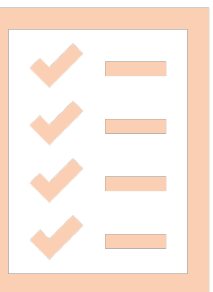
Builder: Base

```
public interface Machine
{
    HardwareComponents Components { get; set; }
    void Play();
    void DevelopCode();
}
```



Builder: One Concrete Product

```
public class Desktop : Machine
{
    public HardwareComponents Components { get; set; }
    = new HardwareComponents();
    public void Play()
    {
        Console.WriteLine("The super expensive graphics " +
            " card now makes a little more sense..");
    }
    public void DevelopCode()
    {
        Console.WriteLine("NOW we are doing something really interesting!");
    }
}
```

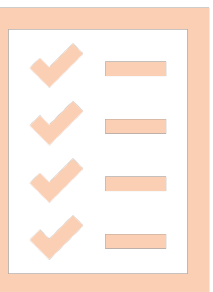


Builder : Another Concrete Product

```
public class GamingConsole : Machine
{
    public HardwareComponents Components { get; set; } = new HardwareComponents();

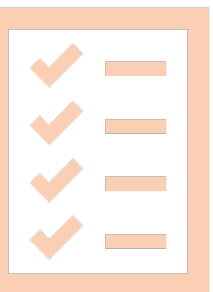
    public void DevelopCode()
    {
        Console.WriteLine("Write code on a console? You cannot be serious!");
    }

    public void Play()
    {
        Console.WriteLine("now let's play, you cannot write code all day!");
    }
}
```



Builder: the “interface”

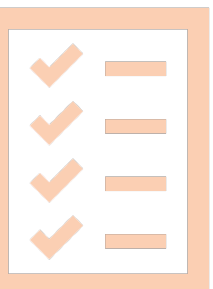
```
public interface HardwareBuilder
{
    HardwareBuilder AddRam(int giga);
    HardwareBuilder AddSSD(int giga);
    HardwareBuilder AddHD(int giga);
}
```



Builder: Test

Usage





```
[Fact]
public void BuildDesktop()
{
    DesktopBuilder builder = new DesktopBuilder();
    HardwareDirector director = new HardwareDirector(builder);
    director.Assemble();
    Machine desk = builder.Build();
    Assert.Equal(300, desk.Components.HD);
    Assert.Equal(16, desk.Components.RAM);
}
```

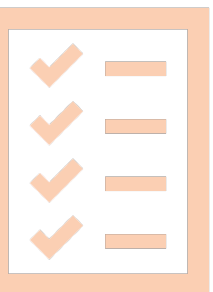


Limitations Of Design Patterns Solutions

Limitations Of Design Patterns

Solutions

-  Introducing complexity (unless using the simple, limited, idiom)
-  Ending up with multiple factories, hard to keep track.
-  Abstract Factory abstraction is rigid: what if you have to add other families of products?
-  What if the products have dependencies on other objects that we want to be flexible? Products using other factories?

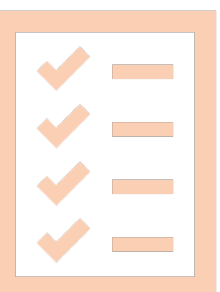




Object Instantiation on Steroids, Dependency Injection

Object Instantiation on Steroids, Dependency Injection

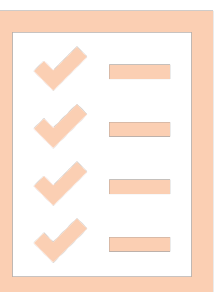
- ✓ These limitations typically come out when the dependencies we need to instantiate are services.
- ✓ To overcome these limitations frameworks have been created with the goal of doing Dependency Injection

