# NODE Technical Book Club

## C++ Software Design - Klaus Iglberger

# G11: Understand the Purpose of Design Patterns

A design pattern:

- Has a name.
- Carries an intent.
- Introduces an abstraction.
- Has been proven.

## Example conversation:

- **ME:** I would use a Visitor for that.
- **YOU:** I don’t know. I thought of using a Strategy.
- **ME:** Yes, you may have a point there. But since we’ll have to extend operations fairly often, we probably should consider a Decorator as well.

# G12: Beware of Design Pattern Misconceptions

Design patterns are **not**:

- A goal
- About implementation details
- Limited to object-oriented programming
- Outdated or obsolete

# G13: Design Patterns are Everywhere

- Response to the misconception that design patterns are outdated or obsolete.
- They are everywhere, any kind of abstraction and any attempt to decouple likely represents a design pattern.
- Example: STL is full of design patterns.

# G14: Use a Design Pattern's Name to Communicate Intent

From:

```
template< class InputIt, class T, class BinaryOperation >
constexpr T accumulate( InputIt first, InputIt last, T init,
BinaryOperation op );
```

To:

```
template< class InputIt, class T, class BinaryReductionStrategy >
constexpr T accumulate( InputIt first, InputIt last, T init,
BinaryReductionStrategy op );
```

# G15: Design For the Addition of Types or Operations

- Design choice in dynamic polymorphism: Do you want to extend the ypes or the operations?

# A Procedural Solution

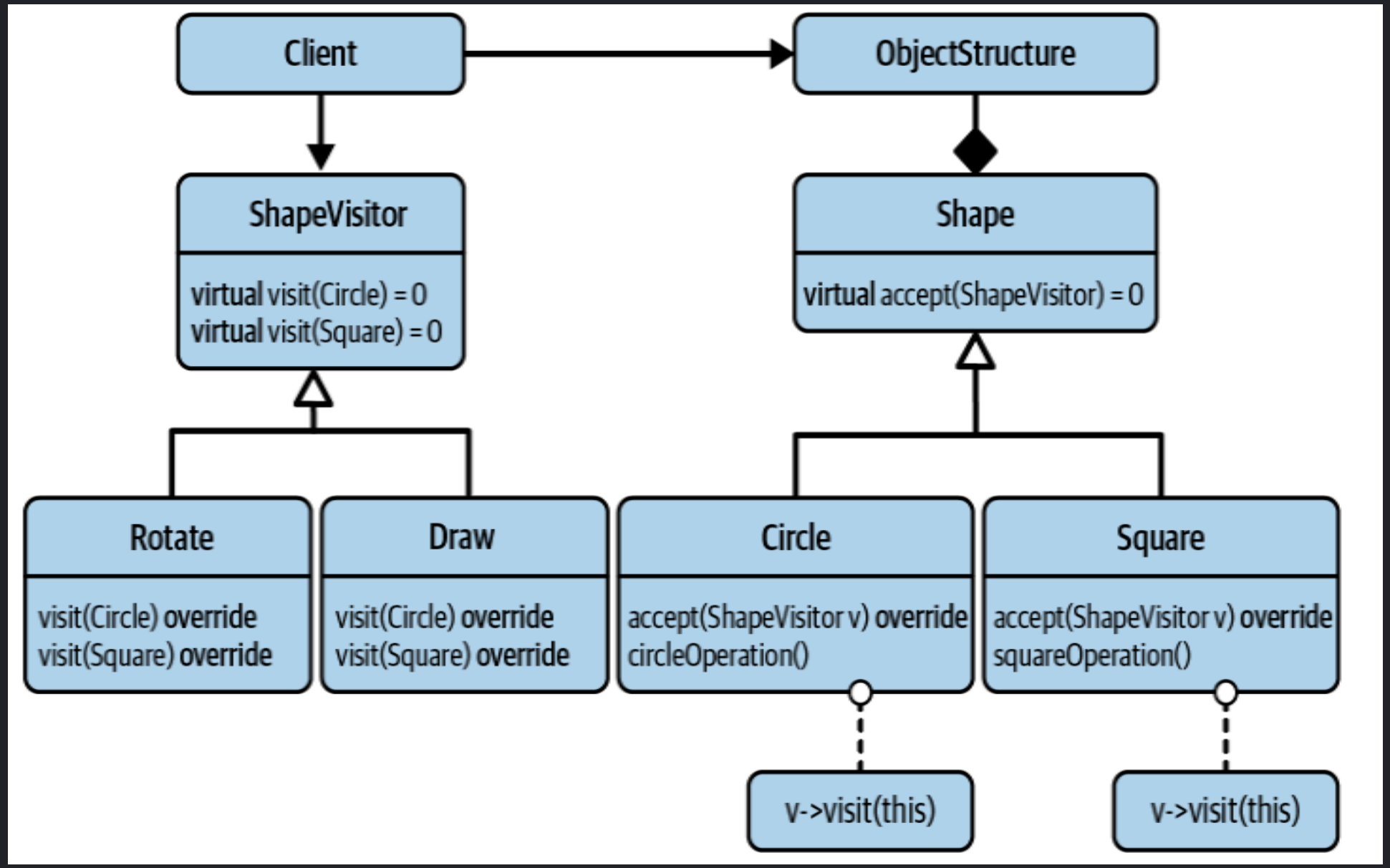
```
void drawAllShapes(std::vector<std::unique_ptr<Shape>> const& shapes)
{
        for(auto const& shape: shapes)
        {
                switch(shape->getType())
                {
                        case circle:
                                draw(static_cast<Circle const&>(*shape));
                                break;
                        case square:
                                draw(static_cast<Square const&>(*shape));
                                break;
                }
        }
}
```

