Breaking Dependencies

The Visitor Design Pattern

KLAUS IGLBERGER

Cppcon
The C++ Conference

2022
September 12th-16th

C++ Trainer/Consultant

Author of the blaze C++ math library

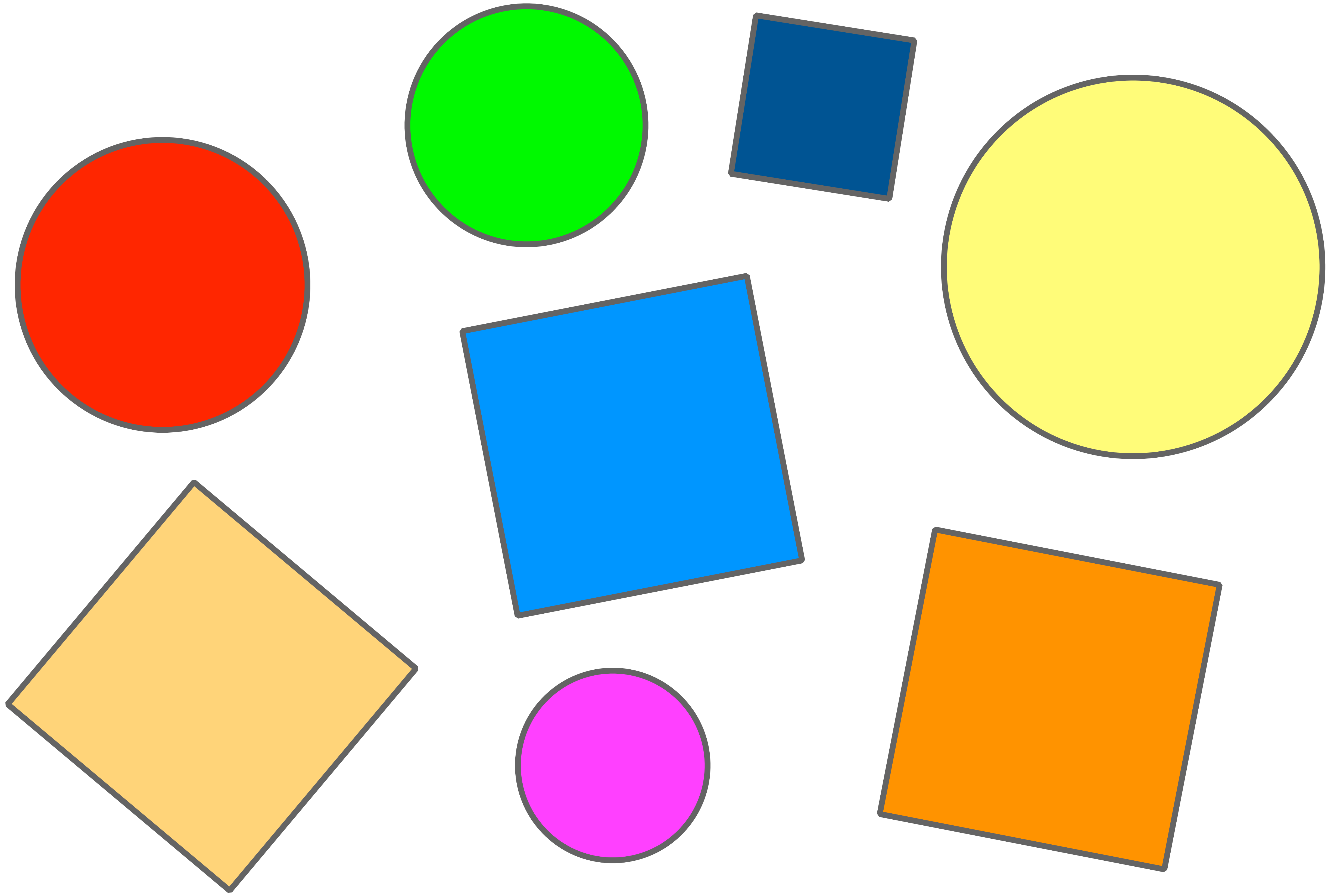
(Co-)Organizer of the Munich C++ user group

