объектно ориентированное программирование

поведенческие паттерны

наблюдатель

снижение связанности логики за счёт реактивной модели взаимодействия, публикации ивентов и их обработки

наблюдатель

структура
наблюдатель

- издатель / publisher
  сущность, о действиях с которой/об изменениях состояния которой должен знать кто-то другой

- подписчик / наблюдатель / subscriber / observer
  сущность, которая должна каким-либо образом обрабатывать какие-либо действия/изменения другой сущности

- событие
  информация о том, что с издателем было произведено какое-то действие/изменение

- подписка
  объект, представляющий связь между событием издателя и обработчиком подписчика

проблематика

наблюдатель

- сильная связанность
  - классам, выполняющим действия нужно знать обо всех обработчиках
  - нарушение OCP при расширении логики обработки
  - отсутствие стабильного контракта обработчиков
- ограниченность в логике работы с действиями

проблематика

наблюдатель

```
public class Baby
{
    private readonly Mother _mother;
    private readonly Father _father;

    public void Poop()
    {
        _father.AskMotherToWipeBaby();
        _mother.WipeBaby();
    }
}
```

```
public class Mother
{
    public void WipeBaby()
    {

    }
}
```

```
public class Father
{
    private readonly Mother _mother;

    public void AskMotherToWipeBaby()
    {
        _mother.WipeBaby();
    }
}
```

решение

наблюдатель

- выделить общий интерфейс обработчика событий – подписчика
- при выполнении действий, оповещать о событиях – публиковать
- выделить абстракцию подписки, чтобы можно было прекратить обработку событий без раскрытия деталей реализации

решение

наблюдатель

```
public interface IBabyPoopSubscriber
{
    void OnBabyPooped();
}

public interface IBabyPoopSubscription
{
    void Unsubscribe();
}
```

```
public class Baby
{
    private readonly List<IBabyPoopSubscriber> _subscribers = [];

    public IBabyPoopSubscription Subscribe(IBabyPoopSubscriber subscriber)
    {
        _subscribers.Add(subscriber);
        return new Subscription(this, subscriber);
    }

    ...

    private class Subscription(
        Baby baby,
        IBabyPoopSubscriber subscriber) : IBabyPoopSubscription
    {
        public void Unsubscribe()
        {
            baby._subscribers.Remove(subscriber);
        }
    }
}
```

решение

наблюдатель

```
public class Baby
{
   ...

    public void Poop()
    {
        Console.WriteLine("I have pooped");

        foreach (var subscriber in _subscribers)
        {
            subscriber.OnBabyPooped();
        }
    }

   ...
}
```

```
public class Mother : IBabyPoopSubscriber
{
    public void OnBabyPooped()
    {
        Console.WriteLine("Wiping...");
    }
}

public class Father : IBabyPoopSubscriber
{
    private readonly Mother _mother;

    public void OnBabyPooped()
    {
        _mother.OnBabyPooped();
    }
}
```

решение

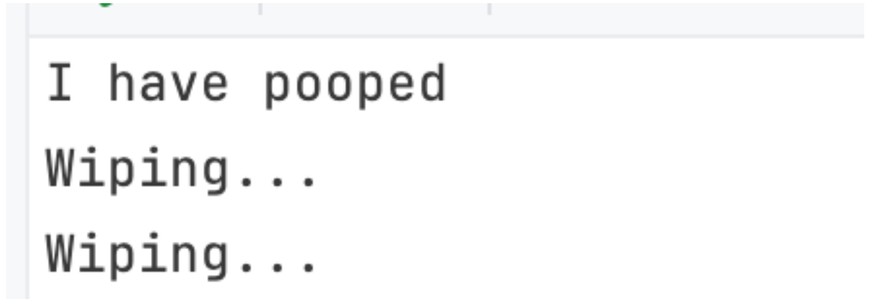
наблюдатель

```
var baby = new Baby();

var mother = new Mother();
var father = new Father(mother);

IBabyPoopSubscription motherSubscription = baby.Subscribe(mother);
IBabyPoopSubscription fatherSubscription = baby.Subscribe(father);

baby.Poop();
```

