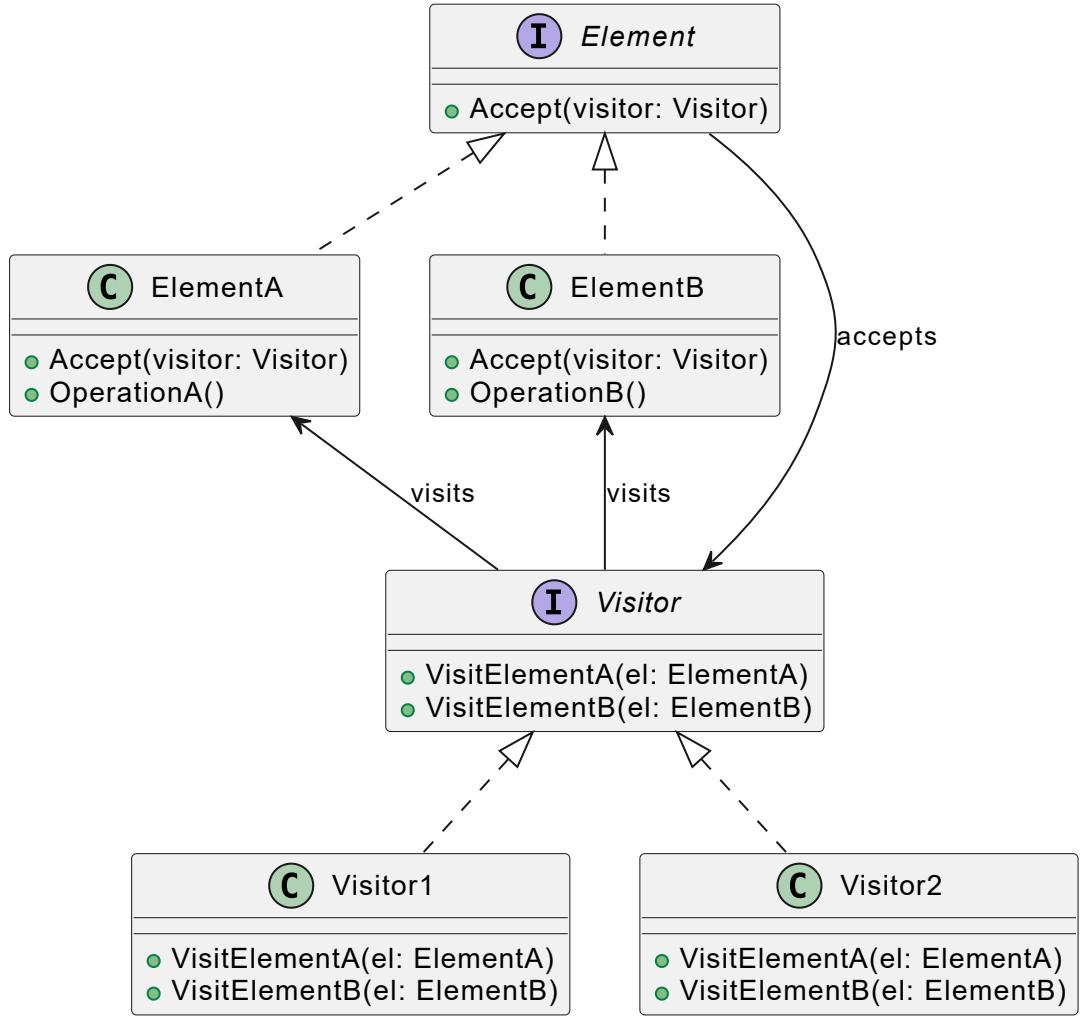
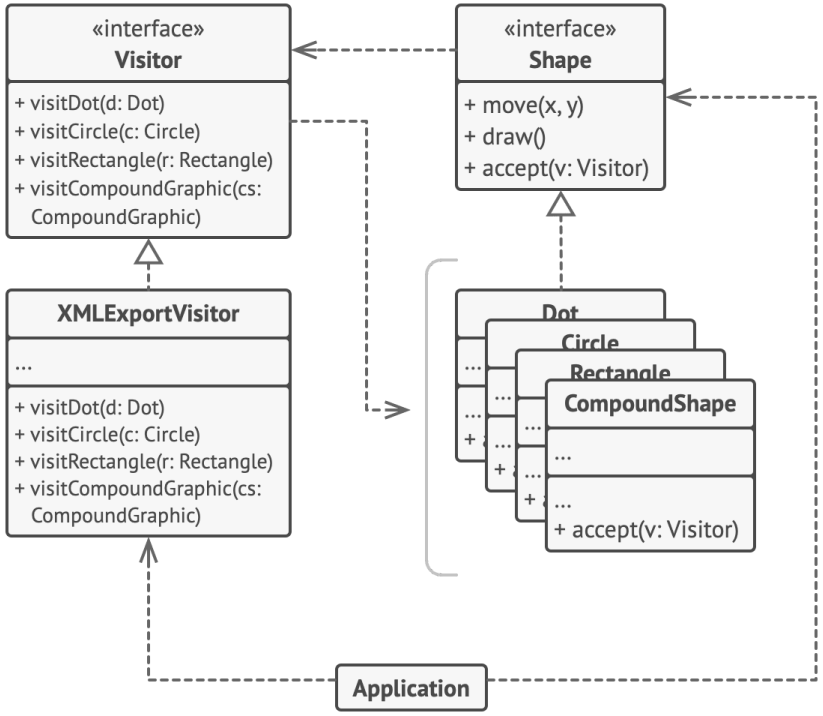
Visitor

