**Посетитель**

**Также известен как:**Visitor

**Суть паттерна**

**Посетитель** — это поведенческий паттерн проектирования, который позволяет добавлять в программу новые операции, не изменяя классы объектов, над которыми эти операции могут выполняться.

**Проблема**

Ваша команда разрабатывает приложение, работающее с геоданными в виде графа. Узлами графа являются городские локации: памятники, театры, рестораны, важные предприятия и прочее. Каждый узел имеет ссылки на другие, ближайшие к нему узлы. Каждому типу узлов соответствует свой класс, а каждый узел представлен отдельным объектом.

Ваша задача — сделать экспорт этого графа в XML. Дело было бы плёвым, если бы вы могли редактировать классы узлов. Достаточно было бы добавить метод экспорта в каждый тип узла, а затем, перебирая узлы графа, вызывать этот метод для каждого узла. Благодаря полиморфизму, решение получилось бы изящным, так как вам не пришлось бы привязываться к конкретным классам узлов.

Но, к сожалению, классы узлов вам изменить не удалось. 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.

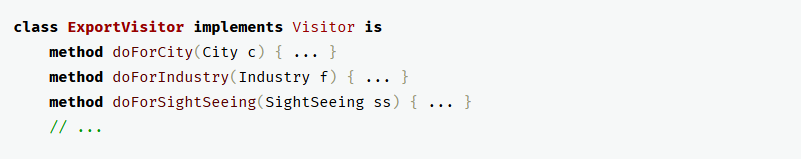
К тому же он сомневался в том, что экспорт в XML вообще уместен в рамках этих классов. Их основная задача была связана с геоданными, а экспорт выглядит в рамках этих классов чужеродно.

Была и ещё одна причина запрета. Если на следующей неделе вам бы понадобился экспорт в какой-то другой формат данных, то эти классы снова пришлось бы менять.

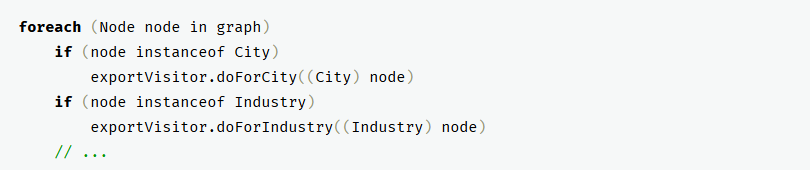
**Решение**

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

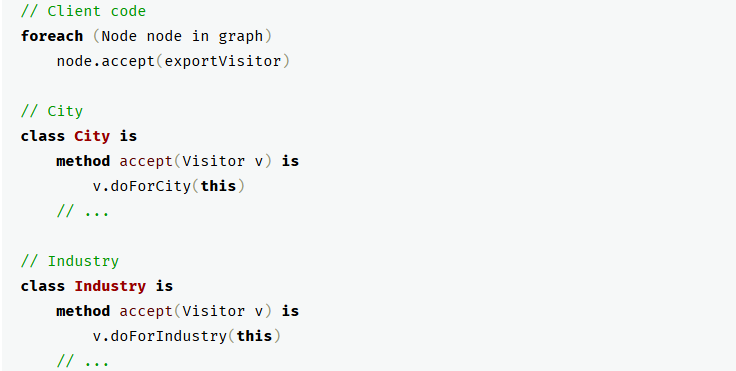


Здесь возникает вопрос: как подавать узлы в объект-посетитель? Так как все методы имеют отличающуюся сигнатуру, использовать полиморфизм при переборе узлов не получится. Придётся проверять тип узлов для того, чтобы выбрать соответствующий метод посетителя.