Chair of the CppCon B2B track

Email: klaus.iglberger@gmx.de

Klaus Iglberger

Our Toy Problem: Drawing Shapes



A Procedural Solution

```cpp
enum ShapeType
{
   circle,
   square
};

class Shape
{
 public:
   explicit Shape( ShapeType t )
      : type{ t }
   {}

   virtual ~Shape() = default;
   ShapeType getType() const noexcept;

 private:
   ShapeType type;
};


class Circle : public Shape
{
 public:
   explicit Circle( double rad )
      : Shape{ circle }
      , radius{ rad }
      , // ... Remaining data members
   {}

   double getRadius() const noexcept;
```

A Procedural Solution

```
enum ShapeType
{
   circle,
   square
};

class Shape
{
 public:
   explicit Shape( ShapeType t )
      : type{ t }
   {}

   virtual ~Shape() = default;
   ShapeType getType() const noexcept;

 private:
   ShapeType type;
};


class Circle : public Shape
{
 public:
   explicit Circle( double rad )
      : Shape{ circle }
      , radius{ rad }
      , // ... Remaining data members
   {}

   double getRadius() const noexcept;
```

A Procedural Solution

```
enum ShapeType
{
   circle,
   square
};

class Shape
{
 public:
   explicit Shape( ShapeType t )
      , type{ t }
   {}

   virtual ~Shape() = default;
   ShapeType getType() const noexcept;

 private:
   ShapeType type;
};


class Circle : public Shape
{
 public:
   explicit Circle( double rad )
      : Shape{ circle }
      , radius{ rad }
      , // ... Remaining data members
   {}

   double getRadius() const noexcept;
```

A Procedural Solution

```
 private:
   ShapeType type;
};


class Circle : public Shape
{
 public:
   explicit Circle( double rad )
      : Shape{ circle }
      , radius{ rad }
      , // ... Remaining data members
   {}

   double getRadius() const noexcept;
   // ... getCenter(), getRotation(), ...
   // ...

 private:
   double radius;
   // ... Remaining data members
};

void translate( Circle&, Vector2D const& );
void rotate( Circle&, double const& );
void draw( Circle const& );


class Square : public Shape
{
 public:
   explicit Square( double s )
      : Shape{ square }
      , side{ s }
```

A Procedural Solution

```
void translate( Circle&, Vector2D const& );
void rotate( Circle&, double const& );
void draw( Circle const& );


class Square : public Shape
{
 public:
   explicit Square( double s )
      : Shape{ square }
      , side{ s }
      , // ... Remaining data members
   {}

   double getSide() const noexcept;
   // ... getCenter(), getRotation(), ...
   // ...

 private:
   double side;
   // ... Remaining data members
};

void translate( Square&, Vector2D const& );
void rotate( Square&, double const& );
void draw( Square const& );


void drawAllShapes( std::vector<std::unique_ptr<Shape>> const& shapes )
{
   for( auto const& s : shapes )
   {
      switch ( s->getType() )
      {
```

A Procedural Solution

```
   double side;
   // ... Remaining data members
};

void translate( Square&, Vector2D const& );
void rotate( Square&, double const& );
void draw( Square const& );


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


int main()
{
   using Shapes = std::vector<std::unique_ptr<Shape>>;

   // Creating some shapes
   Shapes shapes;
   shapes.emplace_back( std::make_unique<Circle>( 2.0 ) );
   shapes.emplace_back( std::make_unique<Square>( 1.5 ) );
   shapes.emplace_back( std::make_unique<Circle>( 4.2 ) );
```

A Procedural Solution

```
            draw( *static_cast<Circle const*>( s.get() ) );
            break;
         case square:
            draw( *static_cast<Square const*>( s.get() ) );
            break;
      }
   }
}


int main()
{
   using Shapes = std::vector<std::unique_ptr<Shape>>;

   // Creating some shapes
   Shapes shapes;
   shapes.emplace_back( std::make_unique<Circle>( 2.0 ) );
   shapes.emplace_back( std::make_unique<Square>( 1.5 ) );
   shapes.emplace_back( std::make_unique<Circle>( 4.2 ) );

   // Drawing all shapes
   drawAllShapes( shapes );
}
```

It works!

