

объектно-ориентированное программирование

порождающие паттерны

фабричный метод

**вариативность создания объектов при
помощи наследования и полиморфизма**

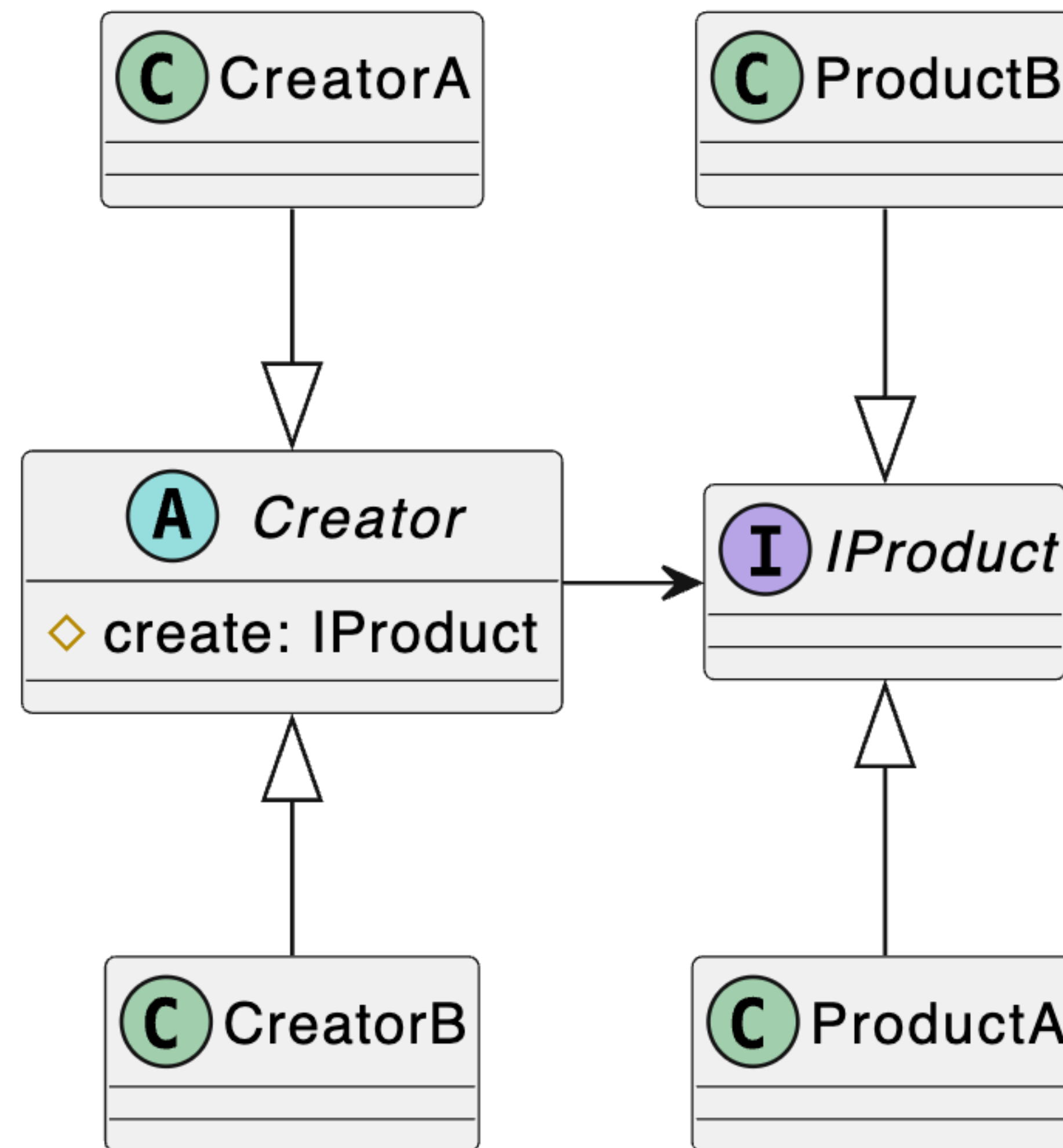


фабричный метод

схема

фабричный метод

- creator
тип, в котором содержится логика, в рамках которой создаются объекты
наследники реализуют логику создания объектов
- product
тип, создаваемых объектов
наследники создаются в конкретных creator'ах



