объектно-ориентированное программирование

порождающие паттерны

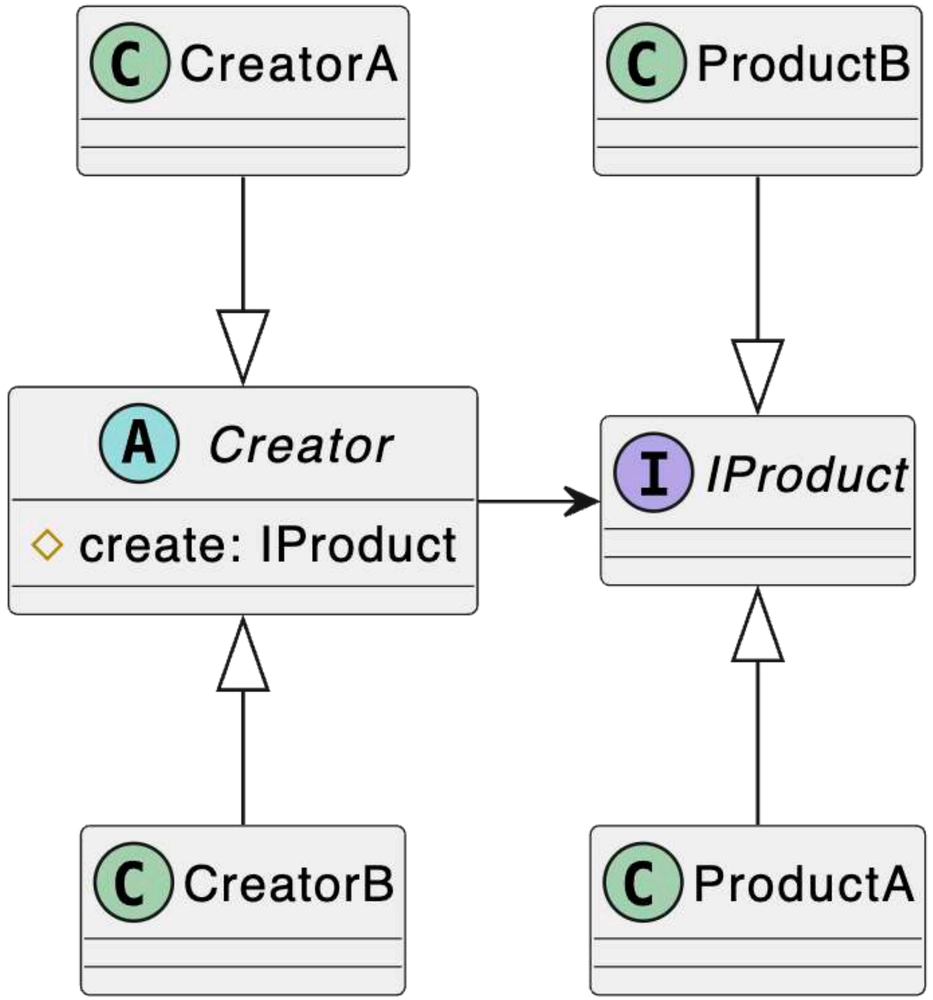
factory method

фабричный метод

схема

- 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



фабричный метод

пример применимости

```
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);
    }
}
```

