참고자료

- https://refactoring.guru/design-patterns/visitor

Intent

분류

- behavioral design pattern
    - object의 behavior을 캡슐화하고 요청을 object에 위임하는 패턴

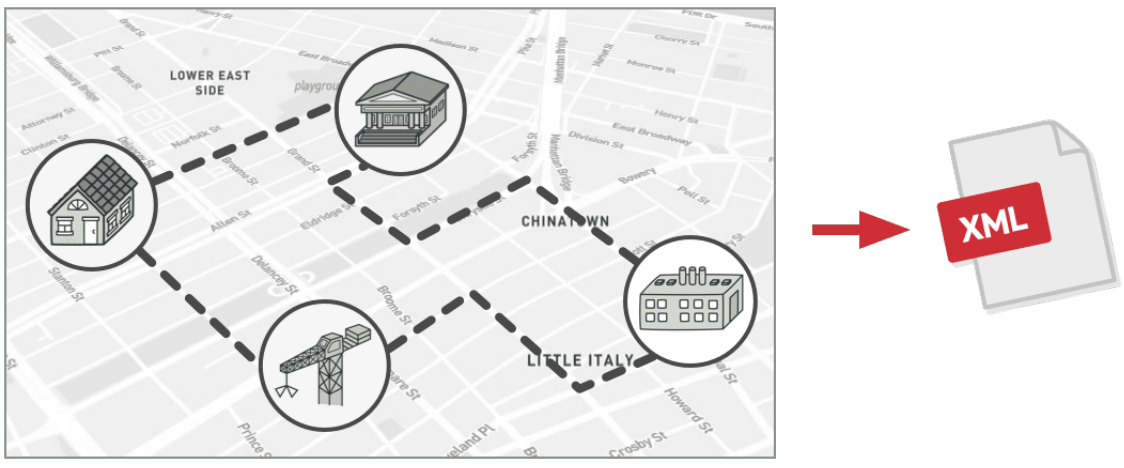
정의

- object로부터 알고리즘을 분리하는 패턴

Problem

상황

- 하나의 거대한 그래프로 구성된 지형 정보 앱을 개발하고 있습니다.
- 노드는 도시 또는 도서관같은 것이 될 수 있습니다.
- 노드 사이에 도로가 있다면 연결될 수 있습니다.
- 노드의 클래스는 노드의 유형을 나타내고, 각 특정한 노드는 객체를 나타냅니다.



요구

- 그래프를 XML format으로 export하라는 요구를 받았습니다.

안 좋은 접근

방법 - 노드에 로직 추가

- 노드에 XML exporting method를 추가합니다.
- 연결관계있는 노드들을 찾아서 recursive하게 XML로 바꿔주는 것이죠.