пример использования фабричный метод

```
public record OrderItem(  
    decimal Price,  
    int Amount)  
{  
    public decimal Cost ⇒ Price * Amount;  
}  
  
public record Order(  
    IEnumerable<OrderItem> Items)  
{  
    public decimal TotalCost ⇒ Items.Sum(x ⇒ x.Cost);  
}
```

```
public record CashPayment(decimal Amount);  
  
public class PaymentCalculator  
{  
    public CashPayment Calculate(Order order)  
    {  
        var totalCost = order.TotalCost;  
  
        // Apply discounts and coupons  
        ...  
  
        return new CashPayment(totalCost);  
    }  
}
```

пример использования фабричный метод

```
public interface IPayment
{
    decimal Amount { get; }
}
```

```
public record CashPayment(
    decimal Amount) : IPayment;

public record BankPayment(
    decimal Amount,
    string ReceiverAccountId) : IPayment;
```

пример использования фабричный метод

```
public abstract class PaymentCalculator
{
    public IPayment Calculate(Order order)
    {
        var totalCost = order.TotalCost;

        // Apply discounts and coupons
        ...

        return CreatePayment(totalCost);
    }

    protected abstract IPayment CreatePayment(decimal amount);
}
```

```
public class CashPaymentCalculator : PaymentCalculator
{
    protected override IPayment CreatePayment(decimal amount)
        => new CashPayment(amount);
}

public class BankPaymentCalculator : PaymentCalculator
{
    private readonly string _currentReceiverAccountId;

    public BankPaymentCalculator(string currentReceiverAccountId)
    {
        _currentReceiverAccountId = currentReceiverAccountId;
    }

    protected override IPayment CreatePayment(decimal amount)
    {
        return new BankPayment(amount, _currentReceiverAccountId);
    }
}
```

пример использования фабричный метод

```
public abstract class PaymentCalculator
{
    public IPayment Calculate(Order order)
    {
        var totalCost = order.TotalCost;

        // Apply discounts and coupons
        ...

        return CreatePayment(totalCost);
    }

    protected abstract IPayment CreatePayment(decimal amount);
}
```

```
public class CashPaymentCalculator : PaymentCalculator
{
    protected override IPayment CreatePayment(decimal amount)
        => new CashPayment(amount);
}

public class BankPaymentCalculator : PaymentCalculator
{
    private readonly string _currentReceiverAccountId;

    public BankPaymentCalculator(string currentReceiverAccountId)
    {
        _currentReceiverAccountId = currentReceiverAccountId;
    }

    protected override IPayment CreatePayment(decimal amount)
    {
        return new BankPayment(amount, _currentReceiverAccountId);
    }
}
```


