Черная магия паттерна «Посетитель»

Дмитрий Нестерук

@dnesteruk

Что мы рассмотрим

- Зачем нужен «посетитель»
- Интрузивный подход
- Рефлекторный посетитель
  - + функциональные вариации
- Что такое dispatch?
- Динамический посетитель
- Классическая реализация (double dispatch)
- Ацикличный посетитель
- Трансформации (map-reduce)

Сценарий

- Вы прочитали (распарсили) математическое выражение
  - Например “1+2+3”
- У вас есть объектная модель, и вы хотите
  - Вычислить значение
  - Оптимизировать выражение
  - Вывести выражение на экран
  - Трансформировать выражение во что-то ещё
  - И т.д.

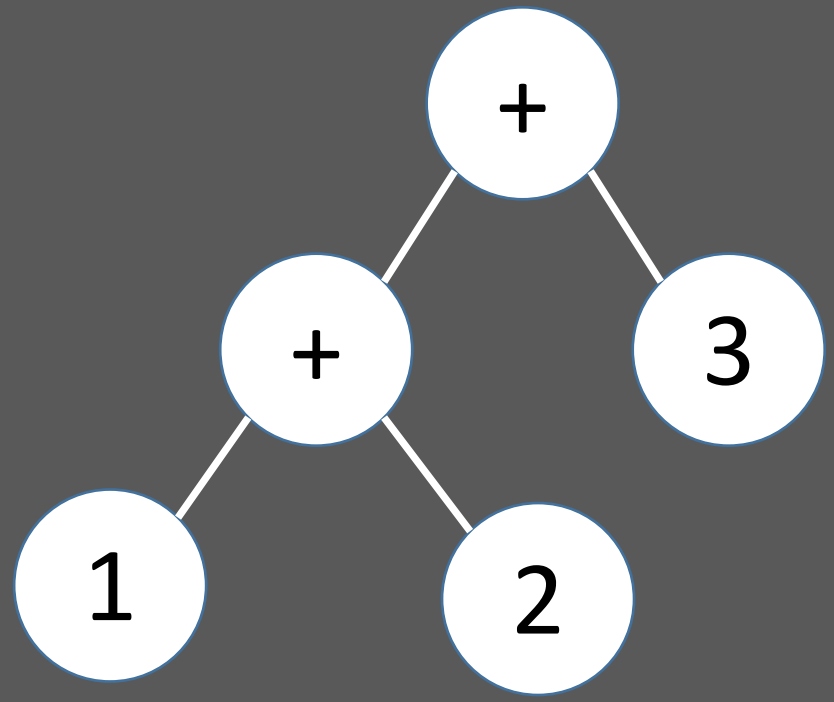
Выражение — это дерево

```
public abstract class Expression {}

public class DoubleExpression : Expression
{
  public readonly double Value;
  public DoubleExpression(double value) => Value = value;
}

public class AdditionExpression : Expression
{
  public readonly Expression Left, Right;

  public AdditionExpression(Expression left, Expression right)
  {
    Left = left;
    Right = right;
  }
}
```



Паттерн “Посетитель”

- Объект AdditionExpression
  - Может обладать бесконечной вложенностью
  - Например, AE{AE{AE{2,3},4}, AE{5,6}}
  - Полиморфизм: содержит какие-то Expression, но в compile-time их тип неизвестен
- Дерево нужно обходить рекурсивно
- Посетитель = компонента, которая обходит дерево
- Где должен быть основной функционал обхода?
  - Жестко зашит внутри самой иерархии
  - Внешняя компонента

Интрузивный подход

- Мы хотим чтобы каждый Expression умел печатать себя как текст
  - Например, в выданный ему StringBuilder
- Самое очевидное – добавить набор соответствующих методов Print() в каждый класс-наследник
- Intrusive = вламываемся и меняем уже написанный код
- Грубое нарушение принципа открыт-закрыт (Open-Closed Principle)
- Добавлять функционал придется во все интерфейсы/классы иерархии
- Это придется делать для каждого набора функционала