# An Object-Oriented Solution

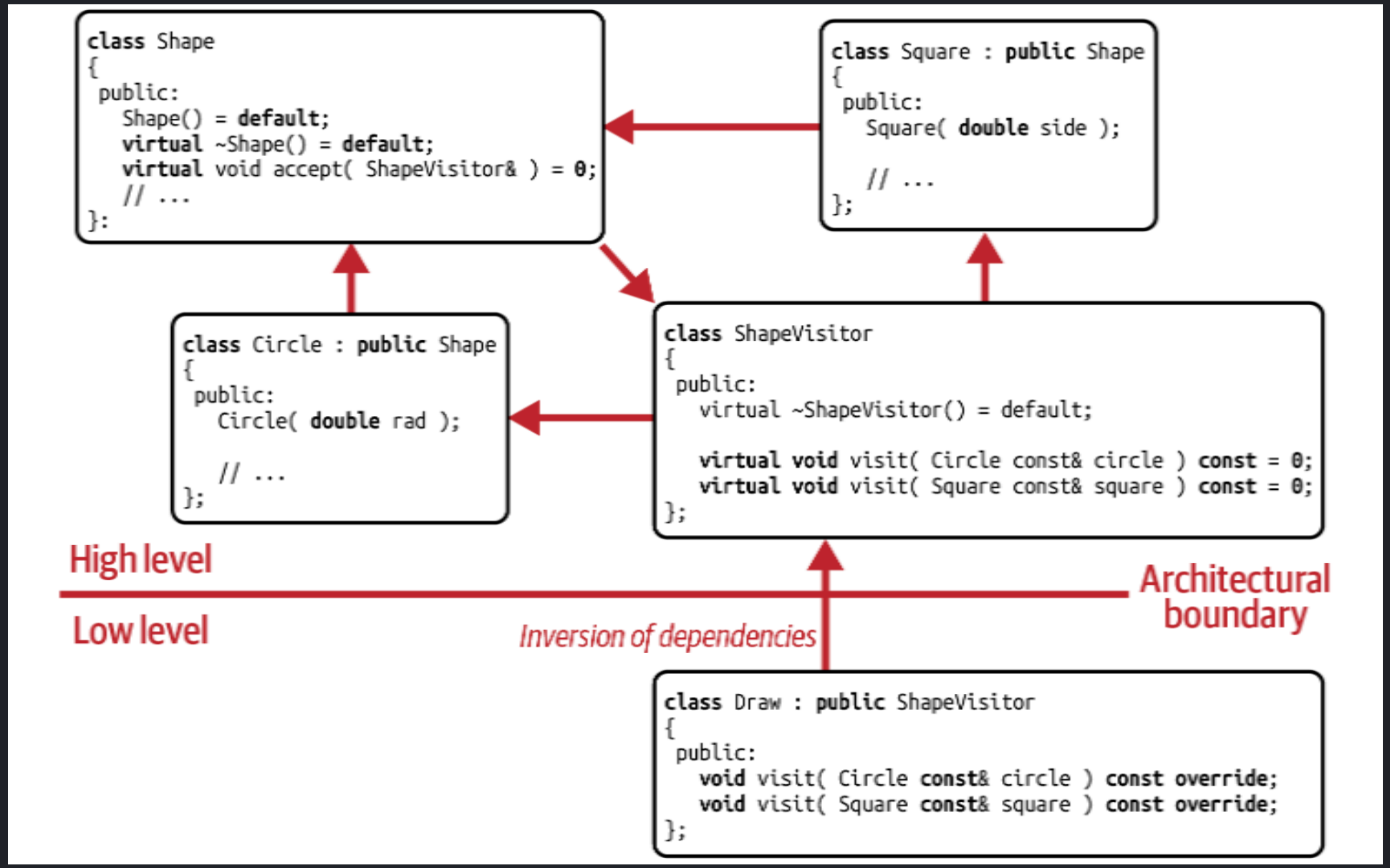
```
void drawAllShapes(std::vector<std::unique_ptr<Shape>> const& shapes)
{
        for(auto const& shape: shapes)
        {
                shape->draw();
        }
}
```

# G16: Use Visitor to Extend Operations

- In OOP solution, every new operation requires adding a new virtual function to the base class. But:
  - It is not always possible.
  - Need to know how to implement it for all shapes.
  So if you want to extend operations, use the **Visitor** design pattern.


Client
ObjectStructure
ShapeVisitor
virtual visit(Circle) = 0
virtual visit(Square) = 0
Shape
virtual accept(ShapeVisitor) = 0
Rotate
visit(Circle) override
visit(Square) override
Draw
visit(Circle) override
visit(Square) override
Circle
accept(ShapeVisitor v) override
circleOperation()
Square
accept(ShapeVisitor v) override
squareOperation()
v->visit(this)
v->visit(this)

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


class Shape
{
public:
Shape() = default;
virtual ~Shape() = default;
virtual void accept( ShapeVisitor& ) = 0;
// ...
};
class Square : public Shape
{
public:
Square( double side );
// ...
};
class Circle : public Shape
{
public:
Circle( double rad );
// ...
};
class ShapeVisitor
{
public:
virtual ~ShapeVisitor() = default;
virtual void visit( Circle const& circle ) const = 0;
virtual void visit( Square const& square ) const = 0;
};
High level
Low level
Inversion of dependencies
Architectural boundary
class Draw : public ShapeVisitor
{
public:
void visit( Circle const& circle ) const override;
void visit( Square const& square ) const override;
};