фабричный метод

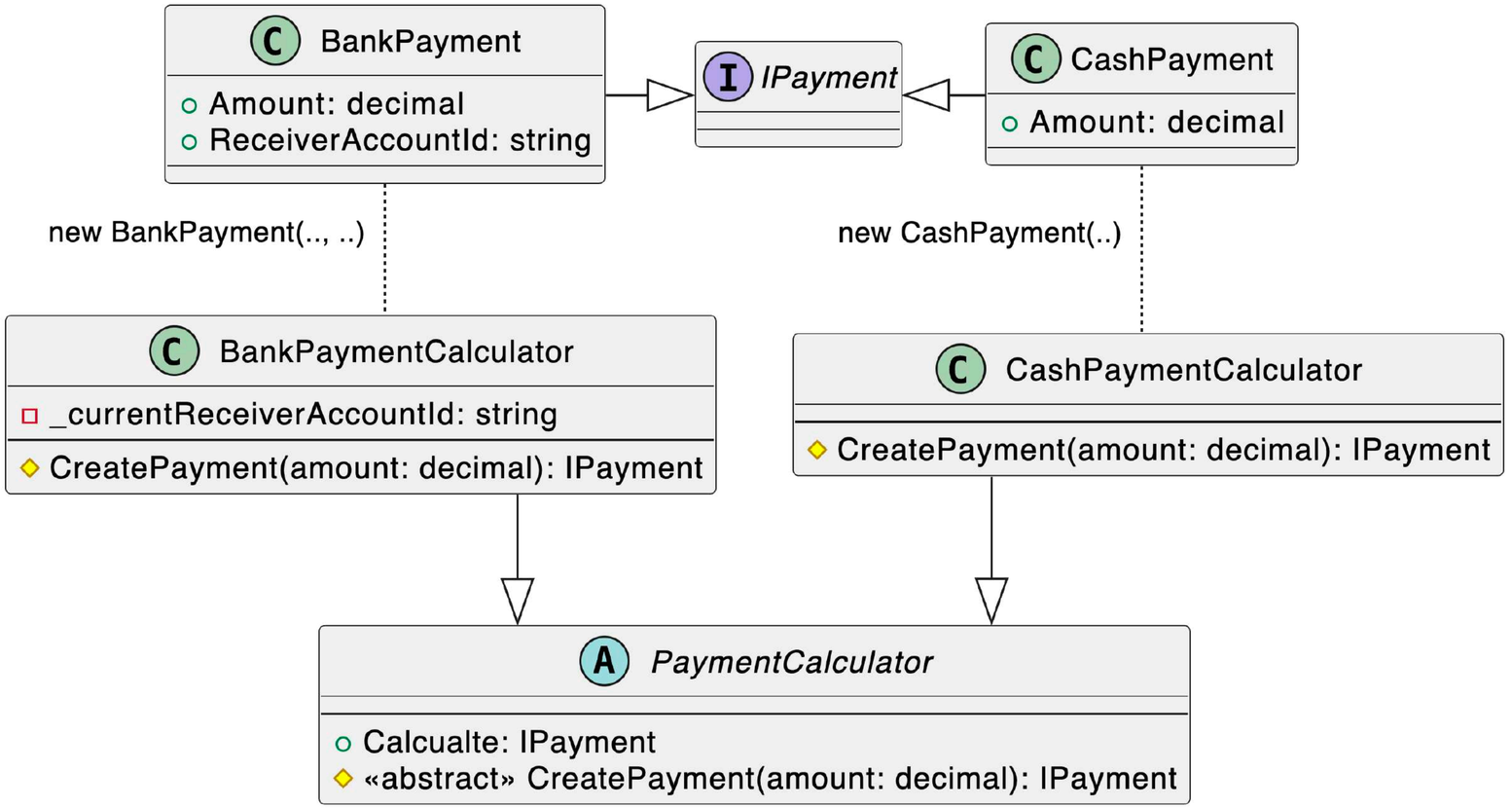
пример использования

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

public record CashPayment(decimal Amount) : IPayment;

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

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);
    }
}
```

фабричный метод

схема использования



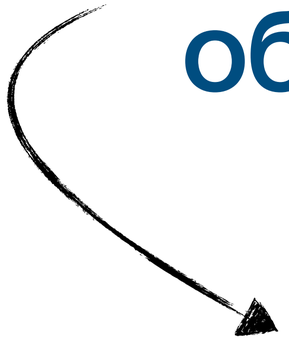
фабричный метод

недостатки

- сильная связанность реализаций
  из-за наследования переиспользования логики в конкретных создателях невозможно

- неявное нарушение SRP
  хоть реализация каждых аспектов операции (логика и создание объектов) находится в разных классах и файлах, конечный объект имеет две ответственности

разделение логики и создания объектов на иерархию типов



фабричный метод

abstract factory

абстрактная фабрика

пример использования

```
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);
    }
}
```

```
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);
    }
}
```

абстрактная фабрика

пример использования

```
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);
    }
}
```

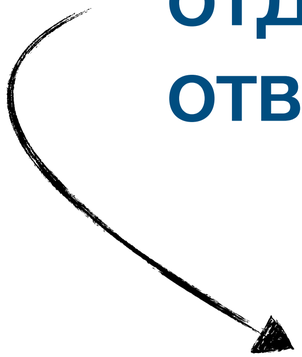
абстрактная фабрика

преимущества

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

- соблюдение OCP
  мы можем добавить в систему новые виды платежей и реализовать фабрики для них, тем самым, расширить логику не меняя реализацию калькуляторов

вынесение логики создания объектов в отдельные типы, объекты которых, будут ответственны только за это



(абстрактная) фабрика

builder

builder

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');
```

builder

виды

- convenience builder
  упрощённое создание объектов с большими конструкторами
- stateful constructor builder
  используется как конструктор, имеющий состояние