Amazing, isn’t it?

😐 😱 🤢

The Problems of Singletons

”This kind of type-based programming has a long history in C, and one of the things we know about it is that it yields programs that are essentially unmaintainable.”

(Scott Meyers, More Effective C++, Item 31)

The Problems of Singletons

”This kind of type-based programming has a long history in C, and one of the things we know about it is that it yields programs that are essentially unmaintainable.”

(Scott Meyers, More Effective C++, Item 31)

There is one constant in **soft**ware development and that is …

Change

The truth in our industry:

Software must be adaptable to frequent changes

The truth in our industry:

Software must be adaptable to frequent changes

A Procedural Solution

```
enum ShapeType
{
   circle,
   square,
   rectangle
};

class Shape
{
 public:
   explicit Shape( ShapeType t )
      : type{ t }
   {}

   virtual ~Shape() = default;
   ShapeType getType() const noexcept;

 private:
   ShapeType type;
};

class Circle : public Shape
{
 public:
   explicit Circle( double rad )
      : Shape{ circle }
      , radius{ rad }
      , // ... Remaining data members
   {}
```

A Procedural Solution

```
 private:
   ShapeType type;
};


class Circle : public Shape
{
 public:
   explicit Circle( double rad )
      : Shape{ circle }
      , radius{ rad }
      , // ... Remaining data members
   {}

   double getRadius() const noexcept;
   // ... getCenter(), getRotation(), ...
   // ...

 private:
   double radius;
   // ... Remaining data members
};

void translate( Circle&, Vector2D const& );
void rotate( Circle&, double const& );
void draw( Circle const& );


class Square : public Shape
{
 public:
   explicit Square( double s )
      : Shape{ square }
      , side{ s }
```

A Procedural Solution

```
void translate( Circle&, Vector2D const& );
void rotate( Circle&, double const& );
void draw( Circle const& );


class Square : public Shape
{
 public:
   explicit Square( double s )
      : Shape{ square }
      , side{ s }
      , // ... Remaining data members
   {}

   double getSide() const noexcept;
   // ... getCenter(), getRotation(), ...
   // ...

 private:
   double side;
   // ... Remaining data members
};

void translate( Square&, Vector2D const& );
void rotate( Square&, double const& );
void draw( Square const& );


void drawAllShapes( std::vector<std::unique_ptr<Shape>> const& shapes )
{
   for( auto const& s : shapes )
   {
      switch ( s->getType() )
      {
```

A Procedural Solution

```
void rotate( Square&, double const& );
void draw( Square const& );


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


int main()
{
   using Shapes = std::vector<std::unique_ptr<Shape>>;

   // Creating some shapes
   Shapes shapes;
   shapes.emplace_back( std::make_unique<Circle>( 2.0 ) );
   shapes.emplace_back( std::make_unique<Square>( 1.5 ) );
   shapes.emplace_back( std::make_unique<Circle>( 4.2 ) );
```

So how would we approach the problem differently?

Object-oriented programming

(of course)

An Object-Oriented Solution

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

   virtual void draw( /*...*/ ) const = 0;
};


class Circle : public Shape
{
 public:
   explicit Circle( double rad )
      : radius{ rad }
      , // ... Remaining data members
   {}

   double getRadius() const noexcept;
   // ... getCenter(), getRotation(), ...

   void draw( /*...*/ ) const override;

   // ...

 private:
   double radius;
   // ... Remaining data members
};
```

An Object-Oriented Solution

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

   virtual void draw( /*...*/ ) const = 0;
};


class Circle : public Shape
{
 public:
   explicit Circle( double rad )
      : radius{ rad }
      , // ... Remaining data members
   {}

   double getRadius() const noexcept;
   // ... getCenter(), getRotation(), ...

   void draw( /*...*/ ) const override;

   // ...

 private:
   double radius;
   // ... Remaining data members
};
```

An Object-Oriented Solution

```
 public:
   Shape() = default;
   virtual ~Shape() = default;

   virtual void draw( /*...*/ ) const = 0;
};


class Circle : public Shape
{
 public:
   explicit Circle( double rad )
      : radius{ rad }
      , // ... Remaining data members
   {}

   double getRadius() const noexcept;
   // ... getCenter(), getRotation(), ...

   void draw( /*...*/ ) const override;

   // ...

 private:
   double radius;
   // ... Remaining data members
};


class Square : public Shape
{
 public:
   explicit Square( double s )
      : side{ s }
```