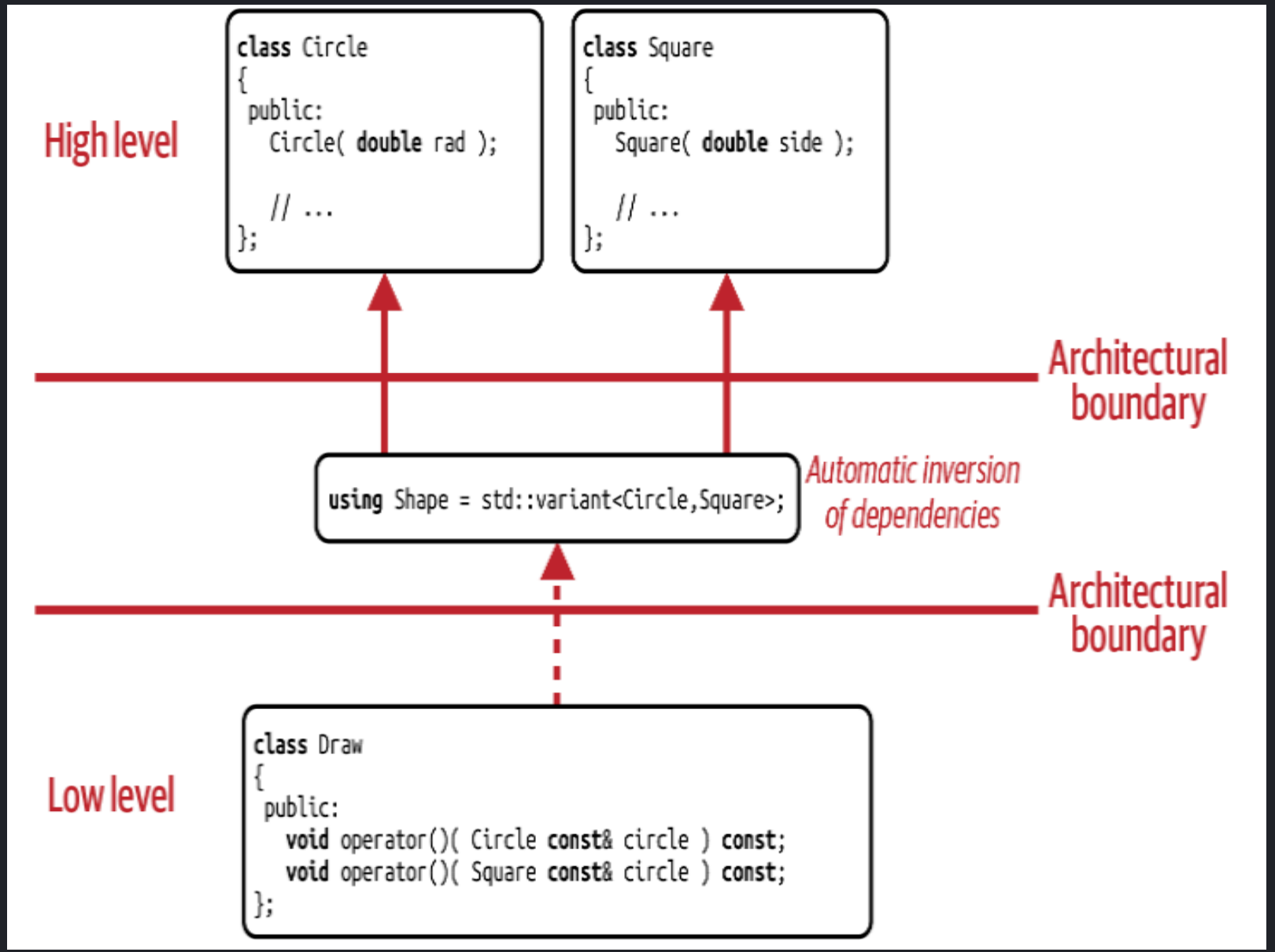
# Shortcomings of the Visitor Pattern

- Low implementation flexibility
- Intrusive nature
- Low performance

# G17: Consider std::variant for Implementing Visitor

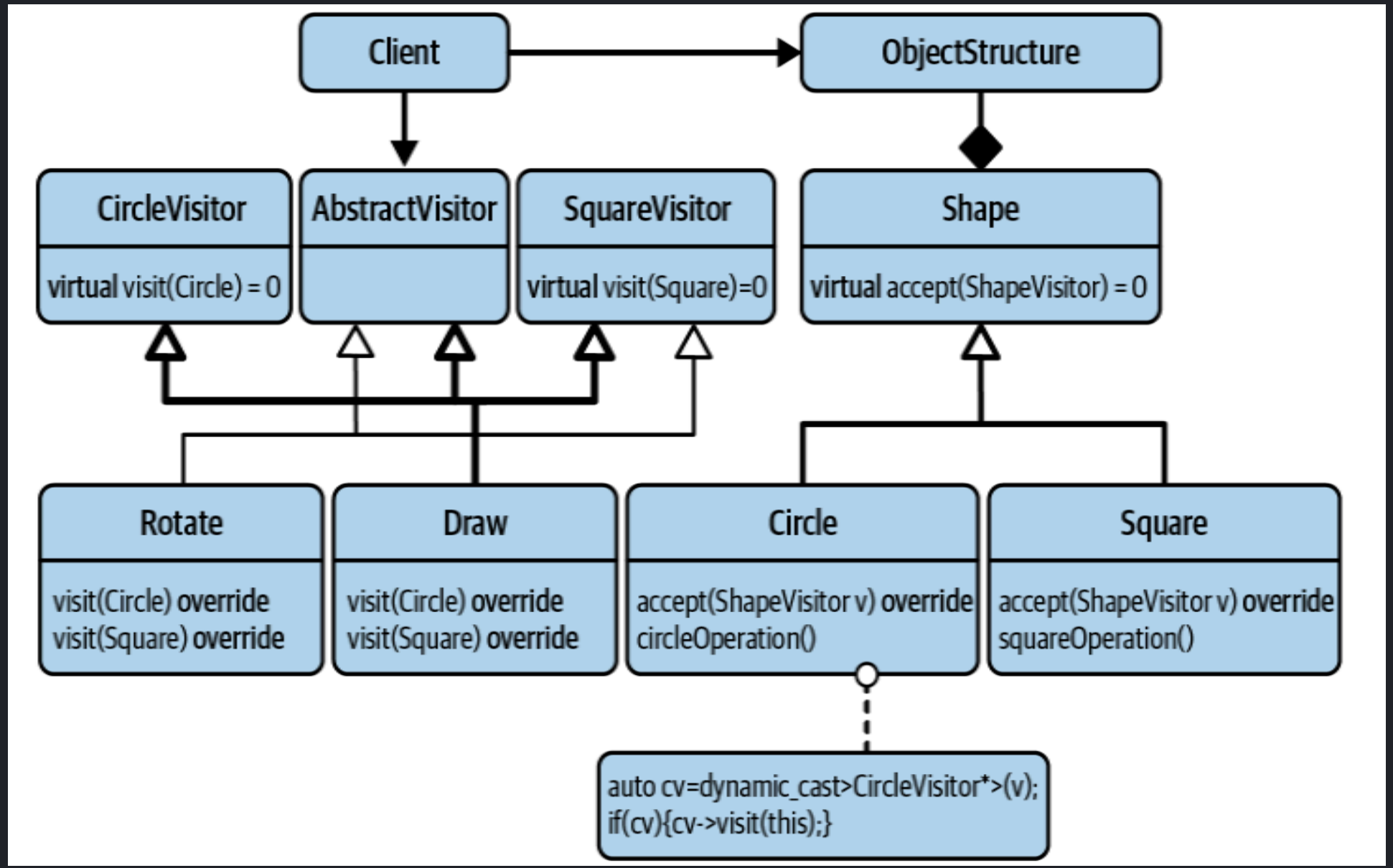
```
using Shape = std::variant<Circle, Square>;

struct Draw
{
        void operator()(Circle const& circle) const;
        void operator()(Square const& square) const;
};
```