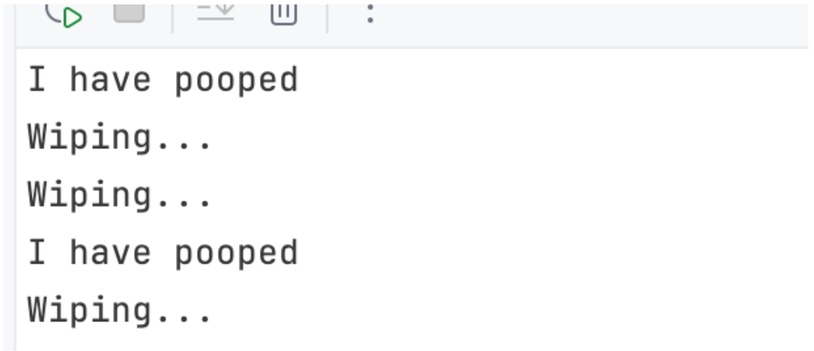


решение

наблюдатель

```
...

fatherSubscription.Unsubscribe();
baby.Poop();
```



шаблонный метод

параметризация логики, путём вынесения частей реализации в производные классы, использует наследование

шаблонный метод

структура

шаблонный метод

- шаблонный класс
  базовый абстрактный класс, содержащий какую-то логику, а также абстрактные методы, являющиеся шагами в этой логике

- шаблонный метод
  НЕ абстрактный метод, который содержит реализацию логики и вызывает абстрактные методы

- абстрактные шаги
  абстрактные методы шаблонного класса, вызываемые в шаблонном методе

- конкретные классы
  наследники шаблонного класса, реализующие абстрактные шаги

проблематика

шаблонный метод

- необходимость вариативности частей логики
- “наивная реализация” нарушает OCP
- “наивная реализация” нарушает SRP

проблематика

шаблонный метод

```
public sealed record Employee(long Id, string Name);

public sealed record EmployeeRating(int TaskCompletedCount, double HoursWorked);

public sealed record RatedEmployee(Employee Employee, EmployeeRating Rating);
```

проблематика

шаблонный метод

```
public sealed record Employee(long Id, string Name);

public sealed record EmployeeRating(int TaskCompletedCount, double HoursWorked);

public sealed record RatedEmployee(Employee Employee, EmployeeRating Rating);
```

```
public enum EvaluationCriterion
{
    Tasks,
    Hours,
}
```

проблематика

шаблонный метод

```
public class EmployeeEvaluator
{
    private readonly EvaluationCriterion _criterion;

    public EmployeeEvaluator(EvaluationCriterion criterion)
    {
        _criterion = criterion;
    }

    public IEnumerable<Employee> Evaluate(IEnumerable<RatedEmployee> employees)
    {
        IOrderedEnumerable<RatedEmployee> sortedEmployees = _criterion switch
        {
            EvaluationCriterion.Tasks ⇒ employees.OrderByDescending(x ⇒ x.Rating.TaskCompletedCount),
            EvaluationCriterion.Hours ⇒ employees.OrderByDescending(x ⇒ x.Rating.HoursWorked),
            _ ⇒ throw new ArgumentOutOfRangeException()
        };

        return sortedEmployees.Select(x ⇒ x.Employee);
    }
}
```

решение

шаблонный метод

- выделить шаблонный класс и шаблонный метод
- оставить в шаблонном методе общую логику
- изменяющуюся логику вынести в наследники

решение

шаблонный метод

```
public abstract class EmployeeEvaluatorBase
{
    public IEnumerable<Employee> Evaluate(IEnumerable<RatedEmployee> employees)
    {
        var sortedEmployees = SortEmployees(employees);
        return sortedEmployees.Select(x ⇒ x.Employee);
    }

    protected abstract IOrderedEnumerable<RatedEmployee> SortEmployees(IEnumerable<RatedEmployee> employees);
}
```

решение

шаблонный метод

```
public sealed class TaskEmployeeEvaluator : EmployeeEvaluatorBase
{
    protected override IOrderedEnumerable<RatedEmployee> SortEmployees(IEnumerable<RatedEmployee> employees)
        => employees.OrderByDescending(x => x.Rating.TaskCompletedCount);
}

public sealed class HoursEmployeeEvaluator : EmployeeEvaluatorBase
{
    protected override IOrderedEnumerable<RatedEmployee> SortEmployees(IEnumerable<RatedEmployee> employees)
        => employees.OrderByDescending(x => x.Rating.HoursWorked);
}
```

недостатки

шаблонный метод

- сильная связанность конкретных классов с шаблонным классом
  - невозможность переиспользования реализаций шагов
  - комбинаторный взрыв при >1 абстрактных шагов
- техническое нарушение SRP
- потенциальное нарушение OCP

стратегия

параметризация логики, путём вынесения частей реализации за отдельные интерфейсы, использует полиморфизм и композицию