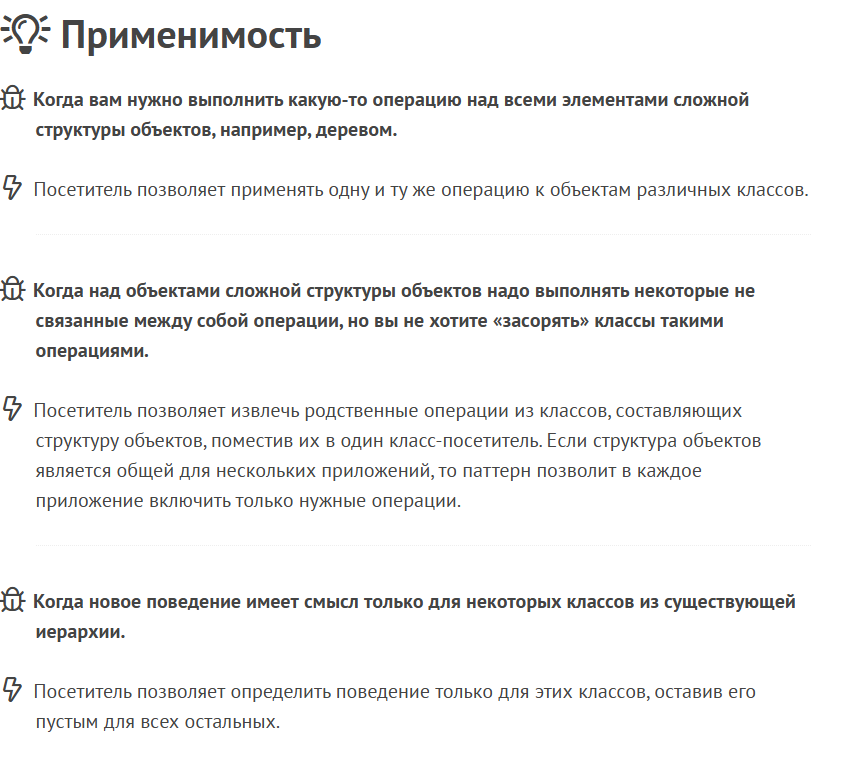


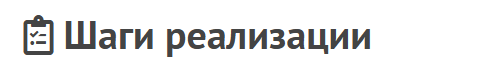
Тут не поможет даже механизм перегрузки методов (доступный в Java и C#). Если назвать все методы одинаково, то неопределённость реального типа узла всё равно не даст вызвать правильный метод. Механизм перегрузки всё время будет вызывать метод посетителя, соответствующий типу Node, а не реального класса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

Но паттерн Посетитель решает и эту проблему, используя механизм [**двойной диспетчеризации**](https://refactoring.guru/ru/design-patterns/visitor-double-dispatch). Вместо того, чтобы самим искать нужный метод, мы можем поручить это объектам, которые передаём в параметрах посетителю. А они уже вызовут правильный метод посетителя.



Как видите, изменить классы узлов всё-таки придётся. Но это простое изменение 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 и будете передавать его в методы узлов.

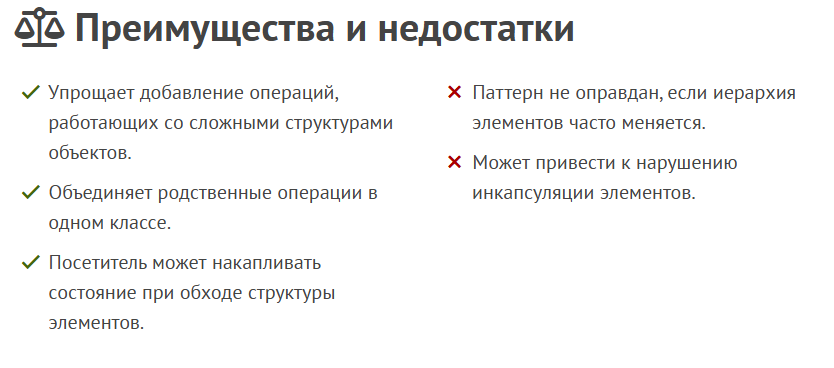




1. Создайте интерфейс посетителя и объявите в нём методы «посещения» для каждого класса элемента, который существует в программе.
2. Опишите интерфейс элементов. Если вы работаете с уже существующими классами, то объявите абстрактный метод принятия посетителей в базовом классе иерархии элементов.
3. Реализуйте методы принятия во всех конкретных элементах. Они должны переадресовывать вызовы тому методу посетителя, в котором тип параметра совпадает с текущим классом элемента.
4. Иерархия элементов должна знать только о базовом интерфейсе посетителей. С другой стороны, посетители будут знать обо всех классах элементов.
5. Для каждого нового поведения создайте конкретный класс посетителя. Приспособьте это поведение для работы со всеми типами элементов, реализовав все методы интерфейса посетителей.

Вы можете столкнуться с ситуацией, когда посетителю нужен будет доступ к приватным полям элементов. В этом случае вы можете либо раскрыть доступ к этим полям, нарушив инкапсуляцию элементов, либо сделать класс посетителя вложенным в класс элемента, если вам повезло писать на языке, который поддерживает вложенность классов.

1. Клиент будет создавать объекты посетителей, а затем передавать их элементам, используя метод принятия.





* [Посетитель](https://refactoring.guru/ru/design-patterns/visitor) можно рассматривать как расширенный аналог [Команды](https://refactoring.guru/ru/design-patterns/command), который способен работать сразу с несколькими видами получателей.
* Вы можете выполнить какое-то действие над всем деревом [Компоновщика](https://refactoring.guru/ru/design-patterns/composite) при помощи [Посетителя](https://refactoring.guru/ru/design-patterns/visitor).
* [Посетитель](https://refactoring.guru/ru/design-patterns/visitor) можно использовать совместно с [Итератором](https://refactoring.guru/ru/design-patterns/iterator). Итератор будет отвечать за обход структуры данных, а Посетитель — за выполнение действий над каждым её компонентом.