пример использования фабричный метод

```
public abstract class PaymentCalculator
{
    public IPayment Calculate(Order order)
    {
        var totalCost = order.TotalCost;

        // Apply discounts and coupons
        ...

        return CreatePayment(totalCost);
    }

    protected abstract IPayment CreatePayment(decimal amount);
}
```

```
public class CashPaymentCalculator : PaymentCalculator
{
    protected override IPayment CreatePayment(decimal amount)
        => new CashPayment(amount);
}

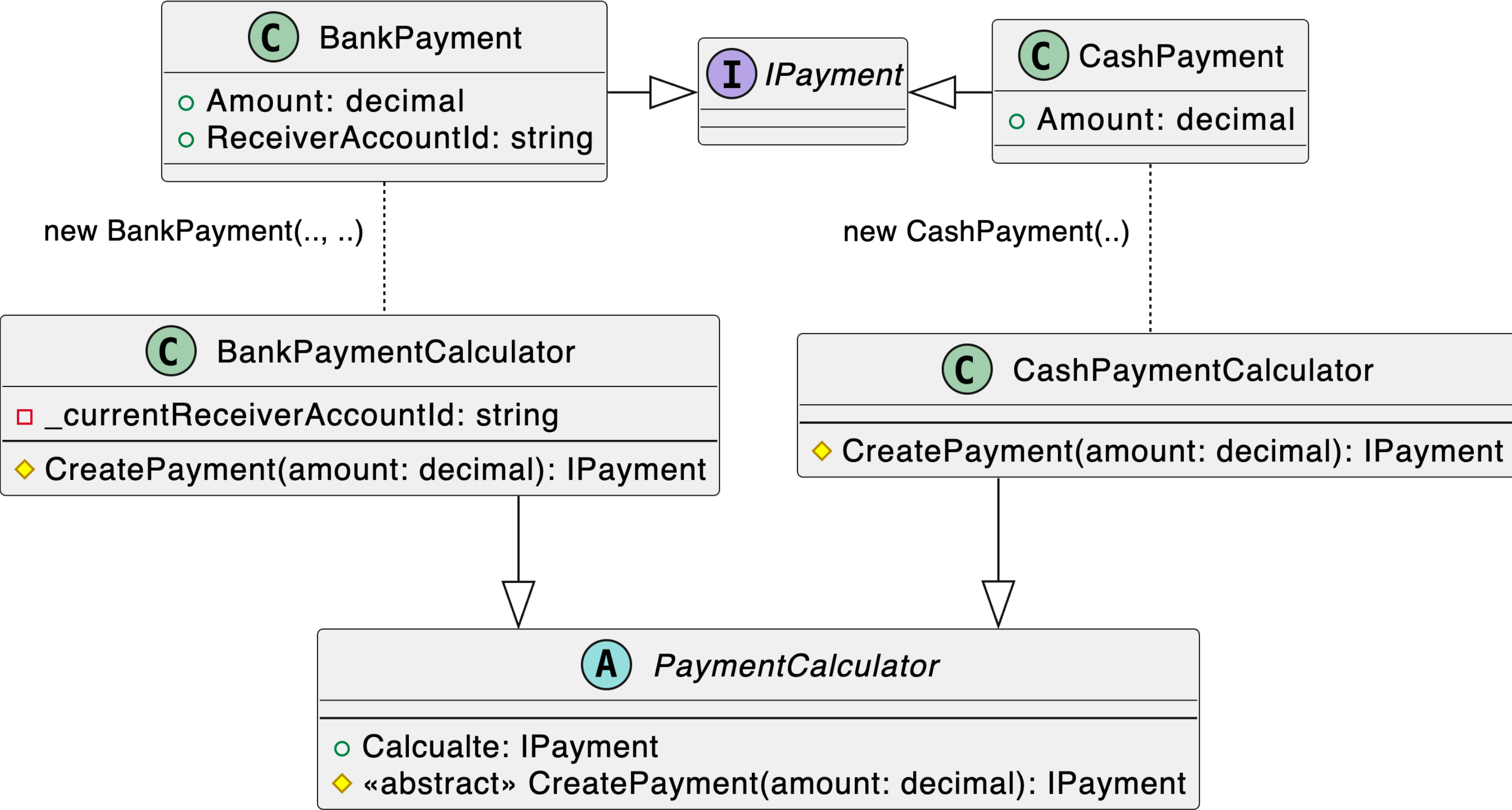
public class BankPaymentCalculator : PaymentCalculator
{
    private readonly string _currentReceiverAccountId;

    public BankPaymentCalculator(string currentReceiverAccountId)
    {
        _currentReceiverAccountId = currentReceiverAccountId;
    }

    protected override IPayment CreatePayment(decimal amount)
    {
        return new BankPayment(amount, _currentReceiverAccountId);
    }
}
```

