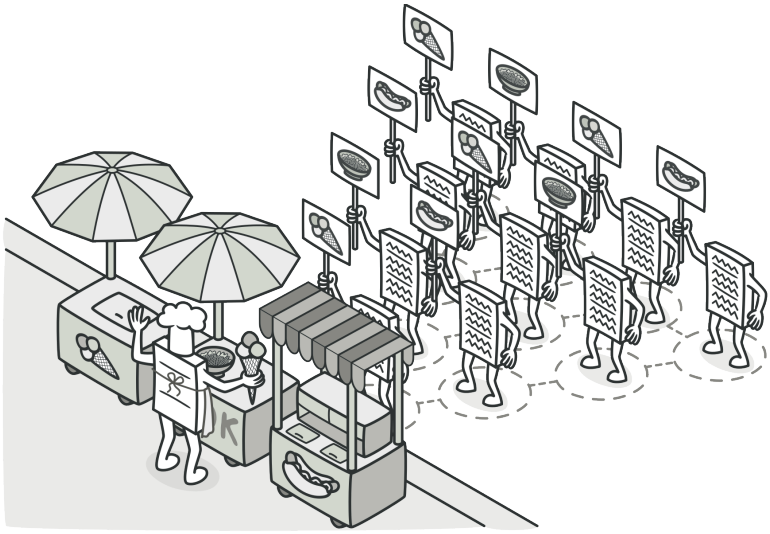


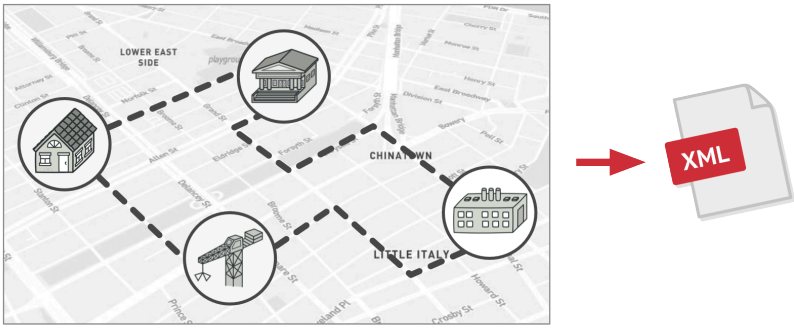
ПОСЕТИТЕЛЬ

Также известен как: Visitor

Посетитель — это поведенческий паттерн проектирования, который позволяет создавать новые операции, не меняя классы объектов, над которыми эти операции могут выполняться.

☹ Проблема

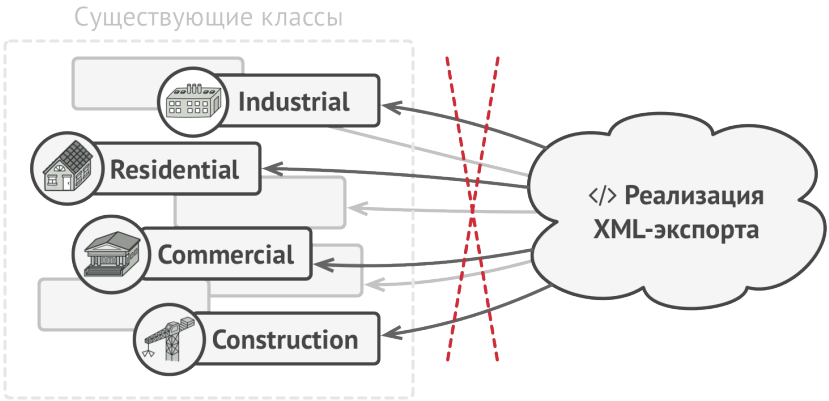
Ваша команда разрабатывает приложение, работающее с геоданными в виде графа. Узлами графа могут быть как города, так и другие локации, будь то достопримечательности, большие предприятия и так далее. Каждый узел имеет ссылки на другие, ближайшие к нему узлы. Для каждого типа узла имеется свой класс, а каждый узел представлен отдельным объектом.



