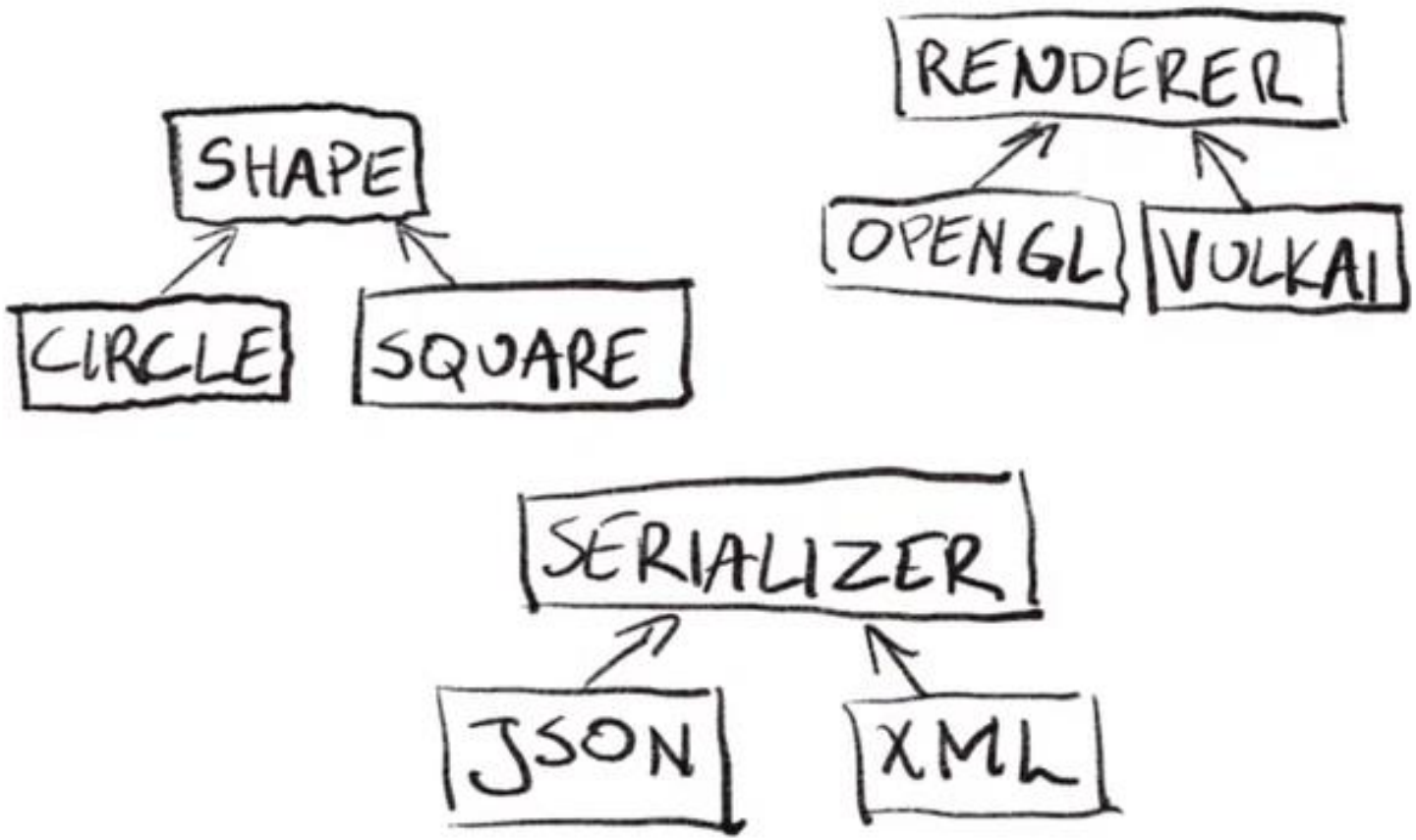
Visitor Pattern

Hien D. Nguyen

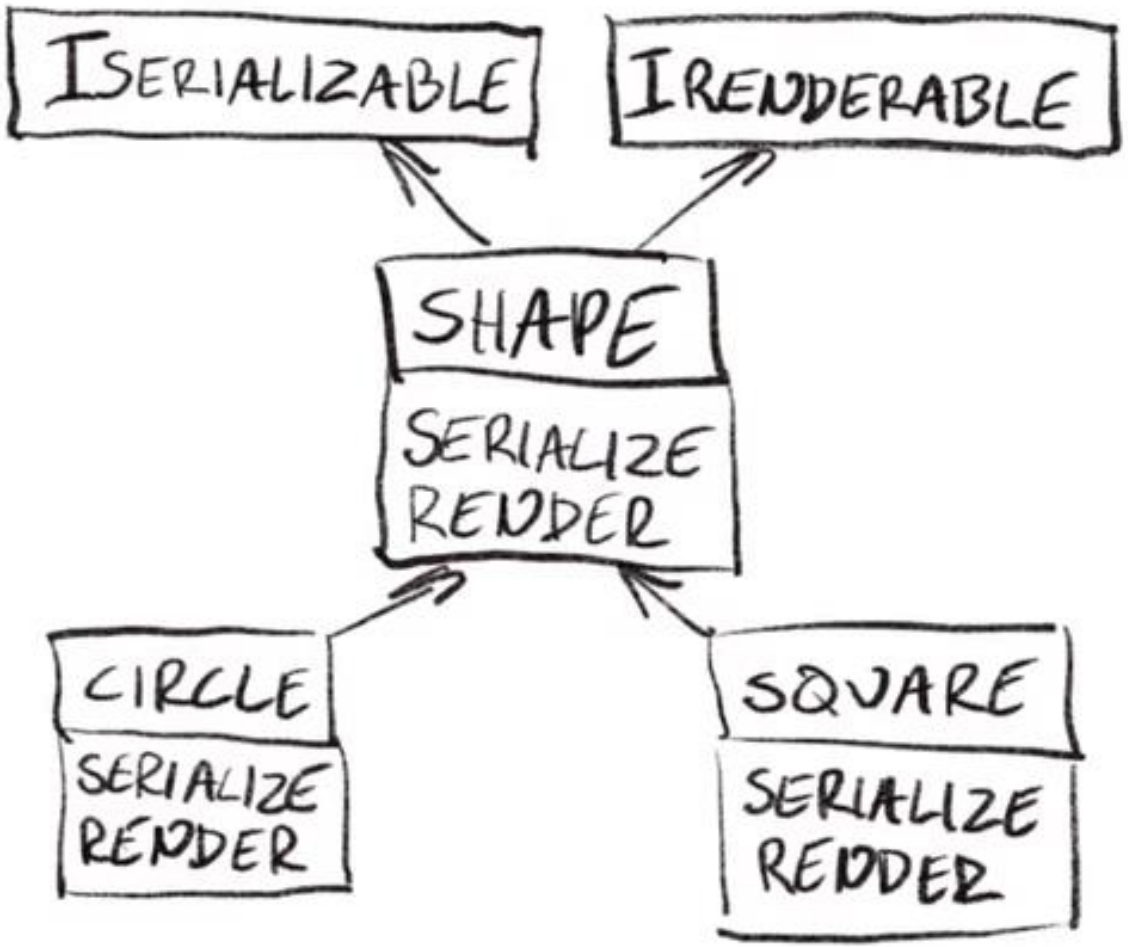
Motivation

- Break dependency between hierarchies of objects and operations performed on them.
- Ability to add new operations to existing hierarchies without needing to modify them.
  - Effectively adding new virtual functions without changing interfaces.
  - This follows the Open-Closed Principle.
- Perform Double-Dispatch in C++ (special case of Multiple-Dispatch).