схема использования

фабричный метод



недостатки

фабричный метод

- сильная связанность конкретных создателей с базовым типом

переиспользуется логика базового типа, но не логика конкретных создателей

недостатки фабричный метод

```
public abstract class PaymentCalculator
{
    public IPayment Calculate(Order order) { ... }

    protected abstract IPayment CreatePayment(decimal amount);
}
```

```
public class CashPaymentCalculator
    : PaymentCalculator { ... }
```

```
public class BankPaymentCalculator
    : PaymentCalculator { ... }
```

```
public abstract class FixedPricePaymentCalculator
{
    public IPayment Calculate(Order order) { }

    protected abstract IPayment CreatePayment(decimal amount);
}
```

```
public class FixedCashPaymentCalculator
    : FixedPricePaymentCalculator { ... }
```

```
public class FixedBankPaymentCalculator
    : FixedPricePaymentCalculator { ... }
```

недостатки

фабричный метод

- сильная связанность конкретных создателей с базовым типом
переиспользуется логика базового типа, но не логика конкретных создателей
- неявное нарушение SRP
объект конкретного создателя ответственен как за реализацию логики, так и за создание продуктов

абстрактная фабрика

**вариативность создания объектов при
помощи композиции и полиморфизма**



(абстрактная) фабрика

пример использования абстрактная фабрика

```
public interface IPayment
{
    decimal Amount { get; }
}
```

```
public record CashPayment(
    decimal Amount) : IPayment;

public record BankPayment(
    decimal Amount,
    string ReceiverAccountId) : IPayment;
```


пример использования абстрактная фабрика

```
public interface IPaymentFactory
{
    IPayment Create(decimal amount);
}
```

```
public class CashPaymentFactory : IPaymentFactory
{
    public IPayment Create(decimal amount)
        ⇒ new CashPayment(amount);
}

public class BankPaymentFactory : IPaymentFactory
{
    private readonly string _currentReceiverAccountId;

    public BankPaymentFactory(string currentReceiverAccountId)
    {
        _currentReceiverAccountId = currentReceiverAccountId;
    }

    public IPayment Create(decimal amount)
    {
        return new BankPayment(amount, _currentReceiverAccountId);
    }
}
```

пример использования абстрактная фабрика

```
public interface IPaymentCalculator
{
    IPayment Calculate(Order order);
}
```

```
public class PaymentCalculator : IPaymentCalculator
{
    private readonly IPaymentFactory _paymentFactory;

    public PaymentCalculator(IPaymentFactory paymentFactory)
    {
        _paymentFactory = paymentFactory;
    }

    public IPayment Calculate(Order order)
    {
        var totalCost = order.TotalCost;

        // Apply discounts and coupons
        ...

        return _paymentFactory.Create(totalCost);
    }
}
```

пример использования абстрактная фабрика

```
public class FixedPaymentCalculator : IPaymentCalculator
{
    private readonly decimal _fixedPrice;
    private readonly IPaymentFactory _paymentFactory;

    public FixedPaymentCalculator(decimal fixedPrice, IPaymentFactory paymentFactory)
    {
        _fixedPrice = fixedPrice;
        _paymentFactory = paymentFactory;
    }

    public IPayment Calculate(Order order)
    {
        var totalCost = order.Items.Sum(item => _fixedPrice * item.Amount);

        // Apply discounts and coupons
        ...

        return _paymentFactory.Create(totalCost);
    }
}
```

абстрактная фабрика

преимущества

- настоящее соблюдение SRP
ведь в такой реализации нет прямой связанности между реализациями

абстрактная фабрика

преимущества

- настоящее соблюдение SRP
ведь в такой реализации нет прямой связанности между реализациями
- соблюдение OCP
мы можем добавить в систему новые виды платежей и реализовать фабрики для них, тем самым, расширить логику не меняя реализацию калькуляторов

билдер

**выделение отдельного типа,
инкапсулирующего логику сбора данных
и создания объекта**



билдер

параметры и аргументы

параметр

набор тип+имя находящийся в сигнатуре метод

```
public void A(int a, char b);
```

аргумент

значение передающееся в метод

```
obj.A(1, '2');
```


пример реализации билдер

```
public record Order(IEnumerable<OrderItem> Items);

public class OrderBuilder
{
    private readonly List<OrderItem> _items = [];

    public OrderBuilder WithItem(OrderItem item)
    {
        _items.Add(item);
        return this;
    }

    public Order Build() => new Order(_items.ToArray());
}
```