Интрузивненько:

```
public abstract class Expression
{
  // adding a new operation
  public abstract void Print(StringBuilder sb);
}

public class DoubleExpression : Expression
{ …
  public override void Print(StringBuilder sb)
  {
    sb.Append(value);
  }
}

public class AdditionExpression : Expression
{ …
  public override void Print(StringBuilder sb)
  {
    sb.Append("(");
    left.Print(sb);
    sb.Append("+");
    right.Print(sb);
    sb.Append(")");
  }
}
```

- В данном случае, посетитель = StringBuilder
- Нарушение принцип открыт-закрыт
- Нарушение принципа единственной ответственности
- Любые вариации в алгоритме обхода порождают N новых методов
- Подход работает только в очень простых случаях

Принцип единственной ответственности

- Элементы дерева – это POCO структуры
  - Можно использовать record types
- Дополнительный функционал лучше держать в отдельных компонентах
  - Общие настройки
  - Когнитивно проще
- Проблема реализации отдельных посетителей в полиморфном поведении

Интрузивный подход ±

+ Простота
+ Легко дебажить
+ Красивый API из коробки

– Требует внедрения N методов для каждого нового посетителя
– Нарушает принцип открыт-закрыт
– Нарушает принцип единственной ответственности

Рефлективный подход

- Все-таки хочется чтобы посетитель был отдельным классом
- Принцип разделения ответственности (Single Responsibility Principle, Separation of Concerns)
- Удаляем все Print() из иерархии
- Пытаемся написать отдельную компоненту

Вот так не получится ☹

```
public static void Print(DoubleExpression de,
  StringBuilder sb)
{
  sb.Append(de.Value);
}

public static void Print(AdditionExpression ae,
  StringBuilder sb)
{
  sb.Append("(");
  Print(ae.Left, sb);  // will not compile
  sb.Append("+");
  Print(ae.Right, sb); // will not compile
  sb.Append(")");
}
```

- Проблема с полиморфностью
- ae.Left и ae.Right имеют тип Expression
- Компилятор не знает их конкретный тип
- Компилятор не может выбрать правильную перегрузку Print() в рантайме

Реализация

```
public static void Print(Expression e, StringBuilder sb)
{
  switch (e)
  {
    case DoubleExpression de:
      sb.Append(de.Value);
      break;
    case AdditionExpression ae:
      sb.Append("(");
      Print(ae.Left, sb);
      sb.Append("+");
      Print(ae.Right, sb);
      sb.Append(")");
      break;
    default:
      // your choice what to do here
      throw new Exception("Unsupported expression type");
  }
}
```

Структурирование

```
using DictType = Dictionary<Type, Action<Expression, StringBuilder>>;

private static DictType actions = new()
{
  [typeof(DoubleExpression)] = (e, sb) =>
  {
    var de = (DoubleExpression) e;
    sb.Append(de.Value);
  },
  [typeof(AdditionExpression)] = (e, sb) =>
  {
    var ae = (AdditionExpression) e;
    sb.Append("(");
    Print(ae.Left, sb);
    sb.Append("+");
    Print(ae.Right, sb);
    sb.Append(")");
  }
};

public static void Print(this Expression e, StringBuilder sb)
{
  actions[e.GetType()](e, sb);
}
```

Рефлективный подход ±

+ Принцип единственной ответственности
+ Не меняет иерархию, не требует сорцов
+ Одно место для дебага всего и вся

– Нет проверок что покрыты все типы из иерархии (а надо ли?)
– Перформанс (не критично, но все же…)
– Чувствителен к порядку проверок (любой Child должен быть выявлен до Parent, иначе unreachable code)

Функционально рефлективный подход

- Допустим мы решили использовать рефлективный подход
- Но хочется создавать новые посетители…
- Список типов придется проверять в каждом посетителе
- Дупликация кода – ай-ай-ай!
- Хочется весь этот набор кастов куда-то вынести:
  - В базовый класс Visitor
  - В корень иерархии, в функциональном стиле (частично intrusive)