An Object-Oriented Solution

```
 private:
   double radius;
   // ... Remaining data members
};


class Square : public Shape
{
 public:
   explicit Square( double s )
      : side{ s }
      , // ... Remaining data members
   {}

   double getSide() const noexcept;
   // ... getCenter(), getRotation(), ...

   void draw( /*...*/ ) const override;

   // ...

 private:
   double side;
   // ... Remaining data members
};


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

An Object-Oriented Solution

```
 private:
   double side;
   // ... Remaining data members
};


void drawAllShapes( std::vector<std::unique_ptr<Shape>> const& shapes )
{
   for( auto const& s : shapes )
   {
      s->draw( /*...*/ );
   }
}


int main()
{
   using Shapes = std::vector<std::unique_ptr<Shape>>;

   // Creating some shapes
   Shapes shapes;
   shapes.emplace_back( std::make_unique<Circle>( 2.0 ) );
   shapes.emplace_back( std::make_unique<Square>( 1.5 ) );
   shapes.emplace_back( std::make_unique<Circle>( 4.2 ) );

   // Drawing all shapes
   drawAllShapes( shapes );
}
```

An Object-Oriented Solution

```
 private:
   double side;
   // ... Remaining data members
};


void drawAllShapes( std::vector<std::unique_ptr<Shape>> const& shapes )
{
   for( auto const& s : shapes )
   {
      s->draw( /*...*/ );
   }
}


int main()
{
   using Shapes = std::vector<std::unique_ptr<Shape>>;

   // Creating some shapes
   Shapes shapes;
   shapes.emplace_back( std::make_unique<Circle>( 2.0 ) );
   shapes.emplace_back( std::make_unique<Square>( 1.5 ) );
   shapes.emplace_back( std::make_unique<Circle>( 4.2 ) );

   // Drawing all shapes
   drawAllShapes( shapes );
}
```

An Object-Oriented Solution

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

   virtual void draw( /*...*/ ) const = 0;



};

class Circle : public Shape
{
 public:
   explicit Circle( double rad )
      : radius{ rad }
      , // ... Remaining data members
   {}

   double getRadius() const noexcept;
   // ... getCenter(), getRotation(), ...

   void draw( /*...*/ ) const override;

   // ...

 private:
   double radius;
   // ... Remaining data members
};
```

An Object-Oriented Solution

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

   virtual void draw( /*...*/ ) const = 0;
   virtual void serialize( /*...*/ ) const = 0;
   virtual void translate( Vector2D const& ) = 0;
   virtual void rotate( double const& ) = 0;
};

class Circle : public Shape
{
 public:
   explicit Circle( double rad )
      : radius{ rad }
      , // ... Remaining data members
   {}

   double getRadius() const noexcept;
   // ... getCenter(), getRotation(), ...

   void draw( /*...*/ ) const override;

   // ...

 private:
   double radius;
   // ... Remaining data members
};
```

An Object-Oriented Solution

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

   virtual void draw( /*...*/ ) const = 0;
   virtual void serialize( /*...*/ ) const = 0;
   virtual void translate( Vector2D const& ) = 0;
   virtual void rotate( double const& ) = 0;
};
```

An OO solution may *appear* better, but has two serious flaws:

- Adding operations is intrusive and thus difficult.
- Adding operations accumulates dependencies.