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


High level
class Circle
{
public:
Circle( double rad );
// ...
};
class Square
{
public:
Square( double side );
// ...
};
Architectural boundary
using Shape = std::variant<Circle,Square>;
Automatic inversion of dependencies
Architectural boundary
Low level
class Draw
{
public:
void operator()( Circle const& circle ) const;
void operator()( Square const& circle ) const;
};

# G18: Beware the Performance of Acyclic Visitor

- Technically it is possible to support both an open set of types and an open set of operations using the **Acyclic Visitor**, but it is impractical.


Client
ObjectStructure
CircleVisitor
virtual visit(Circle) = 0
AbstractVisitor
SquareVisitor
virtual visit(Square)=0
Shape
virtual accept(ShapeVisitor) = 0
Rotate
visit(Circle) override
visit(Square) override
Draw
visit(Circle) override
visit(Square) override
Circle
accept(ShapeVisitor v) override
circleOperation()
Square
accept(ShapeVisitor v) override
squareOperation()
auto cv=dynamic_cast>CircleVisitor*>(v);
if(cv){cv->visit(this);}

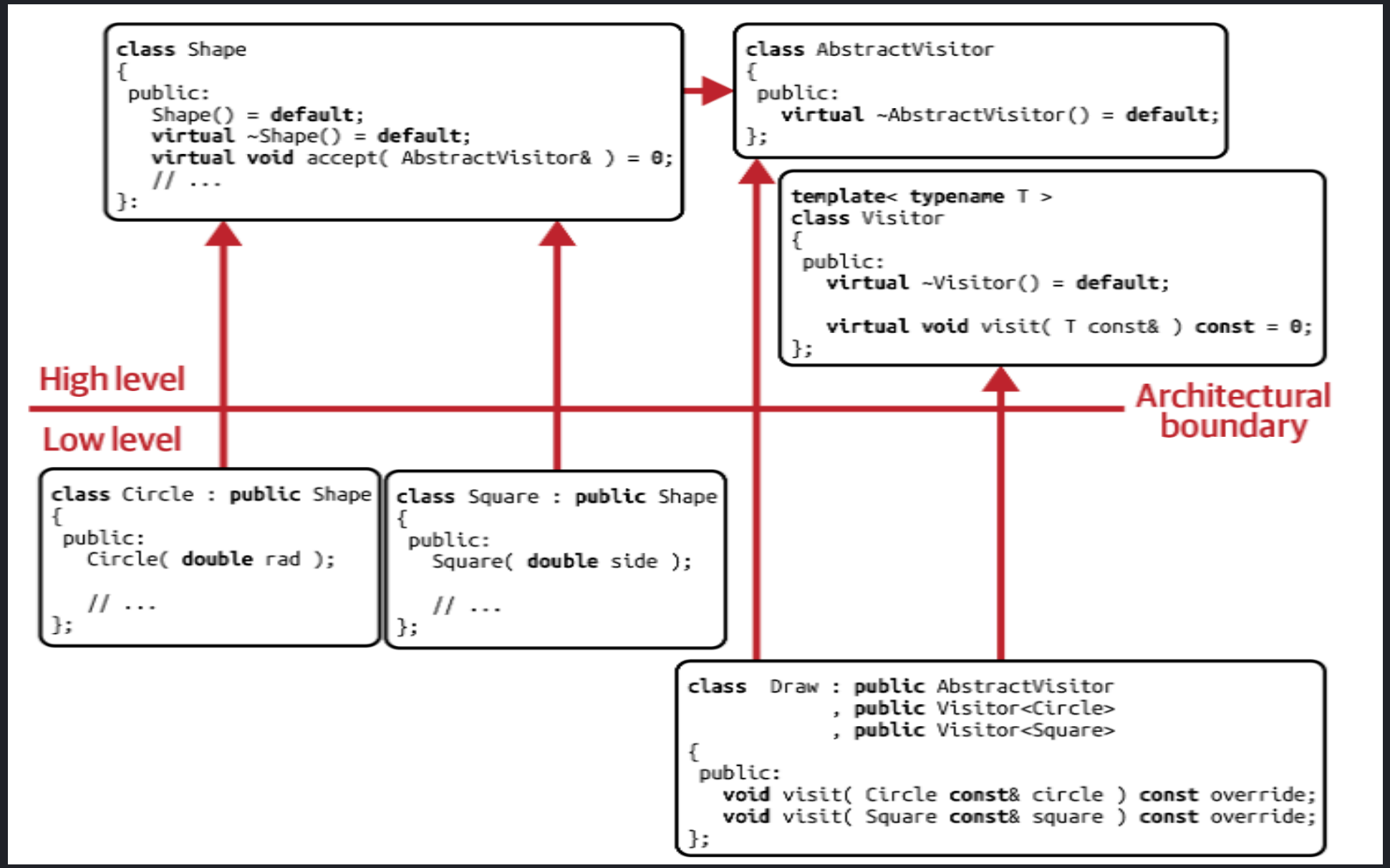

class Shape
{
public:
Shape() = default;
virtual ~Shape() = default;
virtual void accept( AbstractVisitor& ) = 0;
// ...
};
class AbstractVisitor
{
public:
virtual ~AbstractVisitor() = default;
};
template< typename T >
class Visitor
{
public:
virtual ~Visitor() = default;
virtual void visit( T const& ) const = 0;
};
High level
Low level
Architectural boundary
class Circle : public Shape
{
public:
Circle( double rad );
// ...
};
class Square : public Shape
{
public:
Square( double side );
// ...
};
class Draw : public AbstractVisitor
, public Visitor<Circle>
, public Visitor<Square>
{
public:
void visit( Circle const& circle ) const override;
void visit( Square const& square ) const override;
};

*Table 4-3. Performance results for different Visitor implementations*

| Visitor implementation | GCC 11.1 | Clang 11.1 |
|---|---|---|
| Acyclic Visitor | 14.3423 s | 7.3445 s |
| Cyclic Visitor | 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()`) | 1.0252 s | 0.6998 s |

# G19: Use Strategy to Isolate How Things are Done

Intent of the Strategy pattern: Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