Добавляем в Expression

```
public void Match(
  Action<DoubleExpression> visitDoubleExpression,
  Action<AdditionExpression> visitAdditionExpression,
  Action<AbsoluteDoubleExpression> visitAbsoluteDoubleExpression,
  Action<Expression> visitUnknownExpression = null)
{
  switch (this)
  {
    case AbsoluteDoubleExpression e:
      visitAbsoluteDoubleExpression(e);
      break;
    case DoubleExpression e:
      visitDoubleExpression(e);
      break;
    case AdditionExpression e:
      visitAdditionExpression(e);
      break;
    default:
      visitUnknownExpression?.Invoke(this);
      break;
  }
}
```

Как это использовать?

```
public class ExpressionPrinter
{
  private string Visit(DoubleExpression de)
  {
    return de.Value.ToString();
  }

  private string Visit(AdditionExpression ae)
  {
    return "(" + Print(ae.Left) + "+" + Print(ae.Right) + ")";
  }

  private string Visit(AbsoluteDoubleExpression ade)
  {
    return $"|{ade.Value}|";
  }

  public string Print(Expression e)
  {
    return e.Match(Visit, Visit, Visit);
  }
}
```

Функционально рефлективный подход ±

+ Список всех посещаемых типов теперь в одном месте

+ API требует передать обработчик на каждый известный тип

+ Класс-посетитель максимально упрощается: набор интуитивных перегрузок одного и того же метода

– Чувствительность к порядку (Child нужно проверять до Parent)

– Требует изменения корневого элемента иерархии при добавлении в иерархию новых классов

– Вызов e.Match(Visit, Visit, Visit, …) выглядит несколько странно (хотя все прекрасно работает)

Динамический посетитель

- В момент компиляции перегрузку точно не выбрать
- В момент исполнения ее можно было бы выбрать, но C# это не умеет делать
- Python смог бы
- Но в C# есть поддержка динамических языков
- Используем `dynamic`

Вот так не получится ☹

```
public static void Print(DoubleExpression de,
  StringBuilder sb)
{
  sb.Append(de.Value);
}

public static void Print(AdditionExpression ae,
  StringBuilder sb)
{
  sb.Append("(");
  Print(ae.Left, sb);  // will not compile
  sb.Append("+");
  Print(ae.Right, sb); // will not compile
  sb.Append(")");
}
```

А вот так сработает ☺

```
public static void Print(DoubleExpression de,
  StringBuilder sb)
{
  sb.Append(de.Value);
}

public static void Print(AdditionExpression ae,
  StringBuilder sb)
{
  sb.Append("(");
  Print((dynamic)ae.Left, sb);  // ok
  sb.Append("+");
  Print((dynamic)ae.Right, sb); // ok
  sb.Append(")");
}
```

- Использование dynamic откладывает выбор перегрузок в рантайм
- Runtime dispatch
- В момент исполнения, рантайм вызовет правильный метод Print()
- Если его нет, будет исключение ☹

Динамический посетитель ±

+ Проще читать и писать
+ Каждый обработчик в своем методе
+ Код иерархии неизменный; применим к чужому коду

– Существенные тормоза при исполнении
– Отсутствие нужного обработчика = runtime exception (но можно сделать catch-all для корня иерархии)
– Посетителей нельзя наследовать друг от друга, т.к. динамические вызовы не работают с наследованием (!)

Насчет dispatch

- Dispatch = вопрос о том, сколько нужно информации чтобы вызвать правильный метод
- Почему мы не можем вызвать Print(ae.Left)?
- Потому что нам заранее неизвестен конкретный тип ae.Left
  - Это может быть DoubleExpression, AdditionExpression, etc.
- А где этот тип точно известен?
  - Внутри самого ae.Left
  - Кем бы не был ae.Left, у него есть this
- Итого, для вызова правильной перегрузки нам нужно два куска информации:
  - Кто и какой метод вызывает
  - Конкретный тип аргумента чтобы правильно выбрать перегрузку
- Чтобы это заработало, нужен double dispatch

Классика!

- Классический подход отчасти интрузивен
- Мы все же меняем иерархию, добавляя каждому классу метод `void Accept(IExpressionVisitor v)`
- Это подразумевает, что у всех посетителей есть общий интерфейс (ну, или базовый класс)
- Тело метода Accept() в каждом классе иерархии идентично:
- `void Accept(IExpressionVisitor v) { v.Visit(this); }`
- Грубое нарушение DRY?