стратегия

проблематика

стратегия

- такая же как с случае **шаблонного метода**
- недостатки **шаблонного метода**

решение

стратегия

- выделить отдельные интерфейсы для варьируемых шагов
- использовать композицию для связи параметризируемой логики с реализацией шагов

решение

стратегия

```
public interface IEmployeeEvaluationCriterion
{
    IOrderedEnumerable<RatedEmployee> SortEmployees(
        IEnumerable<RatedEmployee> employees);
}
```

```
public sealed class TaskEvaluationCriterion : IEmployeeEvaluationCriterion
{
    public IOrderedEnumerable<RatedEmployee> SortEmployees(IEnumerable<RatedEmployee> employees)
        => employees.OrderByDescending(x => x.Rating.TaskCompletedCount);
}
```

```
public sealed class HoursEvaluationCriterion : IEmployeeEvaluationCriterion
{
    public IOrderedEnumerable<RatedEmployee> SortEmployees(IEnumerable<RatedEmployee> employees)
        => employees.OrderByDescending(x => x.Rating.HoursWorked);
}
```

решение

стратегия

```
public sealed class EmployeeEvaluator
{
    private readonly IEmployeeEvaluationCriterion _criterion;

    public EmployeeEvaluatorBase(IEmployeeEvaluationCriterion criterion)
    {
        _criterion = criterion;
    }

    public IEnumerable<Employee> Evaluate(IEnumerable<RatedEmployee> employees)
    {
        var sortedEmployees = _criterion.SortEmployees(employees);
        return sortedEmployees.Select(x ⇒ x.Employee);
    }
}
```

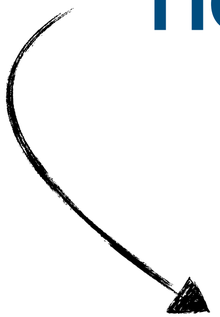
преимущества

стратегия

- “абстрактные шаги” теперь не зависят от конкретной реализации
  - есть возможность их переиспользовать
- использование композиции позволяет простым образом комбинировать разные реализации шагов
  если шагов >1
- соблюдаем SRP
- соблюдаем OCP
  добавление новых шагов в существующую логику не влияет на существующие реализации шагов

визитор

выделение логики обработки сложной, полиморфной структуры в отдельные типы



визитор

проблематика

визитор

- наличие сложной полиморфной структуры (компоновщик)
- наличие различных сценариев работы с этой структурой
  - в большинстве своём эти сценарии не связаны с логикой структуры напрямую
- “наивная реализация” (1) нарушает SRP
- “наивная реализация” (2) нарушает OCP

проблематика

визитор

```
public interface IFileSystemComponent
{
    string Name { get; }
}

public class FileFileSystemComponent : IFileSystemComponent
{
    public string Name { get; }
}

public class DirectoryFileSystemComponent : IFileSystemComponent
{
    public string Name { get; }
    public IReadOnlyCollection<IFileSystemComponent> Components { get; }
}
```

проблематика

визитор

```
public interface IFileSystemComponent
{
    string Name { get; }

    string GetFormattedString(string padding);
}
```

наивное решение (1)

визитор

```
public interface IFileSystemComponent
{
    ...

    string GetFormattedString(string padding);
}
```

```
public class FileFileSystemComponent : IFileSystemComponent
{
    ...

    public string GetFormattedString(int padding)
        => $"{new string(' ', padding)}{Name}";
}
```

наивное решение (1)

визитор

```
public class DirectoryFileSystemComponent : IFileSystemComponent
{
    ...

    public string GetFormattedString(int padding)
    {
        var builder = new StringBuilder();
        builder.Append(new string(' ', padding));
        builder.AppendLine(Name);

        foreach (IFileSystemComponent component in Components)
        {
            builder.AppendLine(component.GetFormattedString(padding + 1));
        }

        return builder.ToString();
    }
}
```

наивное решение (2)

визитор

```
public static class FileSystemComponentFormatter
{
    public static string FormatComponent(IFileSystemComponent component, int padding)
    {
        return component switch
        {
            DirectoryFileSystemComponent dir ⇒ FormatDirectory(dir, padding),
            FileFileSystemComponent file ⇒ FormatFile(file, padding),
            _ ⇒ throw new ArgumentOutOfRangeException(nameof(component)),
        };
    }

    private static string FormatFile(FileFileSystemComponent component, int padding)
        ⇒ $"{new string(' ', padding)}{component.Name}";

    private static string FormatDirectory(DirectoryFileSystemComponent component, int padding)
    {
        ...

        foreach (IFileSystemComponent child in component.Components)
        {
            builder.AppendLine(FormatComponent(child, padding));
        }

        ...
    }
}
```

