4-Visitor

terça-feira, 14 de janeiro de 2025 20:37

Protected constructor within a class
↳ Associated with a Base class → Enforce control on instantiation
↳ Can only be accessed by → Derived classes
→ Class itself
↳ Helps control obj creations with Singletons or Factory

- **Visitor**
  - Add new functionalities to objects without adding new structure
  - Usefull when the object structure is stable but frequently need to change its behavior with new operators
  - Separates algorithms from the objects they operate on. Allowing you to add new operations without modifying the object structure
  - Intent to extend operations instead of types
  - Motivation
    - Both options on how to sove a relation between class and base class.... needs to choose the bette approach used here
      - Do not ignore the weaknesses, and do not put yourself in an unfortunate maintenance hell.

Table 4-1. Strengths and weaknesses of different programming paradigms

Programming paradigm	Strength	Weakness
Procedural programming	Addition of operations	Addition of (polymorphic) types
Object-oriented programming	Addition of (polymorphic) types	Addition of operations

  - The visitor is a OO answer for this limitation
    - Focus on allowing you to frequently add oerations instead of types

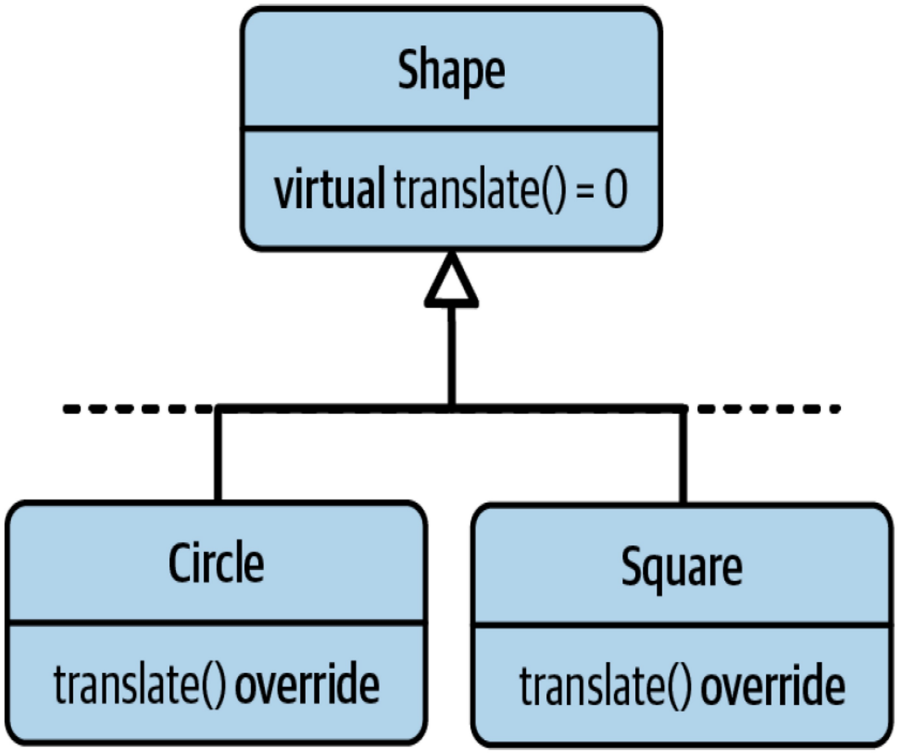
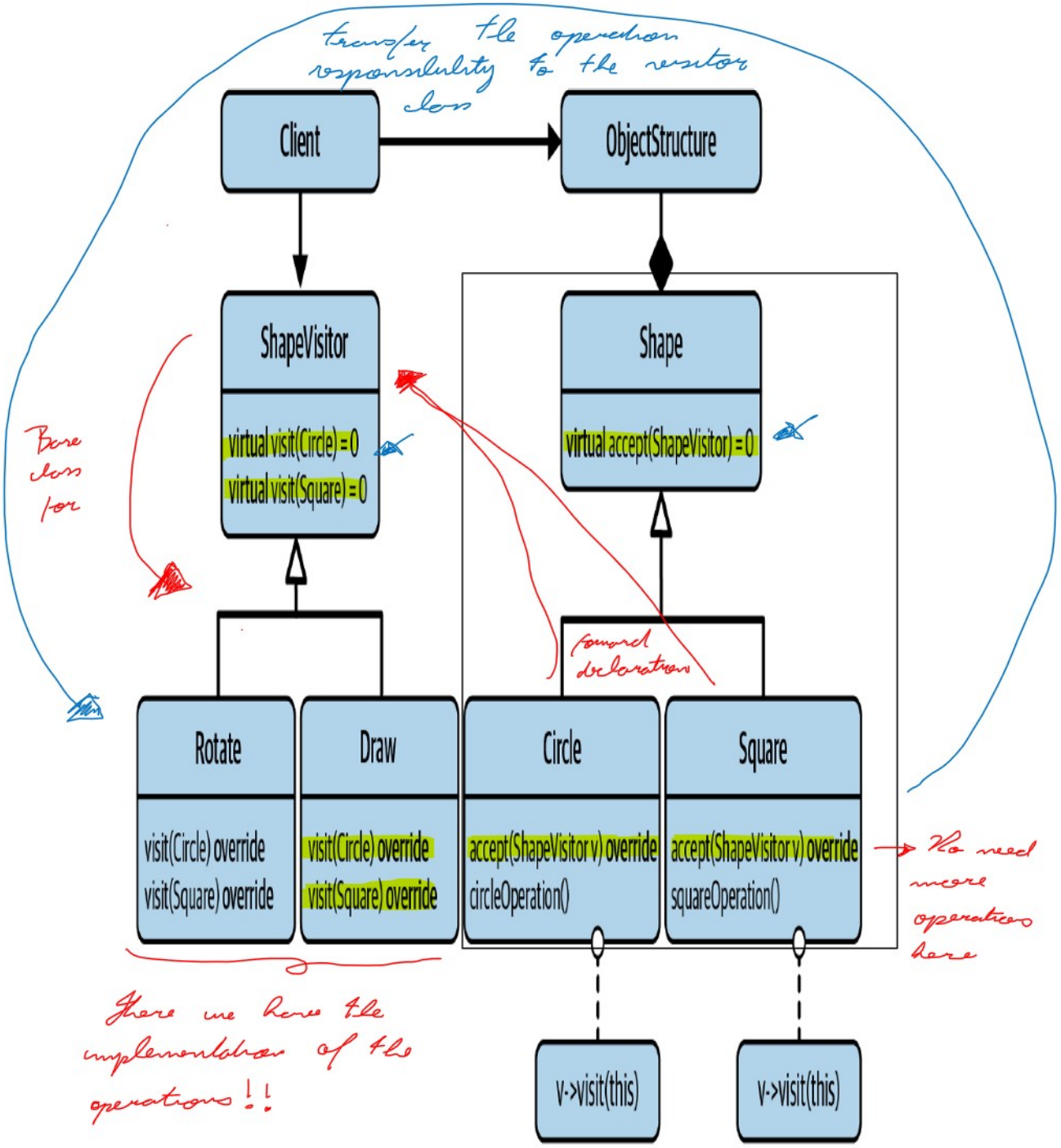