Как это работает

```
public class DoubleExpression : Expression
{
  public readonly double Value;

  public DoubleExpression(double value) => Value = value;

  public override void Accept(IExpressionVisitor visitor)
    => visitor.Visit(this);
}
```

```
public void Visit(AdditionExpression ae)
{
  sb.Append("(");
  ae.Left.Accept(this);
  sb.Append("+");
  ae.Right.Accept(this);
  sb.Append(")");
}
```

```
public void Visit(DoubleExpression de)
{
  sb.Append(de.Value);
}
```

```
public class AdditionExpression : Expression
{
  public readonly Expression Left, Right;

  public AdditionExpression(Expression left, Expression right)
  {
    Left = left;
    Right = right;
    Left.Parent = Right.Parent = this;
  }

  public override void Accept(IExpressionVisitor visitor)
  {
    visitor.Visit(this);
  }
}
```

Классический DD посетитель ±

+ Иерархия меняется, но только один раз

+ Нет тормозов от проверки типов

+ Посетитель всегда (по определению) поддерживает все типы в иерархии

+ Новый посетитель просто переиспользует существующую инфраструктуру

– Иерархию все же надо менять (нужен доступ к сорцам)

– Добавление нового элемента иерархии автоматически означает изменение всей иерархии посетителей

– Немного неинтуитивно и усложняет отладку

– Цикличная зависимость между иерархией и посетителем

Проверки типов все равно будут!

- Например, пусть у нас есть еще и MultiplicationExpression
- Мы можем закодировать (2+3)*5 ≠ 2+3*5
- Тогда мы должны ставить скобки вокруг AdditionExpression
- Но только если их родитель = MultiplicationExpression

```
public void Visit(AdditionExpression ae)
{
  bool needBraces = ae.Parent is MultiplicationExpression;
  if (needBraces) sb.Append("(");
  ae.Left.Accept(this);
  sb.Append("+");
  ae.Right.Accept(this);
  if (needBraces) sb.Append(")");
}
```

Трансформации

- Два способа реализовать «посетитель»
  - Все Accept() возвращают void; посещение базируется на side-effect’ах
  - Все Accept() возвращают унифицированный тип T
- Второй способ дает делать
  - Map (преобразование каждого из элементов в тип T)
  - Reduce (свертывание всех элементов в один T)
  - Map-reduce (преобразование и свертывание)
- Эти реальности могут существовать параллельно:
  - Классика: (I)Visitor/Visit()/Accept()
  - Трансформатор: ITransformer/Transform()/Reduce()

Корневой метод

```
public abstract class Expression
{
  public abstract T Reduce<T>(ITransformer<T> transformer);
}
```

- Возрадуемся generic методам
- Реализация аналогична Accept() – во всех классах она одинакова:

```
public override T Reduce<T>(ITransformer<T> transformer)
{
  return transformer.Transform(this);
}
```

Интерфейс для посещения элементов

```
public interface ITransformer<T>
{
  T Transform(DoubleExpression de);
  T Transform(AdditionExpression ae);
}
```

Трансформатор для рассчета значения

```
public class EvaluationTransformer : ITransformer<double>
{
  public double Transform(DoubleExpression de) => de.Value;

  public double Transform(AdditionExpression ae)
  {
    return ae.Left.Reduce(this) + ae.Right.Reduce(this);
  }
}
```