решение

визитор

- выделить отдельный интерфейс, отвечающий за обработку каждого узла
- переложить ответственность за вызов метода обработки на сами узлы

решение

визитор

```
public interface IFileSystemComponentVisitor
{
    void Visit(FileFileSystemComponent component);

    void Visit(DirectoryFileSystemComponent component);
}
```

```
public interface IFileSystemComponent
{
    ...

    void Accept(IFileSystemComponentVisitor visitor);
}
```

double dispatching

визитор

- диспетчеризация – передача потока выполнения другому методу
- используется два вида полиморфизма
  - динамический, чтобы вызвать Accept
  - статический, чтобы вызвать нужную перегрузку Visit

```
    public void Accept(IFileSystemComponentVisitor visitor)
    {
        visitor.Visit( component: this);
    }
}
```

```
public abstract void Visit(FileFileSystemComponent component)
 in interface IFileSystemComponentVisitor
```

1 usage

реализация

визитор

```
public sealed class FormattingVisitor : IFileSystemComponentVisitor
{
    private readonly StringBuilder _builder = new();
    private int _padding;

    public string Value => _builder.ToString();

    public void Visit(FileFileSystemComponent component)
    {
        _builder.Append(' ', _padding);
        _builder.AppendLine(component.Name);
    }

    public void Visit(DirectoryFileSystemComponent component)
    {
        _builder.Append(' ', _padding);
        _builder.AppendLine(component.Name);

        _padding += 1;

        foreach (IFileSystemComponent child in component.Components)
        {
            child.Accept(this);
        }

        _padding += 1;
    }
}
```

почему лучше (1)

визитор

- соблюдаем SRP
  - сущности не содержат логику, которая к ним не относится непосредственно
- локализация логики
  - в данном случае группировка должна производиться по операции, реализуемой визитором

почему лучше (2)

визитор

- безопасная расширяемость
  - необходимость обработки всех новых узлов на уровне компиляции
- более низкая связанность
  - не завязываемся на конкретные типы входных данных
  - позволяем узлам контролировать собственную обработку
- единый контракт работы с компонентами

на что обращать внимание

визитор

- только для логики, не принадлежащей сущностям непосредственно
  - моделирование логики сущностей через визиторы/double dispatching приводит к сильной связанности
- не нарушаем DIP
  - интерфейсы визиторов должны быть максимально отстранены от логики их реализаций
  - не возвращаем ничего из методов Visit/Accept, визиторы сами ответственны за составление состояния обхода

ИТОГ

визитор

- способ отделить от сущности логику, не принадлежащую ей непосредственно
- способ обязать обработку новых узлов на уровне компиляции
  - предотвращаем ситуацию, когда существующий функционал не работает с новыми типами узлов из-за ошибок/забывчивости

цепочка обязанностей

декомпозиция условной логики в объектную модель

объектно ориентированный switch-case/if-else

цепочка обязанностей

проблематика

цепочка обязанностей

- наличие большого количества условной логики
  - логика относительно состояния объекта
  - логика относительно типа объекта
  - логика относительно состояния полиморфных объектов
- необходимость динамически добавлять/убирать эти условия
- дубликация условной логики в различных реализациях

проблематика

цепочка обязанностей

```
public abstract record DiscountRequestBase(
    decimal TotalCost,
    long CustomerOrderNumber,
    CustomerStatus CustomerStatus);

public sealed record CouponDiscountRequest(
    ...
    string CouponCode,
    DateTimeOffset CouponExpiresAt,
    decimal? RequiredCost) : DiscountRequestBase(...);

public sealed record BonusPointsDiscoundRequest(
    ...
    int PointsToSpend,
    int CustomerPointsBalance) : DiscountRequestBase(...);
```

```
public interface IDiscountService
{
    DiscountResult Apply(DiscountRequestBase request);
}
```