Экспорт гео-узлов в XML.

Ваша задача — сделать экспорт этого графа в XML. Дело было бы плёвым, если бы вы могли редактировать классы узлов. В этом случае, можно было бы добавить метод экспорта в каждый тип узла, а затем, перебирая узлы географического графа, вызывать этот метод для каждого узла. Решение получилось бы изящным, так как полиморфизм позволил бы не привязываться к конкретным классам узлов.

Но, к сожалению, классы узлов вам изменить не получилось. Системный архитектор сослался на то, что код классов узлов сейчас очень стабилен и от него многое зависит, поэтому он не хочет рисковать и позволять кому-либо его трогать.



Код XML-экспорта придётся добавить во все классы узлов, а это слишком накладно.

К тому же он сомневался в том, что экспорт в XML вообще уместен в рамках этих классов. Их основная задача была связана с геоданными, а экспорт выглядит как чужеродное поведение в рамках этих классов.

Была и ещё одна причина запрета. На следующей неделе мог понадобиться экспорт в какой-то другой формат данных, и вам снова пришлось бы трогать эти классы.

☺ Решение

Паттерн Посетитель предлагает разместить новое поведение в отдельном классе, вместо того, чтобы множить его сразу в нескольких классах. Объекты, с которыми должно было быть связано поведение, не будут выполнять его самостоятельно. Вместо этого, вы будете передавать эти объекты в методы посетителя.

Код поведения скорей всего должен отличаться для объектов разных классов, поэтому и методов у посетителя должно быть несколько. Названия и принцип действия этих методов будет схож, но основное отличие будет в типе принимаемого в параметрах объекта, например:

```
class ExportVisitor implements Visitor is
  method doForCity(City c) { ... }
  method doForIndustry(Industry f) { ... }
  method doForSightSeeing(SightSeeing ss) { ... }
  // ...
```

Здесь возникает вопрос, каким образом мы будет подавать узлы в объект посетитель. Так как все методы имеют отличающуюся сигнатуру, использовать полиморфизм при переборе узлов не получится. Придётся проверять тип узлов для того, чтобы выбрать соответствующий метод посетителя.

```
foreach (Node node : graph)
  if (node instanceof City)
    exportVisitor.doForCity((City) node);
  if (node instanceof Industry)
    exportVisitor.doForIndustry((Industry) node);
  // ...
```

Тут не поможет даже механизм перегрузки методов (доступный в Java и C#). Если назвать все методы одинаково, то неопределённость реального типа узла всё равно не даст вызвать правильный метод. Механизм перегрузки всё время будет вызывать метод посетителя, соответствующий типу `Node`, а не реального класса поданного узла.

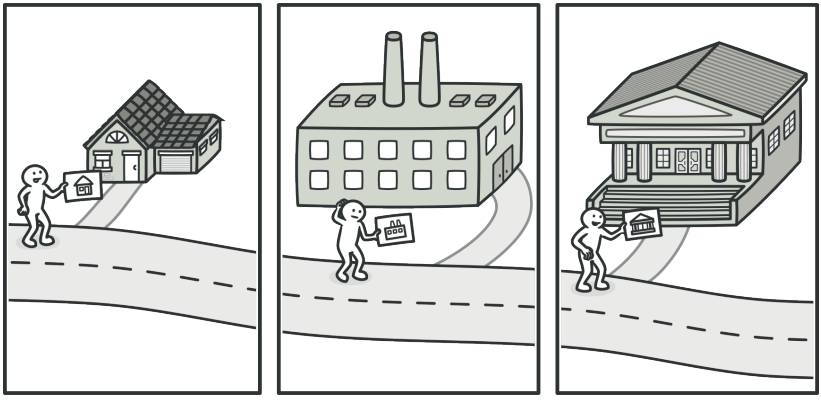
Но паттерн Посетитель решает и эту проблему, используя механизм **двойной диспетчеризации**. Вместо того чтобы самим искать нужный метод, мы можем поручить это объектам, которые передаём в параметрах посетителю. А они уже вызовут правильный метод посетителя.

```
// Client code
foreach (Node node : graph)
  node.accept(exportVisitor);
// City
class City is
  method accept(Visitor v) is
    v.doForCity(this);
  // ...
```

```
// Industry
class Industry is
  method accept(Visitor v) is
    v.doForIndustry(this);
  // ...
```