Figure 4-1. The UML representation of a shape hierarchy with two derived classes (`Circle` and `Square`)

- Once we define all types, but now we need to grow the operations on those types
  - If selected the OO approach, every new operation requires changes on the base and all derived classes (too much)
    - if used a pure virtual function
    - Using regular virtual functions, with base definitions

- Not easy to cover all the cases
- if exception, it means that the derived class must implement it too (same work)

- The visitor enables the opton to add new operations easily
- Intent: Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates



- The ShapeVisitor Base class represents an abstraction of shape operations
  - Comes with one pure virtual visit() function for every concrete shape in shape hierarchy
    - One for Circle
    - One for Square
    - ...
- With this class in place, it's possible to add new operations easily
- All you need to do is to add a new derived class
  - Derived from ShapeVisitor
  - To enable Drawiong shapes for example
    - Create a new Draw class, deriving from ShapeVisitor
    - override the pure virtual function for the draw class
    - Implement the visit function in the actuall .cpp file
      - respecting the parameter type. That will allow you to define several visit functions but with different parameters types. One for each type
      - Creates several of overloads to the same visit() function
- There is no need to change the Shape base class every time we need to add a new operation now

```
class ShapeVisitor
{
 public:
   virtual ~ShapeVisitor() = default;

   virtual void visit( Circle const&, /*...*/ ) const = 0;  ❶
   virtual void visit( Square const&, /*...*/ ) const = 0;  ❷
   // Possibly more visit() functions, one for each concrete shape
};
```

```
class Draw : public ShapeVisitor
{
 public:
   void visit( Circle const& c, /*...*/ ) const override;
   void visit( Square const& s, /*...*/ ) const override;
   // Possibly more visit() functions, one for each concrete shape
};
```