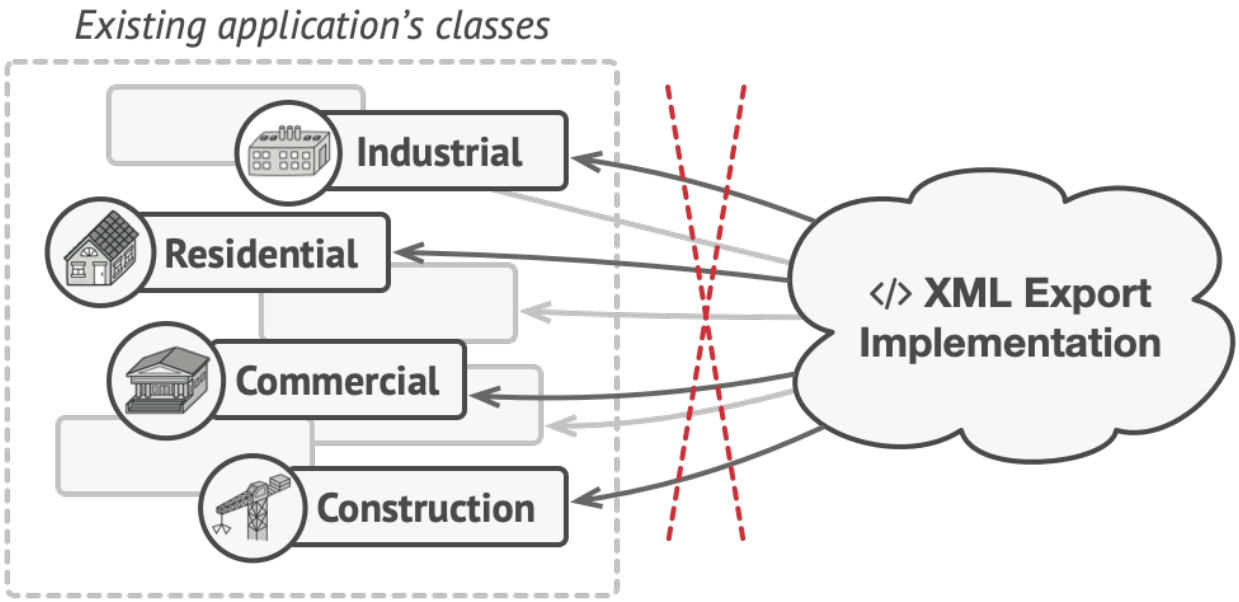
문제점

- SRP에 어긋납니다.

  노드의 주요 책임은 지리 데이터와 작업(work)하는 것인데, 거기에 XML파일을 exporting하라고 했으니까요.
- OCP에 어긋납니다.

  기능 추가로 인해 기존 로직을 변경해야하기 때문이지요. 이 때문에 예기치 않은 버그가 발생할 수 있습니다.
- 추가 기능 요구시 로직이 더더욱 지저분해집니다.

  XML뿐만 아니라 CSV파일로 Exporting하는 것도 요구한다면 또 기존 로직을 수정해야겠지요



Solution

핵심

기존 로직 변경은 최소화하면서 새로운 로직을 다른 클래스로 옮겨라!

방법

Visitor 선언

- visitor는 새로운 로직을 구현할 클래스입니다.
- 기존에 작성된 object를 visitor의 메서드의 argument로 넘겨줍니다.
    - visitor가 object에 관한 로직을 구현해야하니 해당 object를 받아야겠지요.
- 이때 object는 visitor가 필요로 하는 데이터들을 public으로 선언해줍니다.

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

메서드 호출

이 부분이 visitor패턴에서 가장 까다로운 부분입니다. 어떻게 visitor의 메서드를 호출할 지 설명드리겠습니다.

잘못 디자인한다면...

객체가 아니라 메서드이기 때문에 다형성을 적용하지 못합니다.

클라이언트에서 visitor 메서드를 실행한다면 일일이 어떤 객체인지 확인하고, 그 객체에 맞는 메서드를 실행해야하죠.

```
foreach (Node node in graph) {
    if (node instanceof City)
        exportVisitor.doForCity((City) node)
    if (node instanceof Industry)
        exportVisitor.doForIndustry((Industry) node)
    // ...
}
```

반박 - 오버로딩하면되지 않니?

=> 이름이 같아도 타입이 다르기 때문에 알 수 없습니다.

객체를 받아 로직을 수행하는 visitor는 City, Industry... 등등 그 클래스만의 method를 원하니까요. 노드들의 method가 클래스마다 상이하기 때문에 인터페이스로 선언하기 어렵습니다.

해결책 - Double Dispatch

client가 method를 호출하는 것이 아니라, visitor에게 넘길 타입의 object가 visitor method를 실행하도록 합니다. 그리고 그 메서드에 자신을 넣어주는 것이죠. 이는 객체에 accept메서드로 구현이 됩니다.

```
// Client code
foreach (Node node in 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)
    // ...
```

City 클래스를 예로 들어보겠습니다.