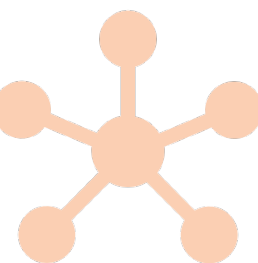
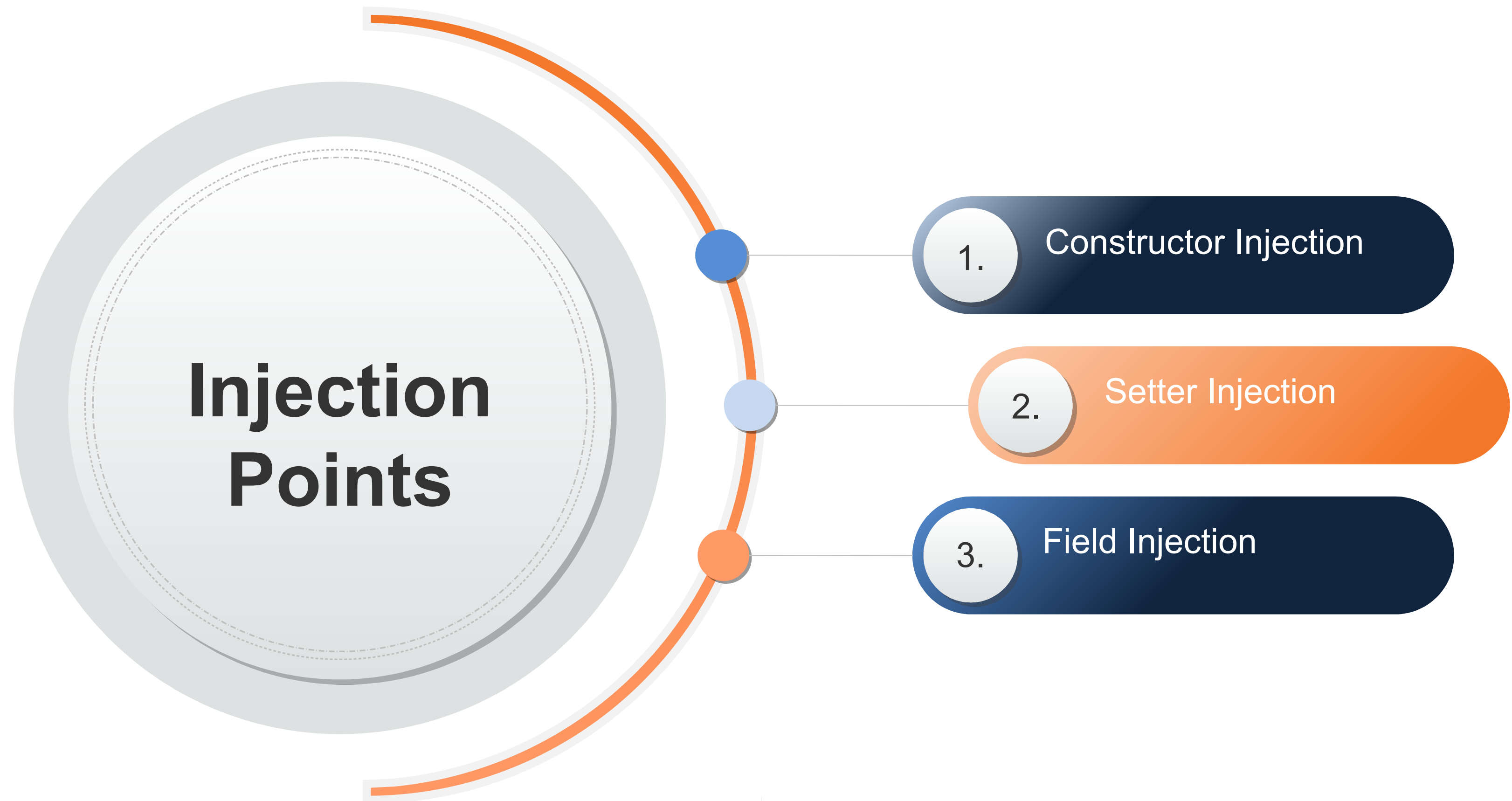


Dependency Injection

Dependency Injection

- ✓ DI is a particular case of a technique known as “Inversion of Control” , or “The Hollywood Principle”
- ✓ The control we are giving up is the decision about which particular implementation to instantiate.
- ✓ We delegate this responsibility to the DI framework.
- ✓ DI frameworks gained traction and popularity in the Java world , with the highly successful Spring Framework.





The Wrong Way..

```
public class CourseEditionController : Controller
{
    private CourseEditionUnitOfWork editionWork;
    public CourseEditionController()
    {
        EditionWork = new CourseEditionUnitOfWork();
    }
}
```



Better Way..

```
public class CourseEditionController : Controller
{
    private CourseEditionUnitOfWork editionWork;
    public CourseEditionController(CourseEditionUnitOfWork uWork)
    {
        EditionWork = uWork;
    }
}
```



Or..

```
public class CourseEditionController : Controller
{
    private CourseEditionUnitOfWork editionWork;
    public CourseEditionController() {}

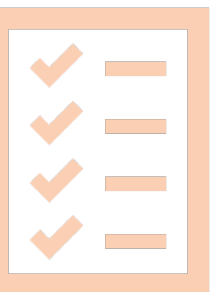
    public CourseEditionUnitOfWork
    {
        set { editionWork = value; }
    }
}
```