- Ok, now to use the visitor (classes derived from ShapeVisitor base class) on shapes
  - Need to add one last function
    - accept()
      - Introduced as pure virtual function in the base class
      - And has to be implemented in every derived class
  - Shape base class cannot provide a base default implementation

```
class Shape
{
 public:
   virtual ~Shape() = default;
   virtual void accept( ShapeVisitor const& v ) = 0;  ❸
   //
```

- Need to add one last function
  - accept()
    - Introduced as pure virtual function in the base class
    - And has to be implemented in every derived class
  - Shape base class cannot provide a base default implementation

```
{
 public:
   virtual ~Shape() = default;
   virtual void accept( ShapeVisitor const& v ) = 0;  ❸
   // ...
};
```

- The implementation is easy. Refers to the ShapeVisitor base visit class
  - merely needs to call the corresponding visit() function (from class Draw : public ShapeVisitor)
    - Achieved by passing the (this) pointer as an argument to visit()
  - The implementation of accept() is the same for every derive class
    - Due to a diferent type of the pointer, it'll trigger a different overload of the visit() function

```
class Square : public Shape
{
 public:
   explicit Square( double side )
      : side_( side )
   {
      /* Checking that the given side length is valid */
   }

   void accept( ShapeVisitor const& v ) override { v.visit( *this ); }  ❺

   double side() const { return side_; }

 private:
   double side_;
};
```

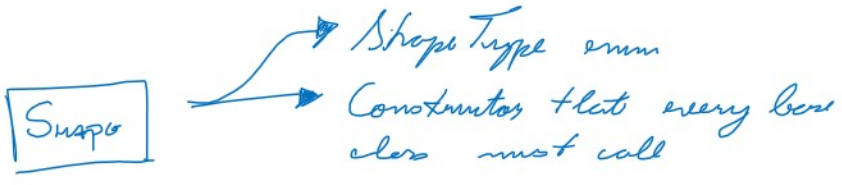
- The accept function can now be used where you need to perform an operation
  - In this casse, it's calling the Draw operation (implemented in the Draw class but called visit)
  - So it call the accept method from base class shape
    - Which will trgger the corresponding derived class from shape
      - Which wil trigger the associated accept method
        - Which will trigger the associated visit overload method from Draw class
          - Which is an pure virtual method overload defined in ShapeVisitor base class
- Woth this in place, it's now easy to add new operations to the shape hierarchy

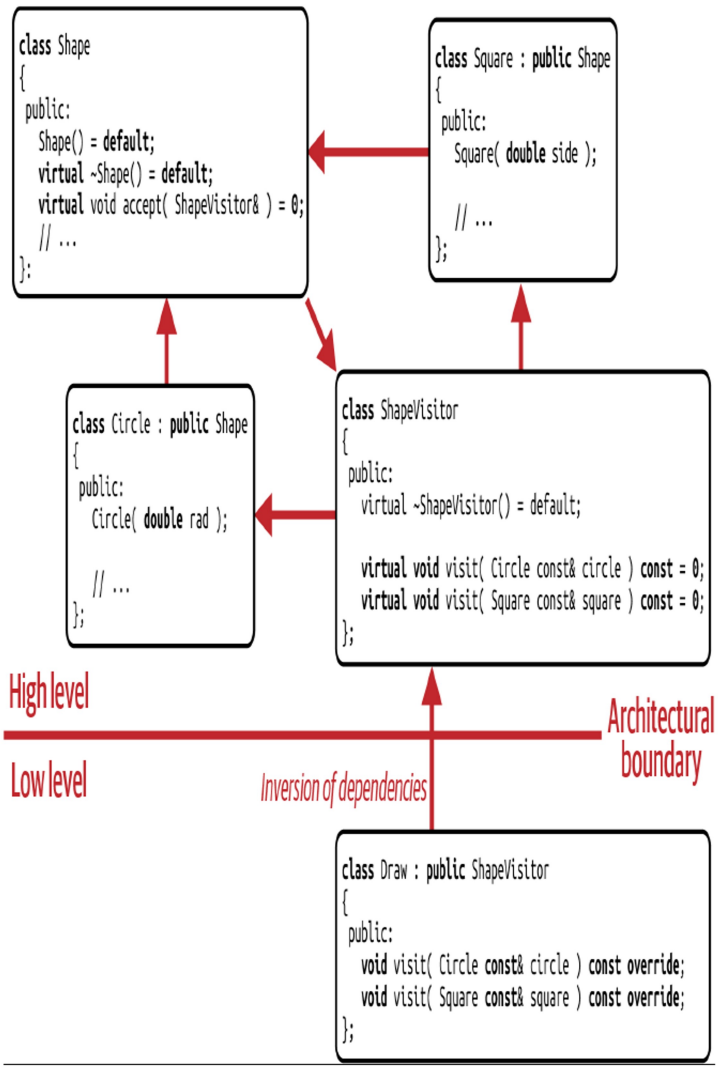
```
void drawAllShapes( std::vector<std::unique_ptr<Shape>> const& shapes )
{
   for( auto const& shape : shapes )
   {
      shape->accept( Draw{} );
   }
}
```