Как видите, изменить классы узлов всё-таки придётся. Но это простое изменение в любое время позволит применять к объектам узлов и другие поведения. Ведь классы узлов будут привязаны не к конкретному классу посетителей, а к их общему интерфейсу. Поэтому если придётся добавить в программу новое поведение, вы создадите новый класс посетителей, реализующий общий интерфейс, и будете передавать его в методы узлов.

Аналогия из жизни

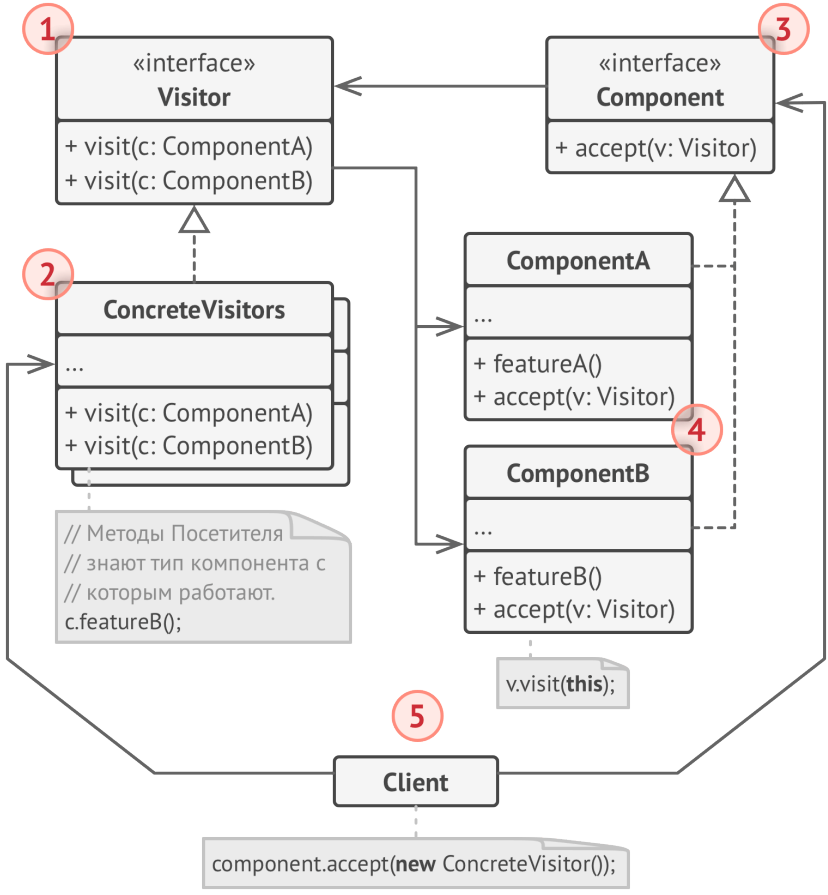


У страхового агента приготовлены полисы для разных видов организаций.

Представьте начинающего страхового агента, жаждущего получить новых клиентов. Он беспорядочно посещает все дома в округе, предлагая свои услуги. Но для каждого из «типов» домов, которые он посещает, у него имеется особое предложение.

- Придя в дом к обычной семье, он предлагает оформить медицинскую страховку.
- Придя в банк, он предлагает страховку от грабежа.
- Придя на фабрику, он предлагает страховку предприятия от пожара и наводнения.

Структура



1. **Посетитель** описывает общий интерфейс для всех типов посетителей. Он объявляет набор методов, которые принимают различные классы компонентов в качестве параметров. В языках, поддерживающих перегрузку методов, эти методы могут иметь одинаковые имена, но типы их параметров должны отличаться.

2. **Конкретные посетители** реализуют какое-то особенное поведение для всех типов компонентов, которые можно подать через методы интерфейс посетителя.