# Example: Shapes with draw methods

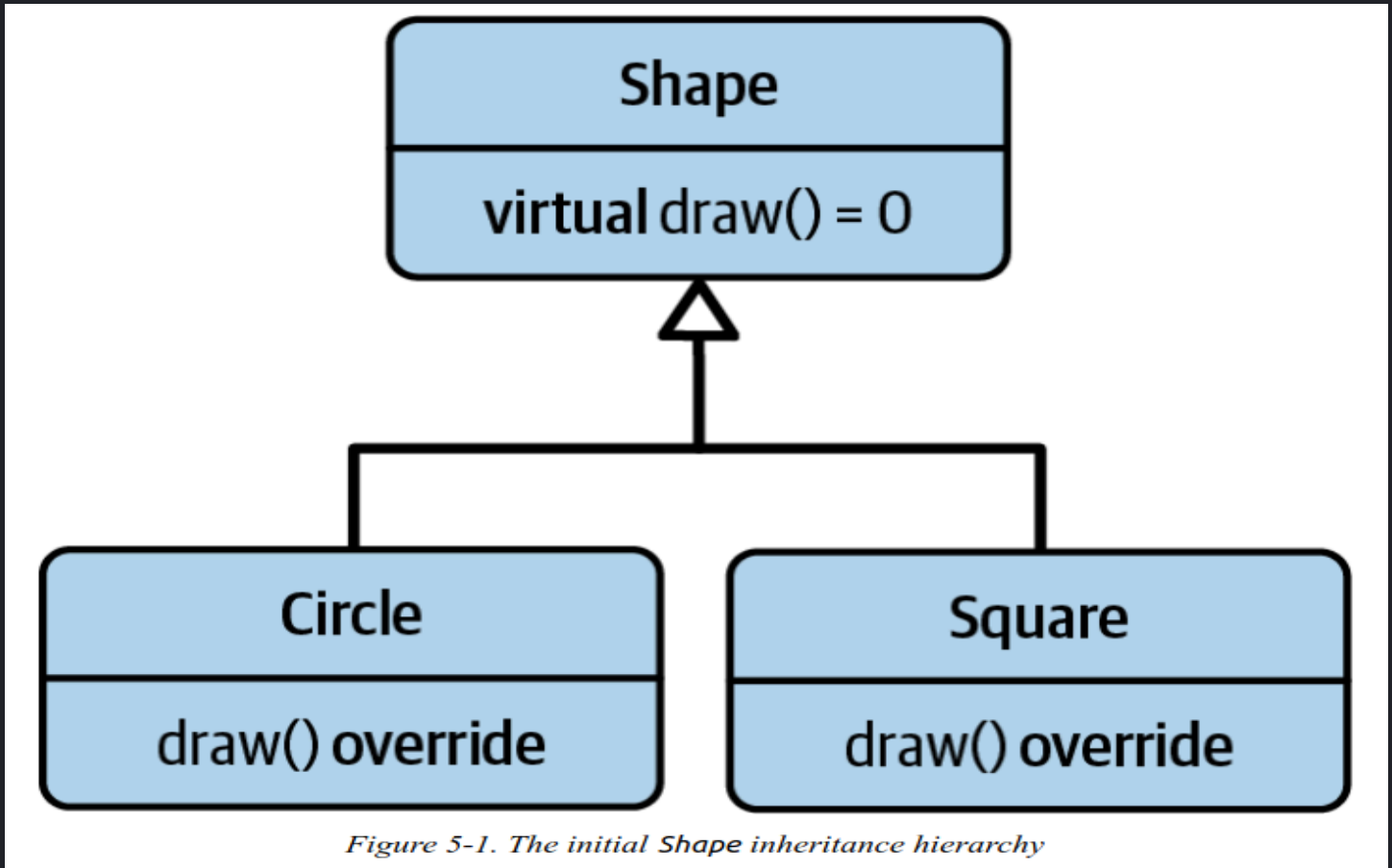
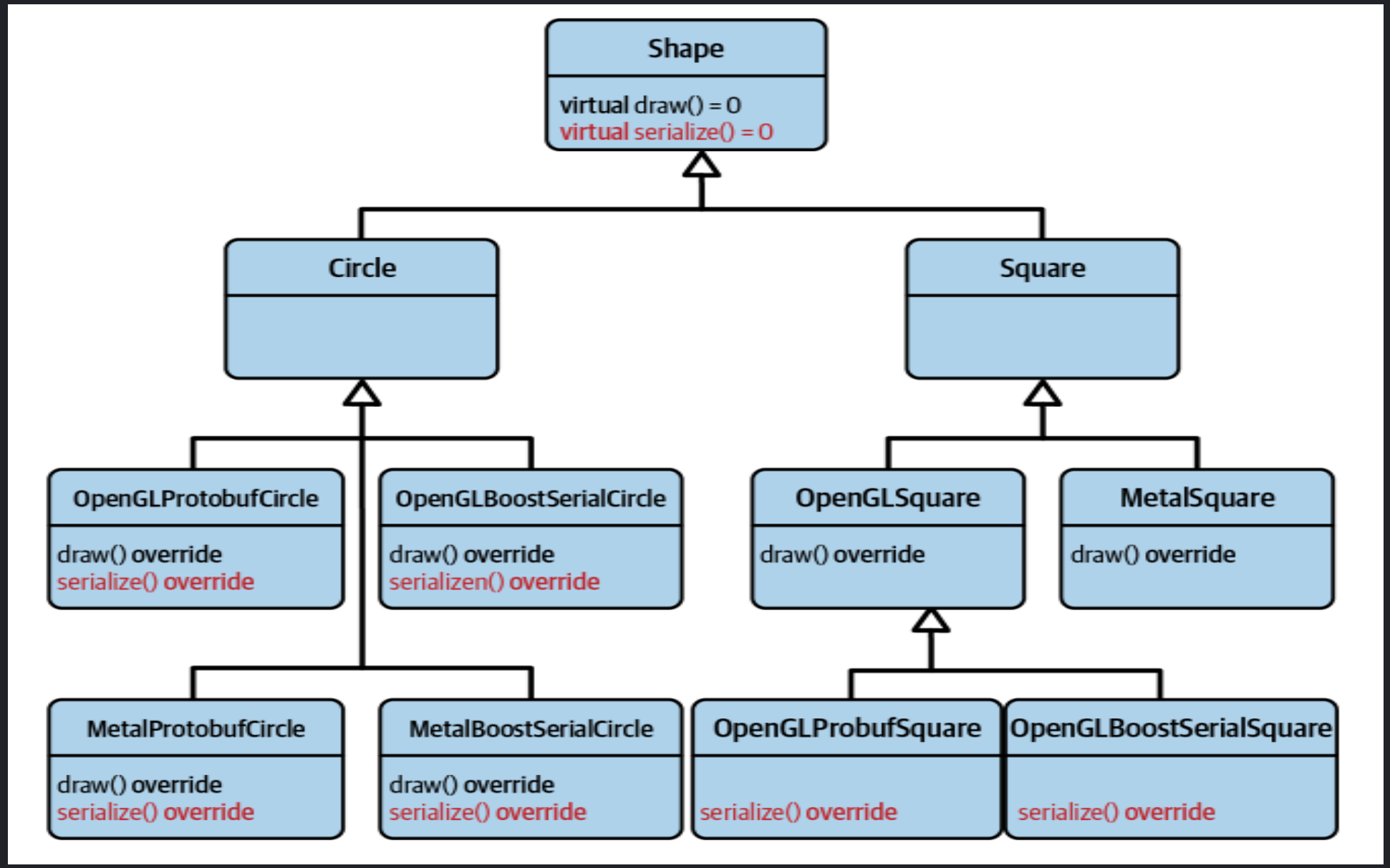



*Figure 5-1. The initial Shape inheritance hierarchy*

# Initial Design: What is the problem here?


Shape
virtual draw() = 0
virtual serialize() = 0
Circle
Square
OpenGLProtobufCircle
draw() override
serialize() override
OpenGLBoostSerialCircle
draw() override
serializen() override
OpenGLSquare
draw() override
MetalSquare
draw() override
MetalProtobufCircle
draw() override
serialize() override
MetalBoostSerialCircle
draw() override
serialize() override
OpenGLProbufSquare
serialize() override
OpenGLBoostSerialSquare
serialize() override