Down sides

- Visitor is a work arround for a OO weakness
- Low implementation flexibility
  - Needs to implemenr a visit() function for every concrete Shape
    - Even if the implementation is identical
      - But we can use templates....
- Return tye of visitor
  - Decision is made in the visitor base class (ShapeVisitor)
    - Approach to store in the visitor and access it later
- Becomes dificult to add new types
  - Add a new shape, requires changes in all the visitor structure
    - From the base to all derived operations
  - It becomes a disavantage
  - Visitor requires a cosed set of types and provides an open set of operations
- There s a cyclic dependency mong the shapeVisitor base class, the concrete shapes and the shae base class
  - Draw -> ShapeVisitor -> Square / Circle -> Shape -> ShapeVisitor
- Intrusive nature of a visitor
  - To add a visitor needs to add a pure virtual function in the base class
    - It requires to change everybody
  - There is another nonintrusive form of the visitor design pattern
    - std::variant
- Accept() is inheritedby any other layer of derived class
  - If create a new layer with Square as base class... it must implement accept as well
  - and requires to update every derived classe from the visitor base class (like adding a new type)
    - can declare the classes as final (to prevent this)
- For each operation ,we now have to call 2 virtual functions
  - acceot()
    - which resolves the concrete type
  - visit()
    - which has the atual implementation
  - Should consider visitor as a slow patter wow
- Cause memory fragmentation by allocating meny small types in a vector
  - That is why we usually use ointers to work with the resulting shaes and visitors
  - Making it hard to peform optimizations
- hard to understand and maintain!!