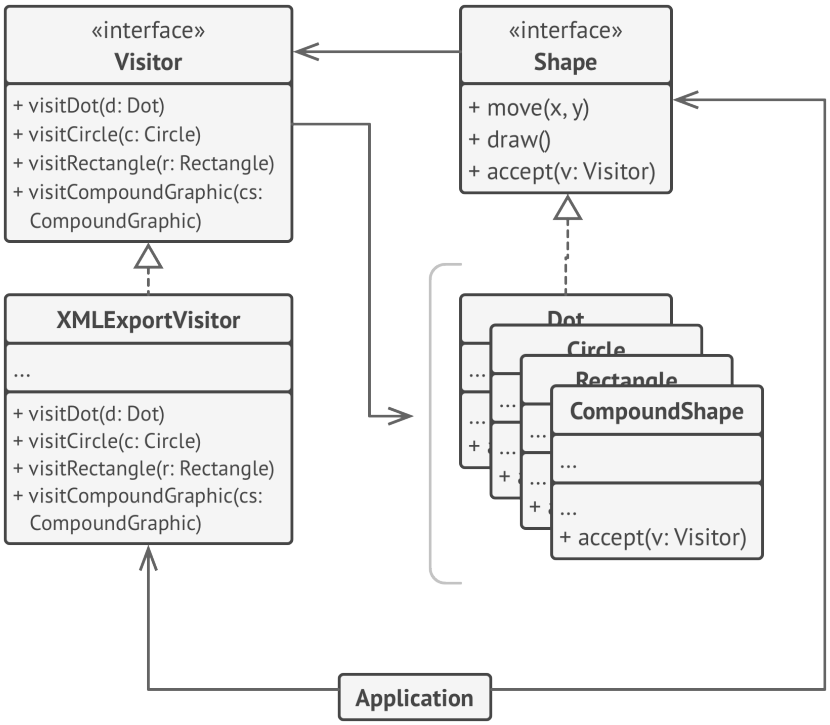
3. **Компонент** описывает метод *принятия* посетителя. Этот метод должен иметь единственный параметр, объявленный с типом интерфейса посетителя.

4. **Конкретные компоненты** реализуют методы *принятия* посетителя. Цель этого метода — вызвать тот метод посещения, который соответствует типу этого компонента. Так посетитель узнает, с каким именно компонентом он работает.

5. **Клиентом** зачастую выступает коллекция или сложный составной объект (например, дерево **Компоновщика**). Клиент не знает конкретные классы своих компонентов.

Псевдокод

В этом примере **Посетитель** добавляет в существующую иерархию классов геометрических фигур возможность экспорта в XML.



Пример организации экспорта объектов в XML через отдельный класс-посетитель.

```
// Сложная иерархия компонентов.
interface Shape is
  method move(x, y)
  method draw()
  method accept(v: Visitor)

// Метод принятия посетителя должен быть реализован в каждом
// компоненте, а не только в базовом классе. Это поможет
// программе определить какой метод посетителя нужно
// вызвать, в случае если вы не знаете тип компонента.
class Dot extends Shape is
  // ...
```

```
  method accept(v: Visitor) is
    v.visitDot(this)

class Circle extends Dot is
  // ...
  method accept(v: Visitor) is
    v.visitCircle(this)

class Rectangle extends Shape is
  // ...
  method accept(v: Visitor) is
    v.visitRectangle(this)

class CompoundShape implements Shape is
  // ...
  method accept(v: Visitor) is
    v.visitCompoundShape(this)


// Интерфейс посетителей должен содержать методы посещения
// каждого компонента. Важно, чтобы иерархия компонентов
// менялась редко, так как при добавлении нового компонента
// придётся менять всех существующих посетителей.
interface Visitor is
  method visitDot(d: Dot)
  method visitCircle(c: Circle)
  method visitRectangle(r: Rectangle)
  method visitCompoundShape(cs: CompoundShape)

// Конкретный посетитель реализует одну операцию для всей
// иерархии компонентов. Новая операция = новый посетитель.
// Посетитель выгодно применять, когда новые компоненты
// добавляются очень редко, а команды добавляются
// очень часто.
```

```
class XMLExportVisitor is
  method visitDot(d: Dot) is
    // Экспорт id и кординатов центра точки.

  method visitCircle(c: Circle) is
    // Экспорт id, кординатов центра и радиуса окружности.

  method visitRectangle(r: Rectangle) is
    // Экспорт id, кординатов левого-верхнего угла, ширины
    // и высоты прямоугольника.

  method visitCompoundShape(cs: CompoundShape) is
    // Экспорт id составной фигуры, а также списка id
    // подфигур, из которых она состоит.


// Приложение может применять посетителя к любому набору
// объектов компонентов, даже не уточняя их типы. Нужный
// метод посетителя будет выбран благодаря проходу через
// метод accept.
class Application is
  field allShapes: array of Shapes

  method export() is
    exportVisitor = new XMLExportVisitor()

    foreach (shape in allShapes) do
      shape.accept(exportVisitor)
```