Constructor Injection: Advantages

Constructor Injection: Advantages

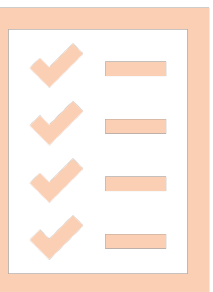
- ✓ Enforces a clear initialization order.
- ✓ Guarantees that objects are created in a complete state



Setter Injection: Advantages

Setter Injection: Advantages

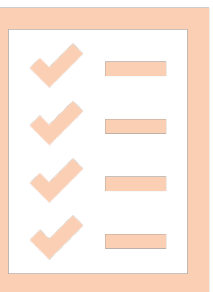
- ✓ More Flexible: not every dependency might be ready at object construction time
- ✓ More Flexible: objects dependencies can be reconfigured after creation time.



Field Injection: Advantages

Field Injection: Advantages

- ✓ “Quick and easy solution” , some might say “quick and dirty”
- ✓ Setter methods are just ugly, especially if you are not the one calling them

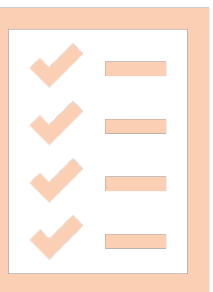


 **Enough Theory! Show Me An Example!**

Enough Theory! Show Me An Example!



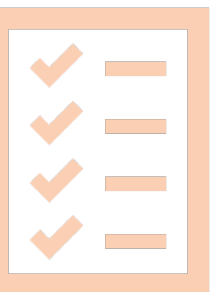
All right, let's see an example of DI in a modern .NET framework,
MVC Core ...



DI in MVC Core

DI in MVC Core

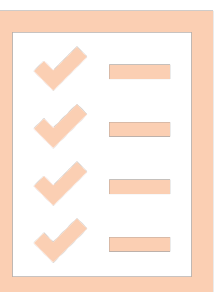
- ✓ In a clean architecture , we want to decouple our web framework from the EF DbContext.
- ✓ Typical solution to isolate the persistence framework is using Repository classes (old school DAO..)
- ✓ But now our controllers will depend on a Repository
- ✓ And our repository will depend on the DbContext.



Repository Pattern

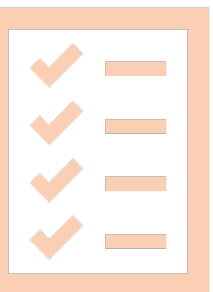
Repository Pattern

- ✓ Mediates between the domain and data mapping layers using a collection-like interface for accessing domain objects (Martin Fowler)
- ✓ In “layman terms” it’s just an object whose responsibility is hiding the persistence framework and database details and give the illusion that we are dealing with a collection of objects in memory.
- ✓ Sounds cool, one of the terms introduced by Domain Driven Design, but really not that different from the “Java Old School” DAOs



Repository Pattern

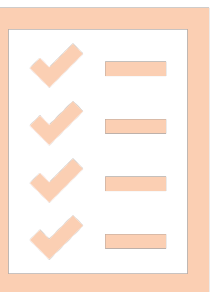
- ✓ Allows reuse of query logic
- ✓ Provides better separation of concerns
- ✓ Decouples business logic from query logic and persistence framework.
- ✓ Same logic as a collection of in memory objects
- ✓ So how about save or update methods?



Unit of Work Pattern

Unit of Work Pattern

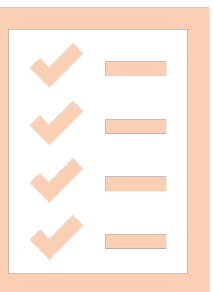
- ✓ Maintains a list of objects affected by a business transaction and coordinates the writing out of changes
- ✓ Again, sounds new and cool, but not that different from the “Java Old School” Business Delegate
- ✓ Coordinating multiple repositories and saving changes is the responsibility of the Unit of Work

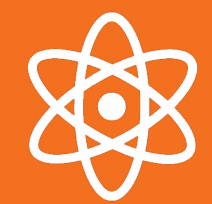


Unit of Work in EF

Unit of Work in EF

- ✓ The Unit of Work pattern is actually implemented by EF itself
- ✓ The `DBSet<T>` are the repositories
- ✓ The `DbContext` is the unit of work

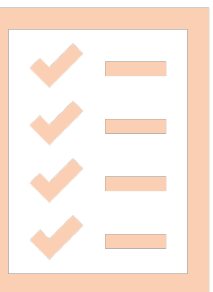




DI in Unit of Work

DI in Unit of Work

- ✓ All right, but where is my Dependency Injection?
- ✓ First, we need to program by interfaces:
- ✓ `ICourseRepository`
- ✓ `IcourseUnitOfWork`
- ✓ Our `AppDbContext` already extends `DbContext`