Трансформатор для печати

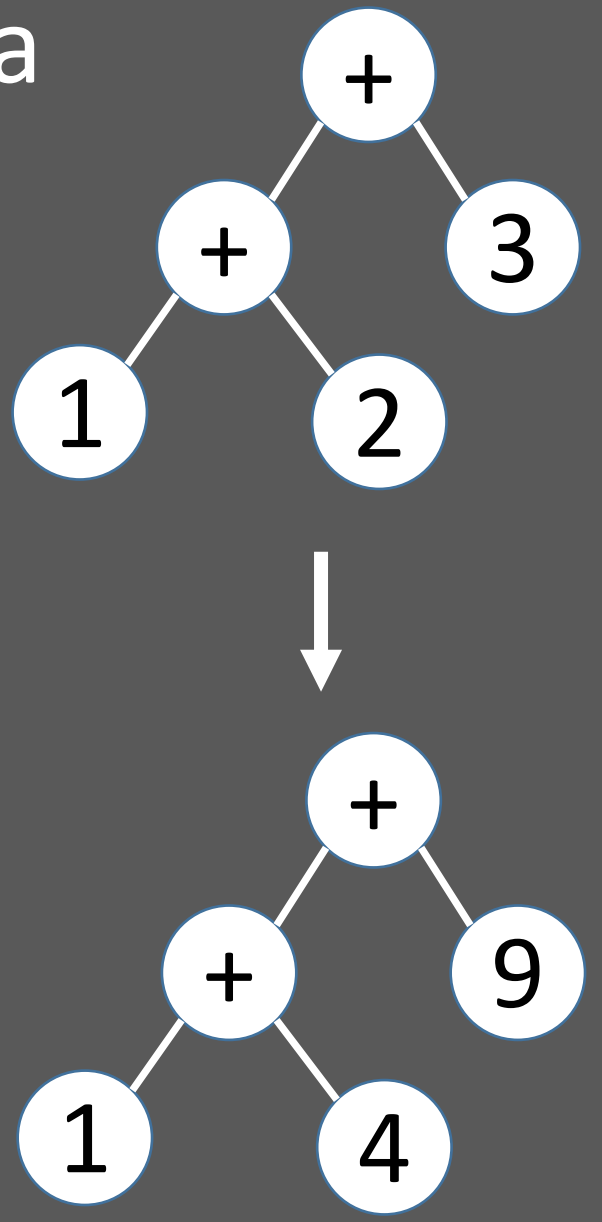
```
public class PrintTransformer : ITransformer<string>
{
  public string Transform(DoubleExpression de)
  {
    return de.Value.ToString();
  }

  public string Transform(AdditionExpression ae)
  {
    return $"({ae.Left.Reduce(this)} + {ae.Right.Reduce(this)})";
  }
}
```

Простое преобразование дерева

```
public class SquareTransformer : ITransformer<Expression>
{
  public Expression Transform(DoubleExpression de)
  {
    return new DoubleExpression(de.Value * de.Value);
  }

  public Expression Transform(AdditionExpression ae)
  {
    return new AdditionExpression(
      ae.Left.Reduce(this),
      ae.Right.Reduce(this));
  }
}
```



Трансформации ±

± Все те же что и у классического DD посетителя, но…

+ Не нужно пологаться на сайд-эффекты в посетителе ∴ посетитель может быть stateless/static/singleton

+ Более понятный код

+ Делает возможными map/reduce сценарии

± Один посетитель может реализовывать более одного обхода типов (реализуя ITransformer для разных типов T)

– Требует чтобы все трансформации сводились к одному типу

Адаптер для посетителей

- Иногда мы получаем данные в слабо типизированном формате
- Например, JSON который описывает бизнес-правила
- Создавать иерархию для этого типа
  - Лень, т.к. она нужна только для обхода посетителем
  - Невозможно т.к. мы не знаем все возможные типы
- Все что нам нужно — это адаптировать данные для посетителя чтобы состоялся обход

Немного неструктурированного JSON

```
{
  "Name": "Operator",
  "Value": "And",
  "Children": [
    {
      "Name": "Operator",
      "Value": "Greater",
      "Children": [
        {
          "Name": "Age",
          "Value": "16",
          "Children": []
        }
      ]
    },
    {
      "Name": "Operator",
      "Value": "Equal",
      "Children": [
        {
          "Name": "Citizen",
          "Value": "True",
          "Children": []
        }
]}]}
```

- Тут закодировано выражение `((Age > 16) && (Citizen == True))`
- Как его напечатать текстом не создавая иерархию классов?
- Решение:
  - Делаем посетитель, которые умеет посещать основные типы
  - Вместо иерархии типов, пишем обобщенный адаптер для всех типов (т.е. один единственный класс)