проблематика

цепочка обязанностей

```
public sealed class WebDiscountService : IDiscountService
{
    public DiscountResult Apply(DiscountRequestBase request)
    {
        if (request.TotalCost < 0
            || request.CustomerStatus is CustomerStatus.Banned)
        {
            return new DiscountResult.Inapplicable();
        }

        if (request is CouponDiscountRequest coupon
            && (coupon.CouponExpiresAt < DateTimeOffset.UtcNow
                || coupon.TotalCost < coupon.RequiredCost
                || (coupon.CustomerStatus is not CustomerStatus.Important
                    && coupon.CustomerOrderNumber == 1)))
        {
            return new DiscountResult.Inapplicable();
        }

        if (request is BonusPointsDiscoundRequest bonus
            && (bonus.PointsToSpend < 0
                || bonus.PointsToSpend > bonus.CustomerPointsBalance
                || bonus.TotalCost < 500))
        {
            return new DiscountResult.Inapplicable();
        }

        ...
    }
}
```

```
public sealed class MobileDiscountService : IDiscountService
{
    public DiscountResult Apply(DiscountRequestBase request)
    {
        if (request.TotalCost < 0
            || request.CustomerStatus is CustomerStatus.Banned)
        {
            return new DiscountResult.Inapplicable();
        }

        if (request is CouponDiscountRequest coupon
            && (coupon.CouponExpiresAt < DateTimeOffset.UtcNow
                || coupon.TotalCost < coupon.RequiredCost
                || coupon.CustomerOrderNumber != 1))
        {
            return new DiscountResult.Inapplicable();
        }

        if (request is BonusPointsDiscoundRequest bonus
            && (bonus.PointsToSpend < 0
                || bonus.PointsToSpend > bonus.CustomerPointsBalance
                || bonus.TotalCost / 2 > bonus.PointsToSpend))
        {
            return new DiscountResult.Inapplicable();
        }

        ...
    }
}
```

решение

цепочка обязанностей

- представить логику в виде отдельных шагов выполнения операции – звеньев
- выделить звенья, отвечающие за валидацию
- выделить звенья, отвечающие за реализацию
- структурировать эти звенья в виде связного списка
  - позволяет элементу цепочки прекратить её выполнение

решение

цепочка обязанностей

```
public interface IDiscountService
{
    DiscountResult Apply(DiscountRequestBase request);
}

public interface IDiscountLink : IDiscountService
{
    IDiscountLink AddNext(IDiscountLink link);
}
```

```
public abstract class DiscountLinkBase : IDiscountLink
{
    private IDiscountLink? _next;

    public abstract DiscountResult Apply(DiscountRequestBase request);

    public IDiscountLink AddNext(IDiscountLink link)
    {
        if (_next is null)
        {
            _next = link;
        }
        else
        {
            _next.AddNext(link);
        }

        return link;
    }

    protected DiscountResult CallNext(DiscountRequestBase request)
    {
        return _next?.Apply(request)
               ?? throw new InvalidOperationException("Chain missing terminal link");
    }
}
```

решение

цепочка обязанностей

```
public class TotalCostValidationLink : DiscountLinkBase
{
    public override DiscountResult Apply(DiscountRequestBase request)
    {
        if (request.TotalCost < 0)
            return new DiscountResult.Inapplicable();

        return CallNext(request);
    }
}
```

```
public class CouponExpirationValidationLink : DiscountLinkBase
{
    public override DiscountResult Apply(DiscountRequestBase request)
    {
        if (request is CouponDiscountRequest coupon
            && coupon.CouponExpiresAt < DateTimeOffset.UtcNow)
        {
            return new DiscountResult.Inapplicable();
        }

        return CallNext(request);
    }
}
```

решение

цепочка обязанностей

```
public interface IDiscountServiceFactory
{
    IDiscountService Create();
}

public sealed class WebDiscountServiceFactory : IDiscountServiceFactory
{
    public IDiscountService Create()
    {
        return new TotalCostValidationLink()
            .AddNext(new CustomerStatusValidationLink())
            .AddNext(new CouponExpirationValidationLink())
            .AddNext(new CouponCostValidationLink())
            .AddNext(new CouponCustomOrderNumberValidationLink())
            .AddNext(new BonusPointsAmountValidationLink())
            .AddNext(new BonusCostValidationLink());
    }
}
```

ИТОГ

цепочка обязанностей

- соблюдаем SRP посредствам декомпозиции
- избавляемся от дублирования повторяющейся логики
- соблюдаем OCP посредствам отделения типо-зависимой логики
- получаем гибкий, модульный дизайн

отличия от визитора

цепочка обязанностей

- logic-oriented vs component-oriented
  цепочка подразумевает разную логику над разными объектами
  визитор подразумевает одну логику над всеми объектами

- разная логика над одними типами компонентов
  цепочка может иметь несколько звеньев работающих с одними типами объектов
  визитор имеет одну логику для обработки одного типа объектов

- динамичность логики
  цепочка подразумевает составление различных итоговых поведений из различных кусков логики
  [конкретный] визитор подразумевает реализацию одного поведения над полиморфной структурой

- применимость к компоновщику
  цепочка не особо применима к компоновщику, тогда как визитор основной паттерн к нему в пару