Visitor

- lets separate algorithms from the objects on which they operate
- we are adding behaviours to objects without changing those objects - we just add a new *concrete visitor* to implement new behaviour
- (typically) connected with graphs/trees (visiting *Composite*)

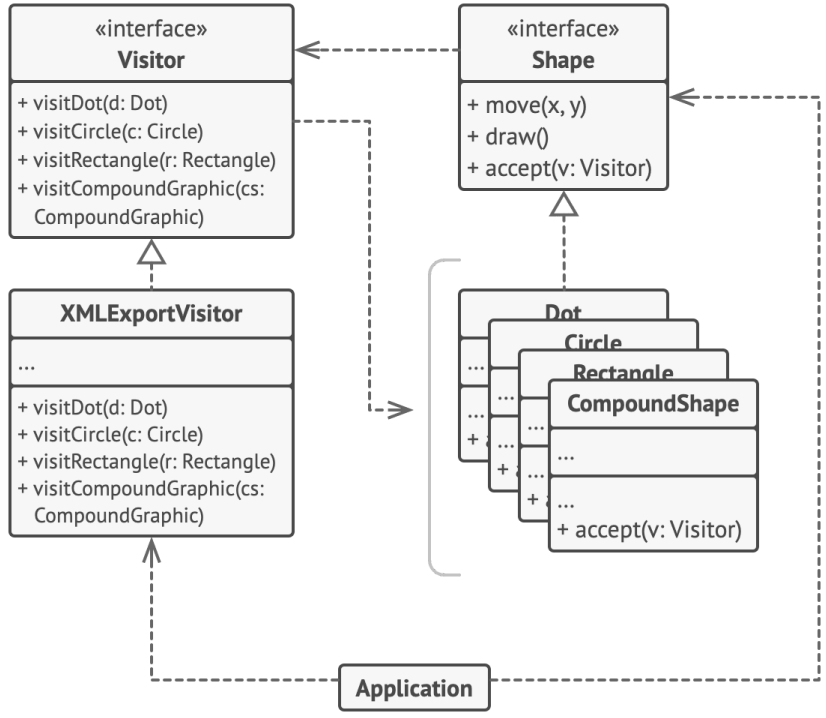


Visitor



- 👍 adding new behaviour (concrete visitor) - is easy and the main goal
- 👎 adding new concrete elements - buu
  - requires change of **`Visitor`** interface and all concrete visitors implementation