Вам не кажется, что вызов метода `accept` – это лишнее звено здесь? Если так, то ещё раз рекомендую вам ознакомиться с проблемой раннего и позднего связывания в статье **Посетитель и Double Dispatch**.

Применимость

Когда вам нужно выполнить операцию над всеми элементами сложной структуры объектов (например, деревом).

Посетитель позволяет применять одну и ту же операцию к объектам различных классов.

Когда над объектами сложной структуры объектов надо выполнять некоторые, не связанные между собой операций, но вы не хотите «засорять» классы такими операциями.

Посетитель позволяет извлечь родственные операции из классов, составляющих структуру объектов, поместив их в один класс-посетитель. Если структура объектов является общей для нескольких приложений, то паттерн позволит в каждое приложение включить только нужные операции.

Когда новое поведение имеет смысл только для некоторых классов из существующей иерархии.

Посетитель позволяет определить поведение только для этих классов и оставить его пустым для всех остальных.

☑ Шаги реализации

1. Создайте интерфейс посетителя и объявите в нём методы «посещения» для каждого класса компонента, который существует в программе.

2. Опишите интерфейс компонентов. Если вы работаете с уже существующими классами, то объявите абстрактный метод принятия посетителей в базовом классе иерархии компонентов.

3. Реализуйте методы принятия во всех конкретных компонентах. Они должны переадресовывать вызовы тому методу посетителя, в котором класс параметра совпадает с текущим классом компонента.

4. Иерархия компонентов должна знать только о базовом интерфейсе посетителей. С другой стороны, посетители будут знать обо всех классах компонентов.

5. Для каждого нового поведения создайте свой конкретный класс. Приспособьте это поведение для всех посещаемых компонентов, реализовав все методы интерфейса посетителей.

   Вы можете столкнуться с ситуацией, когда посетителю нужен будет доступ к приватным полям компонентов. В этом случае, вы можете либо раскрыть доступ к этим полям, нарушив инкапсуляцию компонентов, либо сделать класс

посетителя вложенным в класс компонента, если вам повезло писать на языке, который поддерживает вложенность классов.

6. Клиент будет создавать объекты посетителей, а затем передавать их компонентам, используя метод принятия.

⚖ Преимущества и недостатки

✔ Упрощает добавление новых операций над всей связанной структурой объектов.

✔ Объединяет родственные операции в одном классе.

✔ Посетитель может накоплять состояние при обходе структуры компонентов.

✖ Паттерн неоправдан, если иерархия компонентов часто меняется.

✖ Может привести к нарушению инкапсуляции компонентов.

⇄ Отношения с другими паттернами

- **Посетитель** можно рассматривать как расширенный аналог **Команды**, который способен работать сразу с несколькими видами получателей.

- Вы можете выполнить какое-то действие над всем деревом **Компоновщика** при помощи **Посетителя**.

- **Посетитель** можно использовать совместно с **Итератором**. *Итератор* будет отвечать за обход структуры данных, а *Посетитель* — за выполнение действий над каждым её компонентом.