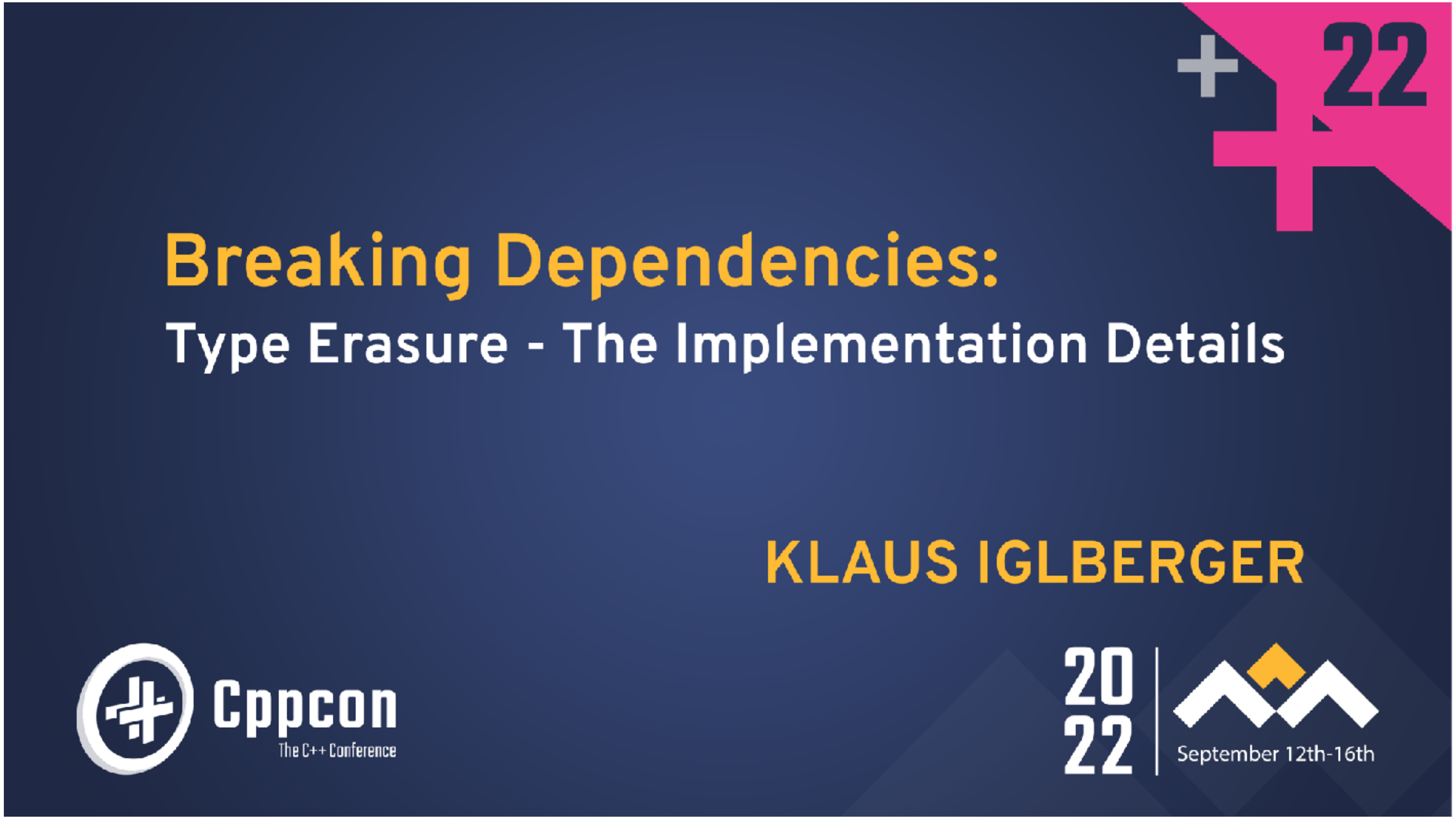
An Object-Oriented Solution

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

   virtual void draw( /*...*/ ) const = 0;
   virtual void serialize( /*...*/ ) const = 0;
   virtual void translate( Vector2D const& ) = 0;
   virtual void rotate( double const& ) = 0;
};
```

In dynamic polymorphism you have to make a choice:

- Design for the addition of types
- Design for the addition of operations

Design for the Addition of Types



Breaking Dependencies:

Type Erasure - The Implementation Details

KLAUS IGLBERGER

22
Cppcon
The C++ Conference
20
22
September 12th-16th

Design for the Addition of Operations

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

   virtual void draw( /*...*/ ) const = 0;
   virtual void serialize( /*...*/ ) const = 0;
   virtual void translate( Vector2D const& ) = 0;
   virtual void rotate( double const& ) = 0;
};
```