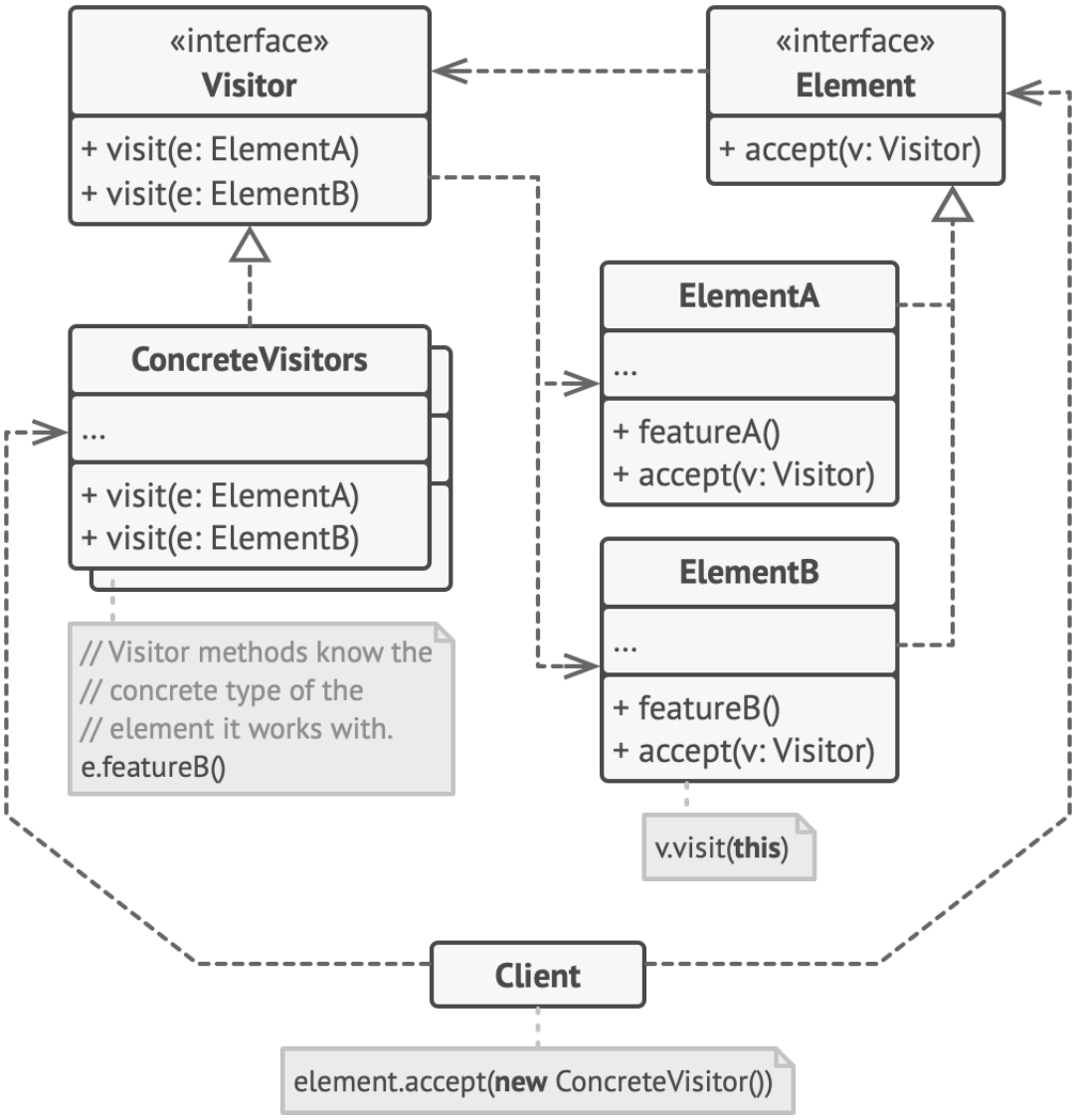
city는 visitor를 받아들이고, visitor에게 자신을 넘기면서 City에 관한 작업을 하라고 위임합니다.

visitor는 전달받은 city object를 통해서 자신의 작업을 수행합니다. city object에 필요한 데이터를 호출하면서 말이죠.

여기서 visitor라고 부른 의미가 명확해집니다.

City객체에서 accept method를 통해 방문해서 자신의 로직을 수행하는 형태가 되기 때문이죠.

구조



1. Visitor(interface)

- concrete element를 argument로 받을 수 있는 visit메서드를 선언합니다.
- 오버로딩을 지원하는 언어라면 동일한 이름으로 작성될 수 있습니다. 하지만 parameter 타입은 달라야하죠.

2. Concrete Visitor

- 서로 다른 concrete element 클래스들에 대한 몇 버전들을 구현합니다.

  위 예제에선 XML format, CSV format...등이 되겠습니다.

3. Element(interface)

- visitor를 받을 수 있는 accept 메서드를 선언합니다.

4. Concrete Element

- 반드시 visitor를 받을 수 있는 accept메서드를 구현해야 합니다.
- 이는 현재 Element class에 대해 적절한 visitor의 method로 redirect하기 위함입니다.

5. Client

- Client는 보통 collection이나 complex object(ex: composite tree)가 됩니다.
- client는 abstract interface를 통해 object와 작업하므로 concrete element를 알지 못합니다.