пример использования билдер

```
var orderBuilder = new OrderBuilder()  
    .WithItem(new OrderItem(Price: 1337, Amount: 2));
```

```
AddDefaultItems(orderBuilder);  
AddRequestedItems(orderBuilder);  
AddForecastedItems(orderBuilder);
```

```
Order order = orderBuilder.Build();
```

convenience builder

билдер

- модель никак не связана с билдером

convenience builder

билдер

- модель никак не связана с билдером
- несёт в себе вспомогательный функционал

convenience builder

билдер

- модель никак не связана с билдером
- несёт в себе вспомогательный функционал
- используется для упрощения создания объектов

convenience builder

билдер

```
public record Order(  
    string CommentForShop,  
    string CommentForDelivery,  
    IEnumerable<OrderItem> Items,  
    DateTimeOffset CreatedAt,  
    string? ReceiverPhoneNumber);
```

```
var order = new Order(  
    CommentForShop: string.Empty,  
    CommentForDelivery: string.Empty,  
    Items: [new OrderItem(Price: 1337, Amount: 2)],  
    CreatedAt: DateTimeOffset.UtcNow,  
    ReceiverPhoneNumber: null);
```

convenience builder

билдер

```
public class OrderBuilder
{
    private readonly List<OrderItem> _items = [];
    private string _commentForShop = string.Empty;
    private string _commentForDelivery = string.Empty;
    private DateTimeOffset _createdAt = DateTimeOffset.UtcNow;
    private string? _receiverPhoneNumber = null;

    public OrderBuilder WithItem(OrderItem item)
    {
        _items.Add(item);
        return this;
    }

    public OrderBuilder WithCommentForShop(string value) { ... }

    public Order Build() ⇒ new Order(...);
}
```

convenience builder

билдер

```
var order = new OrderBuilder()  
    .WithItem(new OrderItem(Price: 1337, Amount: 2))  
    .Build();
```


stateful constructor

билдер

- в билдер выносятся валидации входных данных

stateful constructor

билдер

- в билдер выносятся валидации входных данных
- позволяет выполнять валидации во время сбора данных

stateful constructor

билдер

- в билдер выносятся валидации входных данных
- позволяет выполнять валидации во время сбора данных
 - fail-fast

stateful constructor

билдер

- в билдер выносятся валидации входных данных
- позволяет выполнять валидации во время сбора данных
 - fail-fast
 - упрощение логики валидации

stateful constructor

билдер

- в билдер выносятся валидации входных данных
- позволяет выполнять валидации во время сбора данных
 - fail-fast
 - упрощение логики валидации
 - упрощение определения момента добавления некорректных данных

stateful constructor

билдер

MAX = 20

`public record Order(IEnumerable<OrderItem> Items);`



stateful constructor

билдер

```
public class Order
{
    private Order(IEnumerable<OrderItem> items)
    {
        Items = items;
    }

    public IEnumerable<OrderItem> Items { get; }

    public class OrderBuilder { ... }
}
```

stateful constructor

билдер

```
public class OrderBuilder
{
    private const int MaxOrderItemCount = 20;

    private readonly List<OrderItem> _items = [];

    public OrderBuilder WithItem(OrderItem item)
    {
        if (_items.Count is MaxOrderItemCount)
            throw new ArgumentException();

        _items.Add(item);
        return this;
    }

    public Order Build() => new Order(_items.ToArray());
}
```


stateful constructor

билдер

```
var orderBuilder = new Order.OrderBuilder();  
  
for (int i = 0; i < 20; i++)  
{  
    orderBuilder.WithItem(new OrderItem(Price: i, Amount: 1));  
}
```

 orderBuilder.WithItem(new OrderItem(Price: 1000, Amount: 1));

stateful constructor

билдер

```
var orderBuilder = new Order.OrderBuilder()  
    .WithItem(new OrderItem(Price: 1337, Amount: 2));
```

```
AddDefaultItems(orderBuilder);
```

```
 AddRequestedItems(orderBuilder);
```

```
AddForecastedItems(orderBuilder);
```

```
Order order = orderBuilder.Build();
```