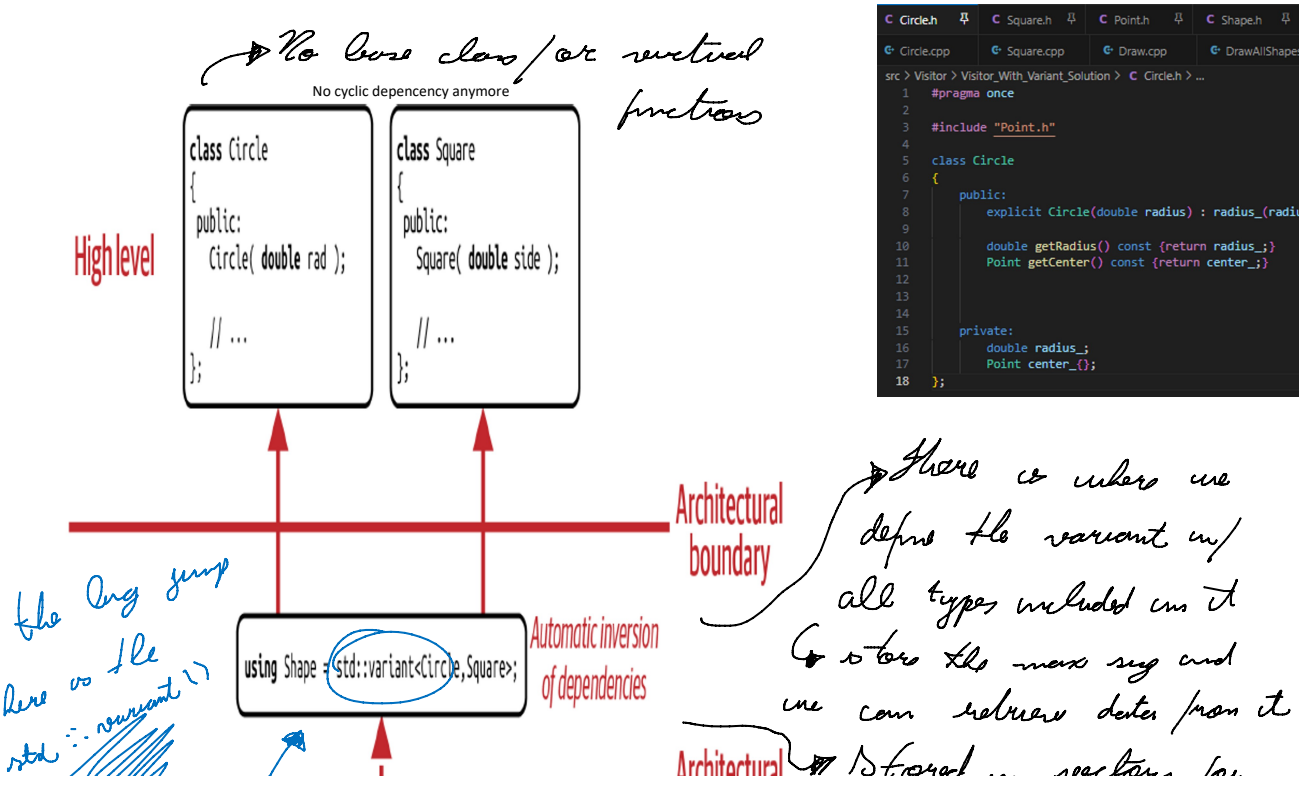
Procedural solution (IMPLEMENTATION):





Cyclc dependency

Guideline 17: Consider std::variant for Implementing Visitor



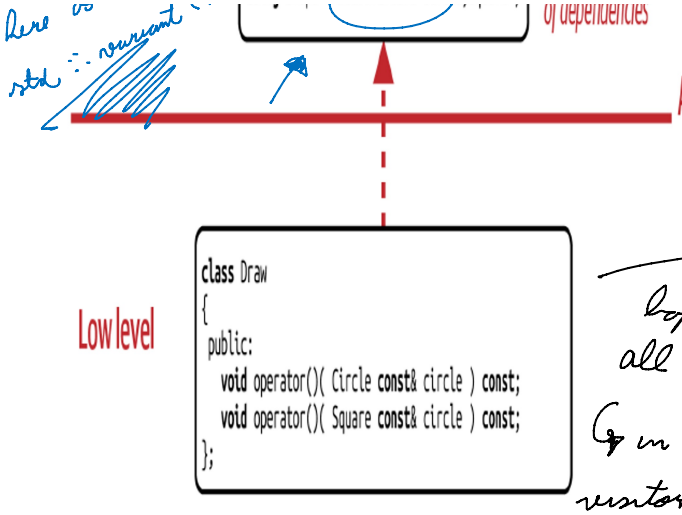


Table 4-2. Benchmark results for different Visitor implementations

Visitor implementation	GCC 11.1	Clang 11.1
Classic Visitor design pattern	1.6161 s	1.8015 s
Object-oriented solution	1.5205 s	1.1480 s
Enum solution	1.2179 s	1.1200 s
`std::variant` (with `std::visit()`)	1.1992 s	1.2279 s
`std::variant` (with `std::get_if()`)	1.0252 s	0.6998 s

-> The most obvious takeaway from the benchmark results is that the variant solution is far more efficient than the classic Visitor solution.

-> The most obvious takeaway from the benchmark results is that the variant solution is far more efficient than the classic Visitor solution. This should not come as a surprise: due to the double dispatch, the classic Visitor implementation contains a lot of indirection and therefore is also hard to optimize.

To ADD new TYPES
- First of all, you would have to update the variant itself, which might trigger a recompilation of all code using the variant type
- you would have to update all operations and add the potentially missing for the new operator() alternative(s). (The compiler will complain if one is missing