builder

convenience

```
class Service
{
    public Service(
        IDependency1? one,
        IDependency2 two,
        IDependency3 three)
    {
        ...
    }

    …
}

internal interface IDependency3 { ... }

internal interface IDependency2 { ... }

internal interface IDependency1 { ... }
```

```
class ServiceBuilder
{
    private IDependency1? _one;
    private IDependency2? _two;
    private IDependency3? _three;

    public ServiceBuilder()
    {
        _one = null;
        _two = new Dependency2();
        _three = new Dependency3();
    }

    public ServiceBuilder WithOne(IDependency1 one) { ... }

    public ServiceBuilder WithTwo(IDependency2 two) { ... }

    public ServiceBuilder WithThree(IDependency3 three) { ... }

    public Service Build()
    {
        return new Service(
            _one,
            _two ?? throw new InvalidOperationException(),
            _three ?? throw new InvalidOperationException());
    }
}
```

builder

stateful constructor

```
public class Model
{
    private Model(IReadOnlyCollection<Data> data, ...)
    {
        Data = data;
        ...
    }

    public IReadOnlyCollection<Data> Data { get; }

    public static ModelBuilder Builder => new ModelBuilder();

    public class ModelBuilder
    {
        private readonly List<Data> _data;
        ...

        public ModelBuilder AddData(Data data)
        {
            _data.Add(data);
            return this;
        }

        public Model Build()
        {
            return new Model(_data, ...);
        }
    }
}
```

builder

смешение типов

- смешивать типы builder’ов можно
- необходимость смешения скорее всего свидетельствует о необходимости декомпозиции модели
  в таком случае лучше разделить модель на несколько и реализовать для них соответствующие builder’ы
- стоит помнить что реализация builder’а зависит от модели, а не наоборот

builder

полиморфизм

```
public interface IModelBuilder
{
    ...

    Model Build();
}

public class ConcreteBuilderA : IModelBuilder
{
    ...

    public Model Build() { ... }
}

public class ConcreteBuilderB : IModelBuilder
{
    ...

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

builder

содержание созданного объекта

```
public class Model
{
    private readonly List<Value> _values = new();

    public void AddValue(Value value) { ... }
}

public class Builder
{
    private Model _model = new Model();

    public Builder AddValue(Value value) { ... }
}
```

builder

director

```
public static class BuilderDirector
{
    public static Builder DirectNumeric(
        this Builder builder,
        int count)
    {
        var enumerable = Enumerable.Range(0, count);

        foreach (var i in enumerable)
        {
            var data = new DataA(i);
            builder = builder.WithDataA(data);
        }

        return builder;
    }
}
```

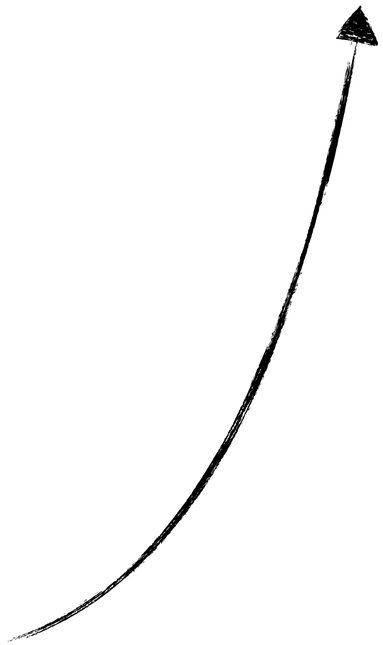
```
public interface IBuilderDirector
{
    Builder Direct(Builder builder);
}

public class InstanceDirector : IBuilderDirector
{
    private readonly int _size;
    private IEnumerable<Model> _prototypes;

    ...

    public Builder Direct(Builder builder) { ... }
}
```

builder

interface driven

```
public interface IAdressBuilder
{
    ISubjectBuilder WithAdress(string adress);
}

public interface ISubjectBuilder
{
    IEmailBuilder WithSubject(string subject);
}

public interface IEmailBuilder
{
    IEmailBuilder WithBody(string body);

    Email Build();
}

public class Email
{
    public static IAdressBuilder Builder ⇒ new EmailBuilder();

    private class EmailBuilder : IAdressBuilder, ISubjectBuilder, IEmailBuilder { }
}
```

```
var email = Email.Builder
    .WithAdress("aboba@email.com")
    .WithSubject("subject")
    .Build();
```



prototype

prototype

почему не просто конструктор?

- логика копирования может быть необходима в нескольких местах
- данные могут быть сокрыты, модифицированны 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 агрументы-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