СМЕШЕНИЕ ТИПОВ

билдер

- смешивать типы builder'ов можно
- НО! необходимость смешения скорее всего свидетельствует о необходимости декомпозиции модели
- стоит помнить что реализация builder'а должна зависеть от модели, а не наоборот

полиморфизм

билдер

```
public interface IOrderBuilder
{
    IOrderBuilder WithItem(OrderItem item);

    Order Build();
}
```

```
public class UnlimitedOrderBuilder : IOrderBuilder { ... }
```

```
public class LimitedOrderBuilder : IOrderBuilder { ... }
```

директор билдер

```
public record Pizza(  
    PizzaSize Size,  
    DoughType DoughType,  
    Sause Sause,  
    IReadOnlyCollection<Topping> Toppings);
```

```
public class PizzaBuilder  
{  
    private readonly List<Topping> _toppings = [];  
    private PizzaSize _size = PizzaSize.Medium;  
    private DoughType _doughType = DoughType.Standard;  
    private Sause _sause = Sause.Tomato;  
  
    public PizzaBuilder WithTopping(Topping topping) { ... }  
  
    public PizzaBuilder WithSize(PizzaSize size) { ... }  
  
    public PizzaBuilder WithDoughType(DoughType type) { ... }  
  
    public PizzaBuilder WithSause(Sause sause) { ... }  
  
    public Pizza Build() { ... }  
}
```

директор билдер

```
public interface IPizzaDirector
{
    PizzaBuilder Direct(PizzaBuilder builder);
}

public class PepperoniPizzaDirector : IPizzaDirector
{
    public PizzaBuilder Direct(PizzaBuilder builder)
    {
        return builder
            .WithDoughType(DoughType.Standard)
            .WithSause(Sause.Tomato)
            .WithSize(PizzaSize.Medium)
            .WithTopping(Topping.Cheese)
            .WithTopping(Topping.Pepperoni);
    }
}
```

```
var pizzaBuilder = new PizzaBuilder();
var pepperoniDirector = new PepperoniPizzaDirector();

var myPizza = pepperoniDirector
    .Direct(pizzaBuilder)
    .WithTopping(Topping.Jalapeno)
    .WithSize(PizzaSize.Large)
    .Build();
```

директор

билдер

```
public static class PizzaBuilderExtensions
{
    public static PizzaBuilder DirectPepperoni(
        this PizzaBuilder builder)
    {
        return builder
            .WithDoughType(DoughType.Standard)
            .WithSause(Sause.Tomato)
            .WithSize(PizzaSize.Medium)
            .WithTopping(Topping.Cheese)
            .WithTopping(Topping.Pepperoni);
    }
}
```


```
var myPizza = new PizzaBuilder()
    .DirectPepperoni()
    .WithTopping(Topping.Jalapeno)
    .WithSize(PizzaSize.Large)
    .Build();
```

interface driven билдер


```
public record Email(  
    string Address,  
    string Subject,  
    string Body);
```

```
public class EmailBuilder  
{  
    private string? _address;  
    private string _subject = string.Empty;  
    private string _body = string.Empty;  
  
    public EmailBuilder WithAddress(string address) { }  
  
    public EmailBuilder WithSubject(string subject) { }  
  
    public EmailBuilder WithBody(string body) { }  
  
    public Email Build()  
    {  
        if (_address is null)  
            throw new ArgumentNullException();  
  
        return new Email(_address, _subject, _body);  
    }  
}
```


interface driven билдер



```
var email = new EmailBuilder()  
    .WithBody("Hello!")  
    .Build();
```