Visitor

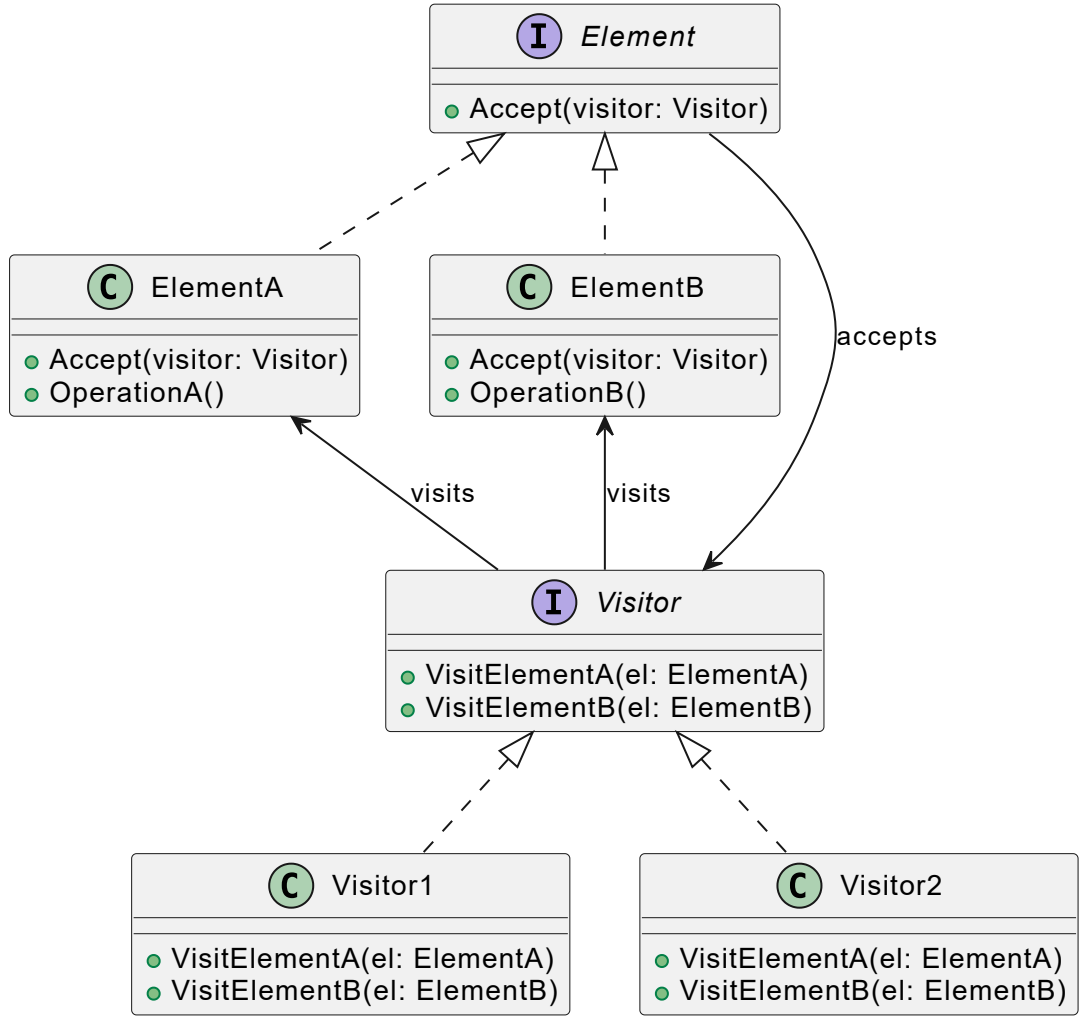


- concrete visitor may accumulate state to do the calculations
- using *Double Dispatch* technique
  - to delegate to proper types of nodes, without using *type checks* and polymorphism
  - the old saying *"We can solve any problem by introducing an extra level of indirection"* - fundamental theorem of software engineering (FTSE) 😇

Visitor

Concrete visitors are having access to the concrete elements 😍

```
Visitor1.VisitElement(el : ElementA)
{
  ...
  el.OperationA();
  ...
}
```



Visitor demo

Visitor - examples

- Generating reports - visiting a limited set of known *concrete types* (financial etc.) to generate various and changing set of reports (*visitors*)

Visitor - examples

- Generating reports - visiting a limited set of known *concrete types* (financial etc.) to generate various and changing set of reports (*visitors*)
- the "book composite" may have "an extension point" in the form of a Visitor, so we can perform various operations on it: calculating the status based on the statuses of items, calculating the total progress, etc.