Интерфейс посетителя

```
{
  "Name": "Operator",
  "Value": "And",
  "Children": [
    {
      "Name": "Operator",
      "Value": "Greater",
      "Children": [
        {
          "Name": "Age",
          "Value": "16",
          "Children": []
        }
      ]
    },
    {
      "Name": "Operator",
      "Value": "Equal",
      "Children": [
        {
          "Name": "Citizen",
          "Value": "True",
          "Children": []
        }
]}]}
```

```
public interface IVisitor
{
  void VisitBinaryOp(Node node, string op);
  void VisitInlineOp(Node node, string op);
}
```

- BinaryOp = оператор у которого 2 Children (&&)
- InlineOp = оператор у которого 1 Child (==, >)

- Оба метода принимают тип Node

Node

- Node = record type для хранения информации
- Содержит имя, значение, и children
- Никакой типизации, просто рекурсивный POCO

```
public sealed record Node(string Name,
  string Value, List<Node> Children);
```

Посещение

- Нам нужно чтобы наш record type был посещяем типом IVisitor
- Вводим соответствующий интерфейс

```
public interface IVisitable
{
  void Accept(IVisitor visitor);
}
```

- Теперь, можно реализовать этот интерфейс
  - В самом Node
  - Сделать адаптер VisitableNode

VisitableNode

```
public class VisitableNode : IVisitable
{
  private readonly Node node;
  public VisitableNode(Node node) => this.node = node;

  public void Accept(IVisitor visitor)
  {
    if (node.Name == "Operator")
    {
      switch (node.Value)
      {
        case "Equal":
          visitor.VisitInlineOp(node.Children[0], "==");
          break;
        case "Greater":
          visitor.VisitInlineOp(node.Children[0], ">");
          break;
        case "And":
          visitor.VisitBinaryOp(node, "&&");
          break; }}}}
```

- Получает Node
- В методе Accept проверяет содержание
- Вызывает соответствующий метод посетителя с дополнительной информацией

Посетитель

```
public class NodePrinter : IVisitor
{
  private StringBuilder sb = new();

  public void VisitBinaryOp(Node node, string op)
  {
    sb.Append("(");
    node.Children[0].ToVisitable().Accept(this);
    sb.Append($" {op} ");
    node.Children[1].ToVisitable().Accept(this);
    sb.Append(")");
  }

  public void VisitInlineOp(Node node, string op)
  {
    sb.Append($"({node.Name} {op} {node.Value})");
  }

  public override string ToString() => sb.ToString();
}
```

- Поскольку посещаемый тип один, приходится давать разные имена
- Поведение схоже, но везде приходится вызывать `ToVisitable()` для создания промежуточных адаптеров

Использование адаптера

```
public static class NodeExtensions
{
  public static IVisitable ToVisitable(this Node node)
  {
    return new VisitableNode(node);
  }
}

class Program
{
  static void Main(string[] args)
  {
    var node = JsonConvert.DeserializeObject<Node>(
      File.ReadAllText("input.json"));
    NodePrinter p = new();
    node.ToVisitable().Accept(p);
    Console.WriteLine(p);
    // ((Age > 16) && (Citizen == True))
  }
}
```

Абстрактный класс или интерфейс?

Абстрактный класс

+ Может иметь no-op реализацию методов

+ Может иметь поля

+ Поддерживает различные полезные плюшки (например, перегрузку операторов)

+ Виртуальные вызовы

– Не дает классам унаследовать что-то еще

Интерфейс

+ Можно нашпиговать класс (или обычными или дженерик-вариантами)

+ Поддержка в struct-ах

+ Можно делать дефолтные реализации (но нужно кастовать)

– No state

Ацикличный посетитель

- В классической реализации, у посетителя и посещаемого цикличная зависимость
  - Базовый класс иерархии (Expression) зависит от базового класса/интерфейса посетителея (Visitor/IVisitor)
  - Visitor обязан иметь перегрузки для всех наследников Expression
  - Каждый наследник Expression явным образом зависит от Expression
- Цикл зависимостей: Expression непрямо зависит от всех наследников
- Каждый новый наследник Expression требует изменения Visitor
- Ацикличный посетитель = попытка разорвать/облегчить зависимость