builder

interface driven

```
public interface IEmailAddressBuilder
{
    IEmailBuilder WithAddress(string address);
}

public interface IEmailBuilder
{
    IEmailBuilder WithSubject(string subject);
    IEmailBuilder WithBody(string body);
    Email Build();
}
```

```
public class Email
{
    public required string Address { get; init; }
    public required string Subject { get; init; }
    public required string Body { get; init; }

    public static IEmailAddressBuilder Builder
        => new EmailBuilder();

    private class EmailBuilder
        : IEmailAddressBuilder, IEmailBuilder
    {
        public IEmailBuilder WithAddress(string address) { ... }


        public IEmailBuilder WithSubject(string subject) { ... }

        public IEmailBuilder WithBody(string body) { ... }


        public Email Build() { ... }
    }
}
```

builder

interface driven



```
var email = Email.Builder  
    .WithBody("Hello!")  
    .Build();
```



prototype

prototype

почему не просто конструктор?

- логика копирования может быть необходима в нескольких местах
- данные могут быть сокрыты, модифицированы в конструкторе
- объект находится в иерархии, при копировании конкретный тип не известен

prototype

shallow copy

```
public class Prototype
{
    private readonly IReadOnlyCollection<int> _relatedEntityIds;

    public Prototype(IReadOnlyCollection<int> relatedEntityIds)
    {
        _relatedEntityIds = relatedEntityIds;
    }

    public Prototype Clone()
    {
        return new Prototype(_relatedEntityIds);
    }
}
```

prototype

deep copy

```
public class WrappedValue
{
    public int Value { get; set; }

    public WrappedValue Clone()
        ⇒ new WrappedValue { Value = Value };
}

public class DeepCopyPrototype
{
    private readonly List<WrappedValue> _values;

    public DeepCopyPrototype(List<WrappedValue> values)
    {
        _values = values;
    }

    public DeepCopyPrototype Clone()
    {
        List<WrappedValue> values = _values.Select(x ⇒ x.Clone()).ToList();
        return new DeepCopyPrototype(values);
    }
}
```

prototype

иерархии

```
public interface IHierarchyPrototype
{
    IHierarchyPrototype Clone();
}
```

```
public class FirstDerivedPrototype : IHierarchyPrototype
{
    private readonly string _name;
    private readonly int _age;

    public FirstDerivedPrototype(string name, int age)
    {
        _name = name;
        _age = age;
    }

    public IHierarchyPrototype Clone()
    {
        return new FirstDerivedPrototype(_name, _age);
    }
}
```

```
public class SecondDerivedPrototype : IHierarchyPrototype
{
    private readonly long _iterationCount;

    public SecondDerivedPrototype(long iterationCount)
    {
        _iterationCount = iterationCount;
    }

    public IHierarchyPrototype Clone()
    {
        return new SecondDerivedPrototype(_iterationCount);
    }
}
```


prototype

типизация прототипов-иерархий

```
public abstract class Prototype
{
    public abstract Prototype Clone();
}

public class ClassPrototype : Prototype
{
    public override ClassPrototype Clone()
    {
        return new ClassPrototype();
    }
}
```

prototype

типизация прототипов-иерархий

```
public interface IPrototype
{
    IPrototype Clone();
}

public class InterfacePrototype : IPrototype
{
    IPrototype IPrototype.Clone()
    {
        return Clone();
    }

    public InterfacePrototype Clone()
    {
        return new InterfacePrototype();
    }
}
```

prototype

проблемы переиспользования: наследование

```
public abstract class Prototype
{
    public void DoSomeStuff() { ... }

    public abstract Prototype Clone();
}

public class ClassPrototype : Prototype
{
    public void DoOtherStuff() { ... }

    public override Prototype Clone()
        ⇒ new ClassPrototype();
}
```

```
public class Scenario
{
    public static Prototype CloneAndDoSomeStuff(Prototype prototype)
    {
        var clone = prototype.Clone();
        clone.DoSomeStuff();

        return clone;
    }

    public static void TopLevelScenario()
    {
        var prototype = new ClassPrototype();
        Prototype clone = CloneAndDoSomeStuff(prototype);

        clone.DoOtherStuff();
    }
}
```