Visitor - examples

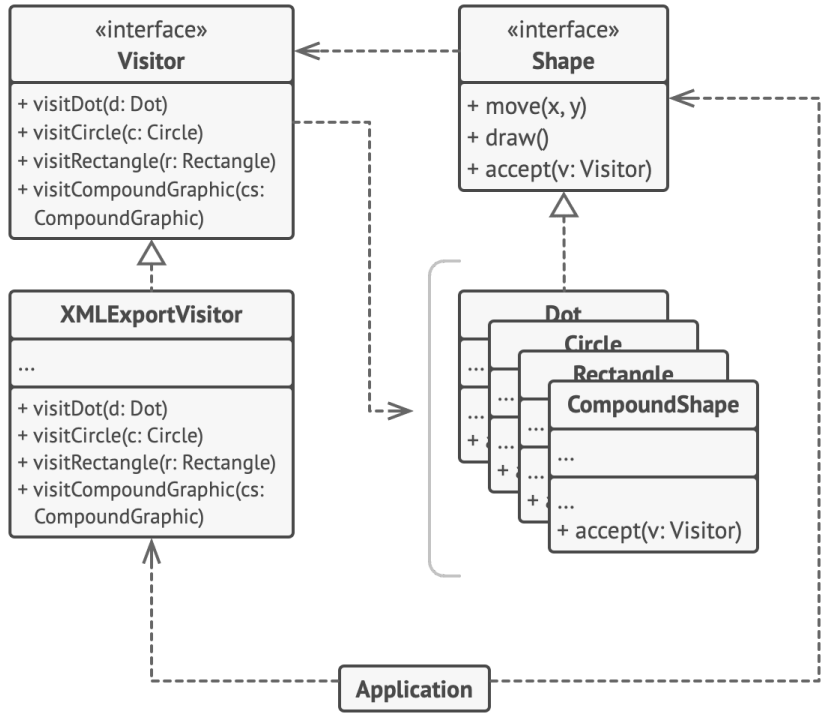
- Generating reports - visiting a limited set of known *concrete types* (financial etc.) to generate various and changing set of reports (*visitors*)
- the "book composite" may have "an extension point" in the form of a Visitor, so we can perform various operations on it: calculating the status based on the statuses of items, calculating the total progress, etc.
- *composite* representing the hierarchy of the company as "visitable" because we may want to perform a multitude of different operations on it
  - 👍good for extensions but....
  - 👎suddenly a lot of logic/algorithms are in the visitors and they need to access the visitor data - risk of making everything public for that need
  - so - is it **main** business logic we want to give add/change, or some side logic? Because the main one is unlikely to get scattered around visitors like this (I guess?).

Visitor - examples

- Generating reports - visiting a limited set of known *concrete types* (financial etc.) to generate various and changing set of reports (*visitors*)
- the "book composite" may have "an extension point" in the form of a Visitor, so we can perform various operations on it: calculating the status based on the statuses of items, calculating the total progress, etc.
- *composite* representing the hierarchy of the company as "visitable" because we may want to perform a multitude of different operations on it
  - 👍good for extensions but....
  - 👎suddenly a lot of logic/algorithms are in the visitors and they need to access the visitor data - risk of making everything public for that need
  - so - is it **main** business logic we want to give add/change, or some side logic? Because the main one is unlikely to get scattered around visitors like this (I guess?).
- Syntax walkers, for example:

```
class UsingCollector : CSharpSyntaxWalker
{
public override void VisitUsingDirective(UsingDirectiveSyntax node)
{
    WriteLine($"\tVisitUsingDirective called with {node.Name}.");
    ...
}
}
```

Visitor



😍 use the Visitor to **clean up the business logic of auxiliary behaviors** - the pattern lets you make the primary classes of your app more focused on their main jobs by **extracting** all other behaviors into a set of visitor classes