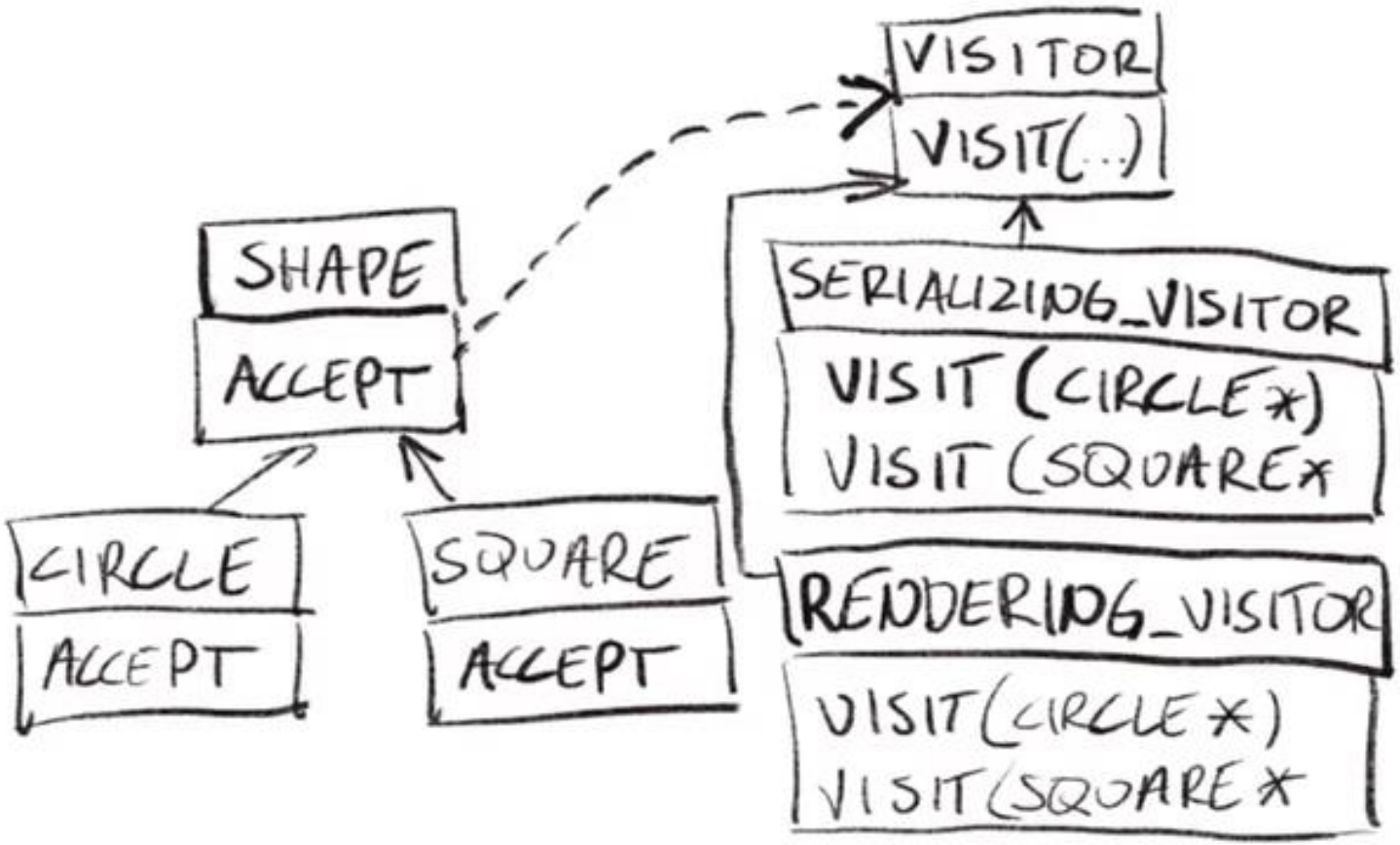
Hierarchies of Classes and Operations



Bad Solution



Visitor Solution



The Interfaces

```
class Shape
{
public:
    virtual ~Shape() = default;
    virtual float Size() const = 0;
    virtual float Area() const = 0;
    virtual void Accept(ShapeVisitor* v) = 0;
};
```

```
class ShapeVisitor
{
public:
    virtual ~ShapeVisitor() = default;
    virtual void Visit(Circle*) = 0;
    virtual void Visit(Square*) = 0;
};
```

- **Shape** and its derived classes know only about **ShapeVisitor** interface.
- **ShapeVisitor** and its children know about every **Shape** and how to *operate* on it.

The Implementation

```
class Circle : public Shape
{
public:
    explicit Circle(float r) : radius(r) {}
    virtual float Size() const override { return radius; }
    virtual float Area() const override { return 3.14159f * radius * radius; }
    virtual void Accept(ShapeVisitor* v) override { v->Visit(this); }
private:
    float radius;
};

class Square : public Shape
{
public:
    explicit Square(float s) : side(s) {}
    virtual float Size() const override { return side; }
    virtual float Area() const override { return side * side; }
    virtual void Accept(ShapeVisitor* v) override { v->Visit(this); }
private:
    float side;
};
```

```
class SerializeShapeVisitor : public ShapeVisitor
{
public:
    virtual void Visit(Circle* c) override { cout << "Serializing Circle ... " << endl
    virtual void Visit(Square* s) override { cout << "Serializing Suqare ... " << endl
};

class RenderShapeVisitor : public ShapeVisitor
{
public:
    virtual void Visit(Circle* c) override { cout << "Rendering Circle ... " << endl;
    virtual void Visit(Square* s) override { cout << "Rendering Suqare ... " << endl;
};
```

The Application

```
Shape* c = new Circle(1.5f);
Shape* s = new Square(2.2f);

ShapeVisitor* sv = new SerializeShapeVisitor;
c->Accept(sv);
s->Accept(sv);

ShapeVisitor* rv = new RenderShapeVisitor;
c->Accept(rv);
s->Accept(rv);
```

- No coupling between **Shape** and **ShapeVisitor** hierarchies.
- **Shape** knows only about one abstract interface and how to interact with it.

Multiple-Dispatch

```
Shape* c = new Circle(1.5f);
Shape* s = new Square(2.2f);

ShapeVisitor* sv = new SerializeShapeVisitor;
ShapeVisitor* rv = new RenderShapeVisitor;

sv->Visit(c);
rv->Visit(s);
```

- Selects the method based on dynamic types of both the object and parameter(s).
- sv will resolve to SerializeShapeVisitor but neither c nor s will resolve to corresponding dynamic type.
- Inside instance of c and s the type is knows, hence the need for double virtual function call.

References

https://github.com/mvorbrodt/blog/blob/master/src/visitor.cpp