prototype

проблемы переиспользования: интерфейсы

```
public interface IPrototype
{
    IPrototype Clone();

    void DoSomeStuff();
}

public class InterfacePrototype : IPrototype
{
    IPrototype IPrototype.Clone()
        ⇒ Clone();

    public InterfacePrototype Clone()
        ⇒ new InterfacePrototype();

    public void DoSomeStuff() { ... }

    public void DoOtherStuff() { ... }
}
```

```
public class Scenario
{
    public static IPrototype CloneAndDoSomeStuff(IPrototype prototype)
    {
        var clone = prototype.Clone();
        clone.DoSomeStuff();

        return clone;
    }

    public static void TopLevelScenario()
    {
        var prototype = new InterfacePrototype();
        IPrototype clone = CloneAndDoSomeStuff(prototype);

        clone.DoOtherStuff();
    }
}
```

prototype

проблемы переиспользования: интерфейсы

```
InterfacePrototype clone = (InterfacePrototype)CloneAndDoSomeStuff(prototype);
```

**параметр-тип, ссылающийся на себя в
ограничениях наложенных на допустимые
агргументы-типы**



рекурсивный параметр-тип

prototype

рекурсивные дженерики

```
public interface IPrototype<T> where T : IPrototype<T>
{
    T Clone();

    void DoSomeStuff();
}

public class Prototype : IPrototype<Prototype>
{
    public Prototype Clone()
        ⇒ new Prototype();

    public void DoSomeStuff() { ... }

    public void DoOtherStuff() { ... }
}
```

```
public class Scenario
{
    public static T CloneAndDoSomeStuff<T>(
        T prototype) where T : IPrototype<T>
    {
        var clone = prototype.Clone();
        clone.DoSomeStuff();

        return clone;
    }

    public static void TopLevelScenario()
    {
        var prototype = new Prototype();
        Prototype clone = CloneAndDoSomeStuff(prototype);

        clone.DoOtherStuff();
    }
}
```

prototype

рекурсивные дженерики: наследование

```
public class SecondPrototype : Prototype, IPrototype<SecondPrototype>
{
    public override SecondPrototype Clone()
    {
        return new SecondPrototype();
    }
}
```


prototype

рекурсивные дженерики: проблемы

```
public interface IPrototype
{
    void DoSomeStuff() { }
}

public interface IPrototype<out T> : IPrototype where T : IPrototype
{
    T Clone();
}

public record Container(IPrototype<IPrototype> Prototype);

static void NonGeneric()
{
    var container = new Container(new Prototype());
}
```

singleton

singleton

реализация

```
public class Singleton
{
    private static readonly object _lock = new();
    private static Singleton? _instance;

    private Singleton() { }

    public static Singleton Instance
    {
        get
        {
            if (_instance is not null)
                return _instance;

            lock (_lock)
            {
                if (_instance is not null)
                    return _instance;

                return _instance = new Singleton();
            }
        }
    }
}
```

singleton

lazy

```
public class Singleton
{
    private static readonly Lazy<Singleton> _instance;

    static Singleton()
    {
        _instance = new Lazy<Singleton>(() => new Singleton(), LazyThreadSafetyMode.ExecutionAndPublication);
    }

    private Singleton() { }

    public static Singleton Instance => _instance.Value;
}
```

singleton

lazy

- None
не гарантируется потокобезопасность, при инициализации несколькими потоками, объект будет создан несколько раз, сохранённое значение не определено
- PublicationOnly
при инициализации несколькими потоками, объект будет создан несколько раз, сохранённое значение – созданное последним потоком начавшим инициализацию
- ExecutionAndPublication
полная потокобезопасность, при инициализации несколькими потоками, объект будет создан лишь один раз

singleton

недостатки

- тестирование
приватный конструктор не даёт возможности контролировать объект в тестах
- внедрение зависимостей
приватный конструктор не даёт возможности передавать значения извне
- время жизни объекта
т.к. объект инициализируется статически, его время жизни нельзя явно контролировать
- статический стейт
объект можно получить из любого места приложения, без какого-либо контроля