# Let's add strategy pattern to the design.

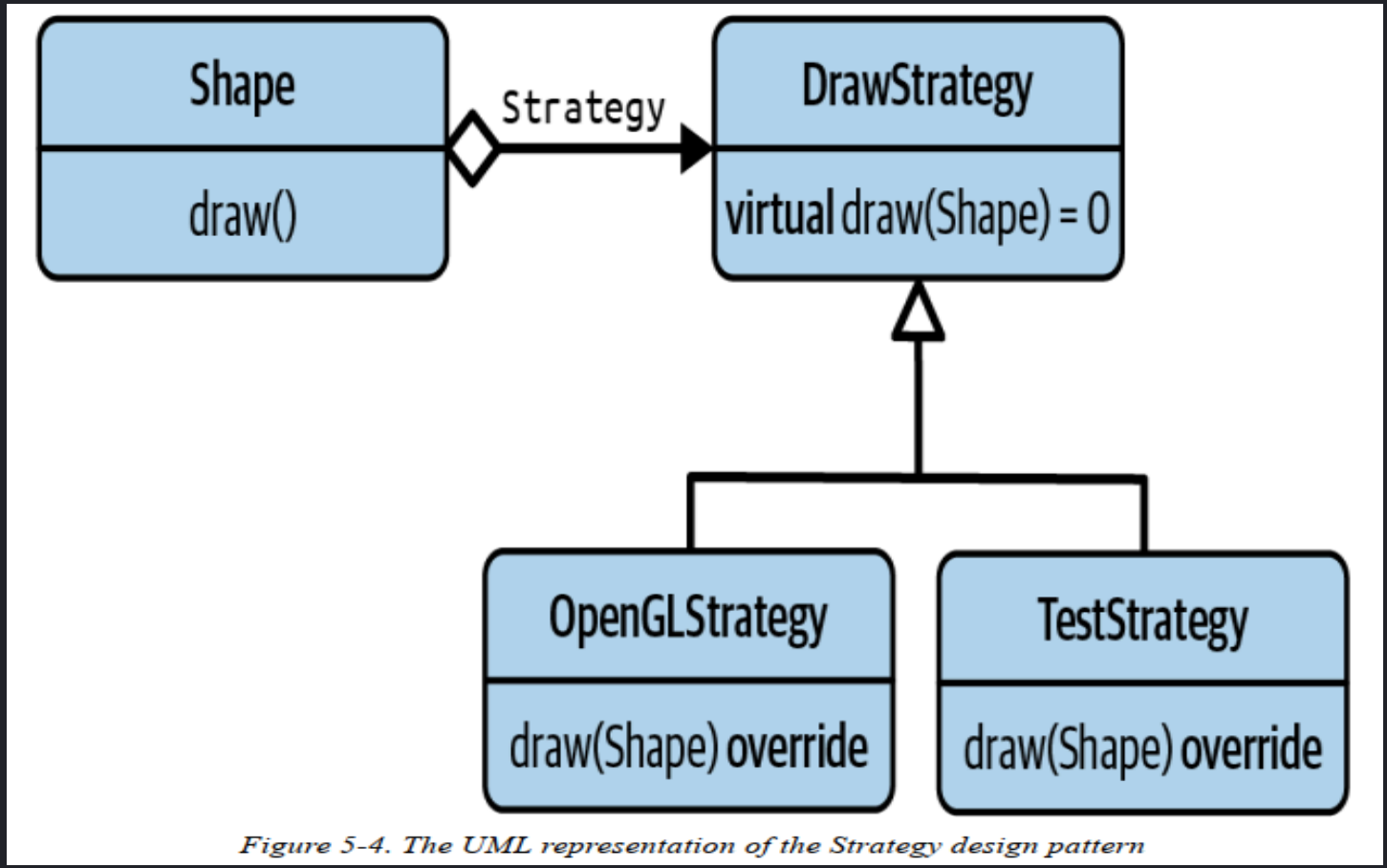
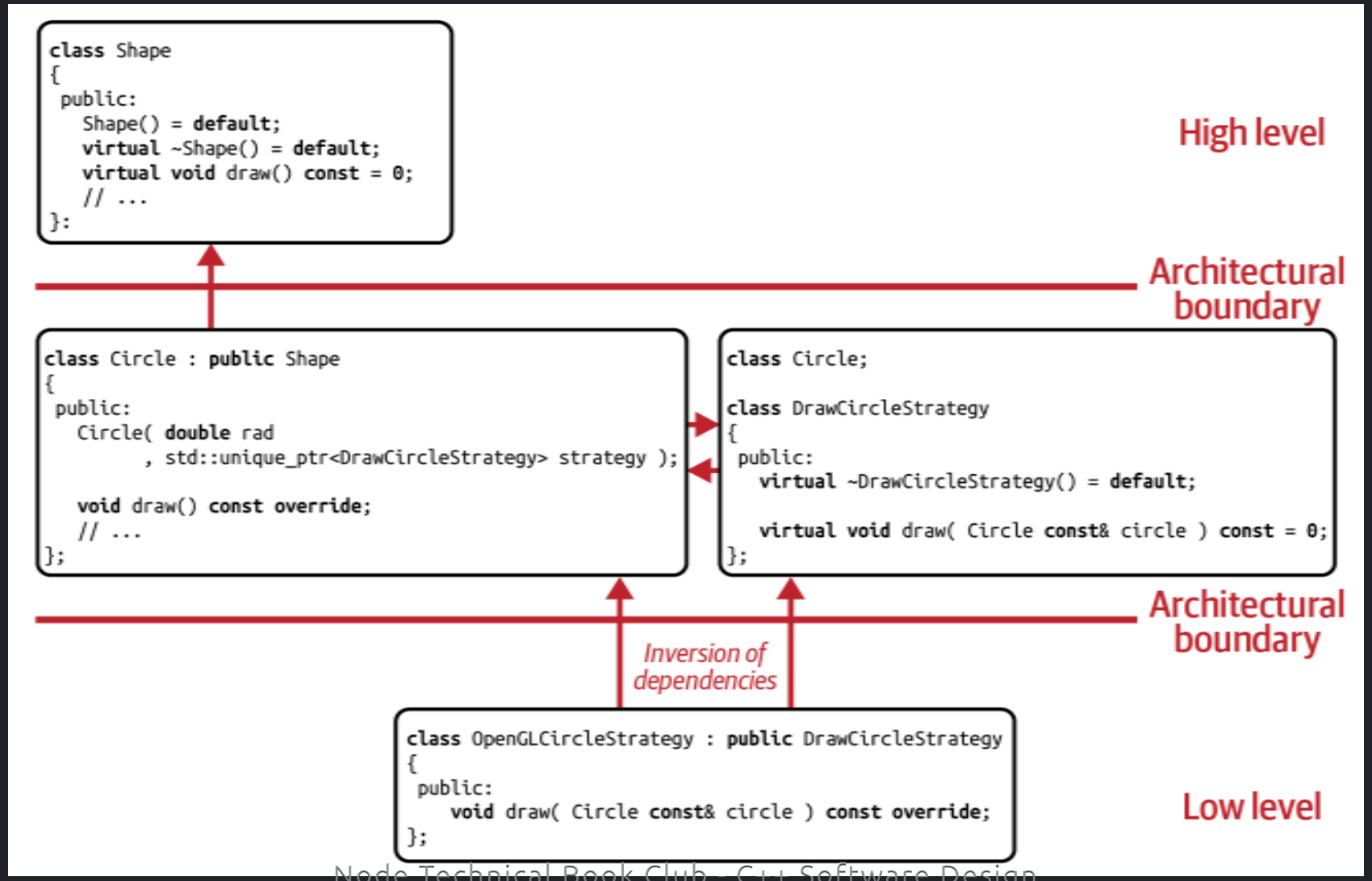