 **Caution**

The Wrong Way..

```
public class CourseEditionController : Controller
{
    private CourseEditionUnitOfWork editionWork;
    public CourseEditionController()
    {
        EditionWork = new CourseEditionUnitOfWork();
    }
}
```



The Wrong Way..

```
public class CourseEditionUnitOfWork
{
    private CourseRepository courseRepository;
    private CourseEditionRepository courseEditionRepository;
    public CourseEditionUnitOfWork()
    {
        courseRepository = new CourseRepository();
        CourseEditionRepository = new CourseEditionREpository();
    }
}
```



The Wrong Way..

```
public class CourseRepository
{
    private CodeGymContext ctx;
    public CourseRepository()
    {
        ctx = new CodeGymContext(....)
    }
}
```



Another Wrong Way..

```
public class CourseRepository
{
    ...
    public void Create(Course course)
    {
        ctx.Courses.Add(course);
ctx.SaveChanges();
    }
}
```



A Better Way..

```
public interface CourseEditionUnitOfWork
{
    CourseEditionRepository CourseEditions { get;}
    CourseRepository Courses{ get;}
    void Begin();
    void End();
    void Save();
    void Cancel(Course c);
}
```



A Better Way..

```
public interface CourseEditionRepository
{
    CourseEdition FindById(long id);
    void Remove(long editionId);
    void Add(CourseEdition edition);
    IEnumerable<CourseEdition> FindByCourseId(long id);
    IEnumerable<CourseEdition> FindAll();
}
```



A Better Way..

```
public class EFCourseEditionRepository : CourseEditionRepository
{
    private CodeGymContext ctx;
    public EFCourseEditionRepository(CodeGymContext ctx)
    {
        this.ctx = ctx;
    }

    public void Add(CourseEdition edition)
    {
        ctx.CourseEditions.Add(edition);
    }

    public CourseEdition FindById(long id)
    {
        return ctx.CourseEditions.Find(id);
    }
}
```



A Better Way..

```
public class EFCourseEditionUnitOfWork
{
    ...
    public EFCourseEditionUnitOfWork(CourseRepository courses,
        CourseEditionRepository courseEditions, CodeGymContext ctx)
    {
        this.Courses = courses;
        this.CourseEditions = courseEditions;
        this.ctx = ctx;
    }

    public void Save()
    {
        ctx.SaveChanges();
    }
    ....
}
```



Objection...



Thats a lot of constructor dependencies, who's gonna take care of them?

The Dependency Injection Features available in MVC Core will!



MVC Dependency Injection

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<CodeGymContext>(opts => opts.UseSqlServer(
        Configuration.GetConnectionString("DefaultConnection")));
    services.AddTransient<CourseRepository, EFCourseRepository>();
    services.AddTransient<CourseEditionRepository, EFCourseEditionRepository>();
    services.AddTransient<EnrollmentRepository, EFEnrollmentRepository>();
    services.AddTransient<CourseEditionUnitOfWork,
EFCourseEditionUnitOfWork>();
    services.AddMvc();
}
```





Demo : DI in MVC Core

Unit Testing

Dependency Injection helps with mocking dependencies in Unit Testing as well:

```
public CourseEditionControllerTests(
{
    mockCourseRepository = new Mock<CourseRepository>();
    mockCourseEditionRepository = new Mock<CourseEditionRepository>();
    mockCourseEditionUnitOfWork = new Mock<CourseEditionUnitOfWork>();
    mockCourseEditionUnitOfWork.SetupGet(u =>
u.Courses).Returns(mockCourseRepository.Object);
    mockCourseEditionUnitOfWork.SetupGet(u =>

u.CourseEditions).Returns(mockCourseEditionRepository.Object);
    controller = new
CourseEditionController(mockCourseEditionUnitOfWork.Object);
}
```



Demo : Unit Testing in MVC Core



Student Exercise: Complete A Unit Test Mocking Dependencies

Summary

Things We Learned



Direct use of constructors can be messy and kills extensibility and unit testing



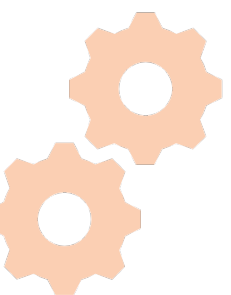
Factory Patterns can be an effective way to abstract away object instantiation



What Dependency Inversion is and why it needs Dependency Injection



How DI helps creating a clean architecture and facilitates Unit Testing, with examples in MVC Core.



THANK YOU!