Ацикличный посетитель

```
public interface IVisitor {}
// marker interface

public interface IVisitor<in TVisitable>
{
  void Visit(TVisitable obj);
}
```

- Два интерфейса!
- Пустой маркер-интерфейс IVisitor позволяет объектам принимать *любой* посетитель
- IVisitor<T> индикатор того, что посетитель умеет посещать конкретный тип T
- Тип T также проверяет посещаемый объект

Accept()

```
public virtual void Accept(IVisitor visitor)
{
  if (visitor is IVisitor<Expression> typed)
    typed.Visit(this);
}
```

- Все классы реализуют один и тот же `Accept()` (опять дупликация!)
- Внутри проверяется, что к нам пришел посетитель, который умеет посещать именно нас
- Если да, вызываем `Visit()`
- Если нет, ничего страшного

А вот и посетитель

- Посетитель реализует как маркер IVisitor так и набор IVisitor<T> для всех знакомых типов
- Можно даже иметь catch-вариант для верха иерархии
- Заметьте: нет никакого базового класса Visitor
- Каждый посетитель обрабатывает то, что хочет

```
public class ExpressionPrinter : IVisitor,
  IVisitor<Expression>,
  IVisitor<DoubleExpression>,
  IVisitor<AdditionExpression>
{
  private readonly StringBuilder sb = new();

  public void Visit(DoubleExpression de)
  {
    sb.Append(de.Value);
  }

  public void Visit(AdditionExpression ae)
  {
    sb.Append("(");
    ae.Left.Accept(this);
    sb.Append("+");
    ae.Right.Accept(this);
    sb.Append(")");
  }

  public void Visit(Expression obj)
  {
    // default handler
  }

  public override string ToString() => sb.ToString();
}
```

Acyclic visitor ±

+ Изменение иерархии типов

+ Изменение посетителей

+ Избирательность и декларативность (я посещаю только X,Y,Z)

+ Посетителей можно группировать через наследование интерфейсов

+ Проверок типов существенно меньше чем в рефлективном посетителе

- Немного сложнее чем классический подход

Null Visitable

- Null Object иногда появляется в контексте использования visitor
- Например, допустим у нас есть новый Expression под названием Variable, т.е. переменная у которой есть имя (напр x)

```
public record Variable(string Name) : INode
{
  public void Accept(IVisitor visitor)
  {
    visitor.Visit(this);
  }
}
```

- Вопрос: как в дереве представить выражение $-x$?

Как записать $-x$?

- Делать новый тип лень
- Добавлять флаг negative как-то странно
- А что если SubtractionExpression(0, x)?
- Это сработает, но принтер напишет не -x а 0-x
- Значит нужно заменить 0 чем-то, что не печатается но в целях вычислений все еще остается нулем
- Null object на помощь!

NullLiteral

```
public sealed record NullLiteral : Literal
{
  public NullLiteral() : base(0) {}

  public override void Accept(IVisitor visitor)
  {
    if (visitor is not ExpressionPrinter)
      base.Accept(visitor);
  }
}
```

- Два варианта реализации
- Либо все посетители игнорируют NullLiteral
- Но тогда как считать значение выражения?
- Или же NullLiteral не дает себя обработать именно ExpressionPrinter
- Сильная связанность кода ☹

Какую реализацию использовать?

- Есть доступ к сорцам
  - Мне «только спросить» - Intrusive
  - Чтобы хорошо и качественно – Classic или Acyclic
  - Иерархия или посетители будут меняться – Acyclic
  - Иерархия не будет кардинально меняться – Functional Reflective
- Доступа к сорцам нет
  - Перформанс не критичен – Dynamic
  - Иначе – Reflective
- Вообще структуры никакой нет – Visitor Adapter

На этом всё!

- Фидбэк → @dnesteruk
- Курс по Паттернам в C#/.NET на Udemy
  - Русский язык: https://bit.ly/3ffkkJw
  - Английский язык: https://bit.ly/39WZPO3
- Книга Design Patterns in .NET

- Спасибо за внимание!