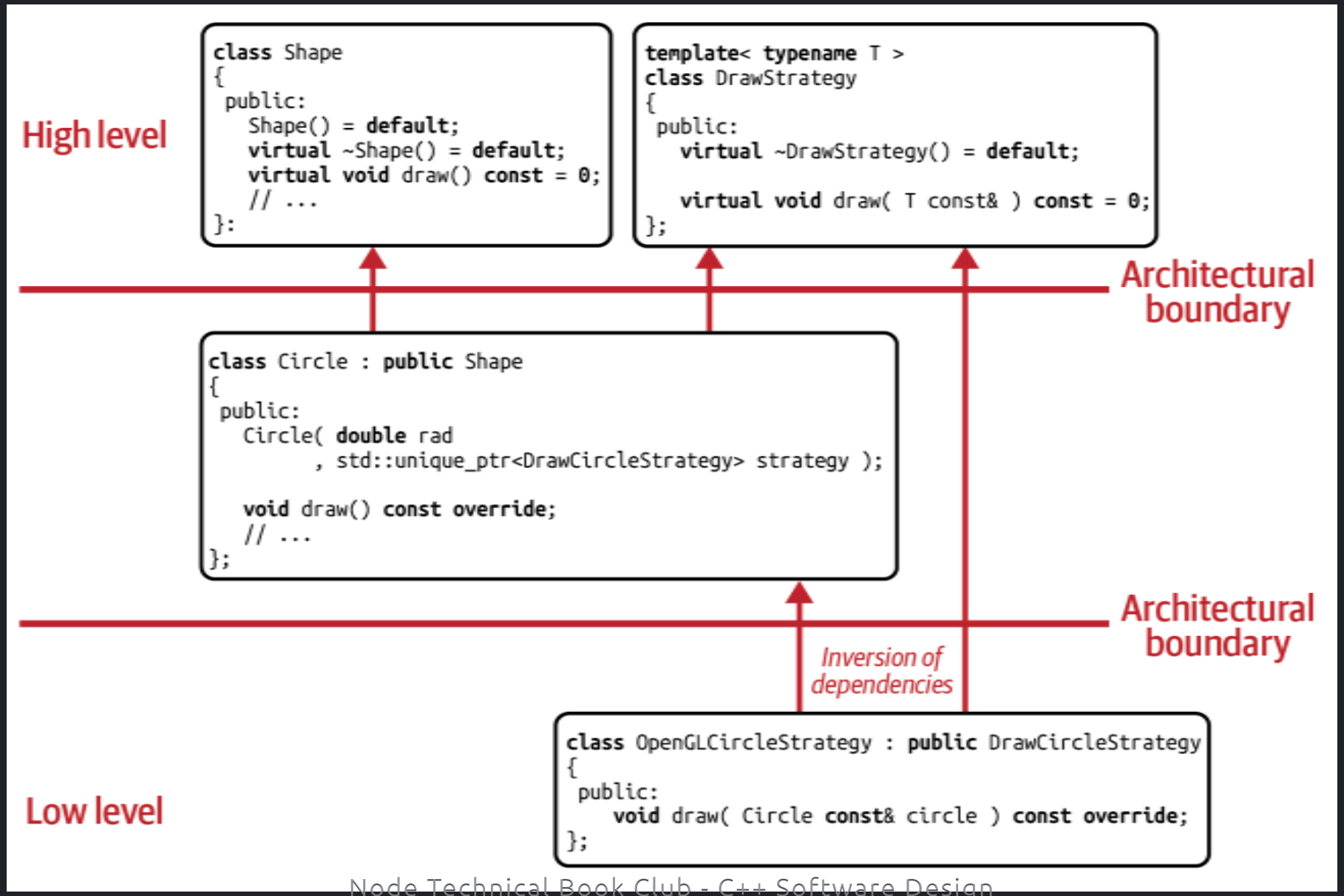

*Figure 5-4. The UML representation of the Strategy design pattern*

- **Naive Solution**: It is not easy to add new types.

# We need to extract the implementation details of each shape separately.



# We can use templates to reuse code.



# Visitor vs. Strategy

- **Visitor**: Addition of operations as varation point. Easy to add new operations but hard to add new types.
- **Strategy**: Implementation details of a single function as variation point. Easy to add new types but hard to add new operations.

# Policy-Based Design

```
template<typename ForwardIt, typename UnaryPredicate>
constexpr ForwardIt
partition(ForwardIt first, ForwardIt last, UnaryPredicate p);

template<typename RandomIt, typename Compare>
constexpr void
sort(RandomIt first, RandomIt last, Compare comp                    );
```

- Both make use of the **Strategy** pattern.
- They allow the user to inject a part of the behavior from the outside.
- This is called **Policy-Based Design**.

# Final Comments?

See you in part 3!