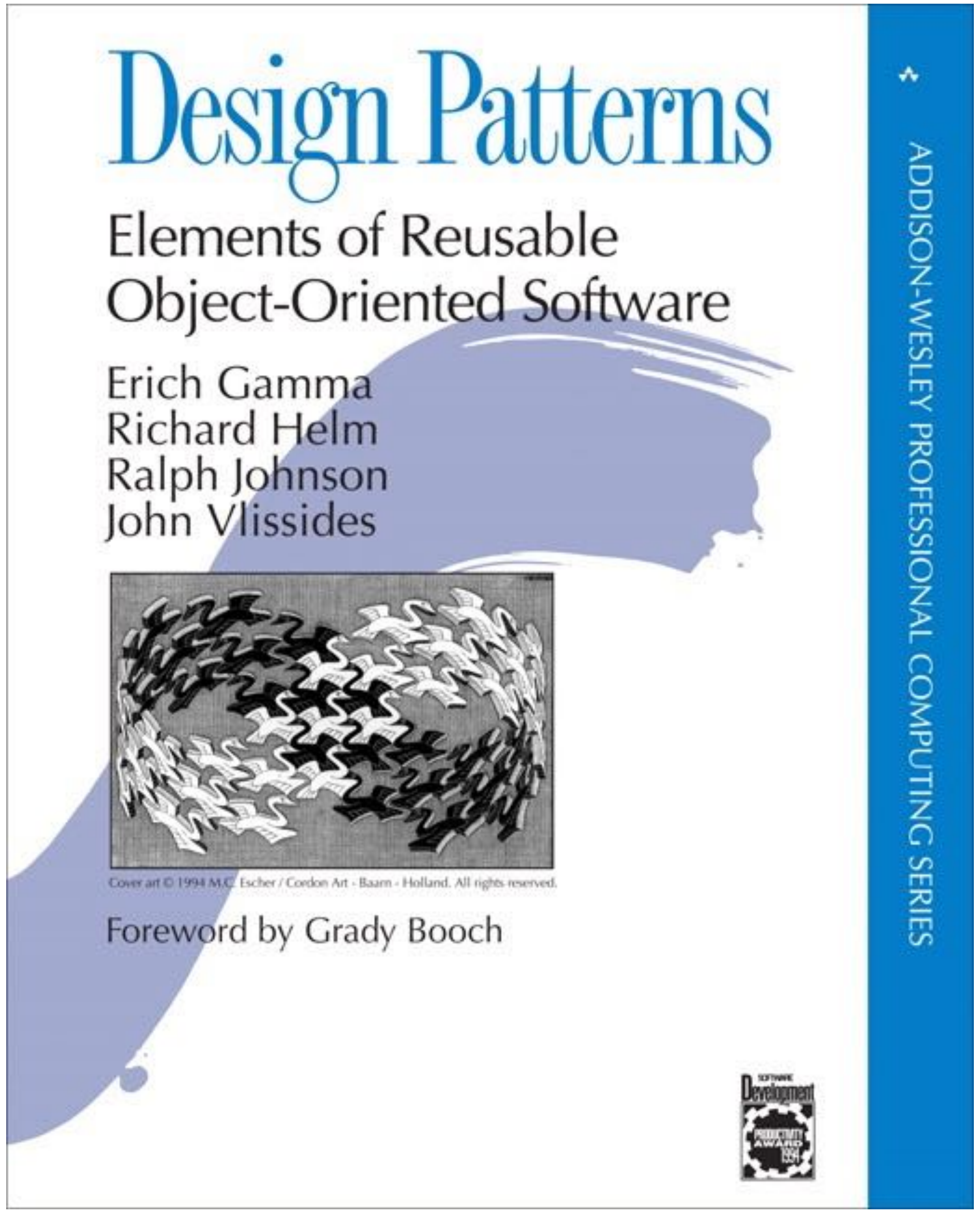
Let’s assume for the remainder of this talk that we want to add operations.

Changing interfaces in OOP is difficult!

OOP is the WRONG choice if you need to add operations!

Or is it?

The Design Pattern Reference



Design Patterns

Elements of Reusable Object-Oriented Software

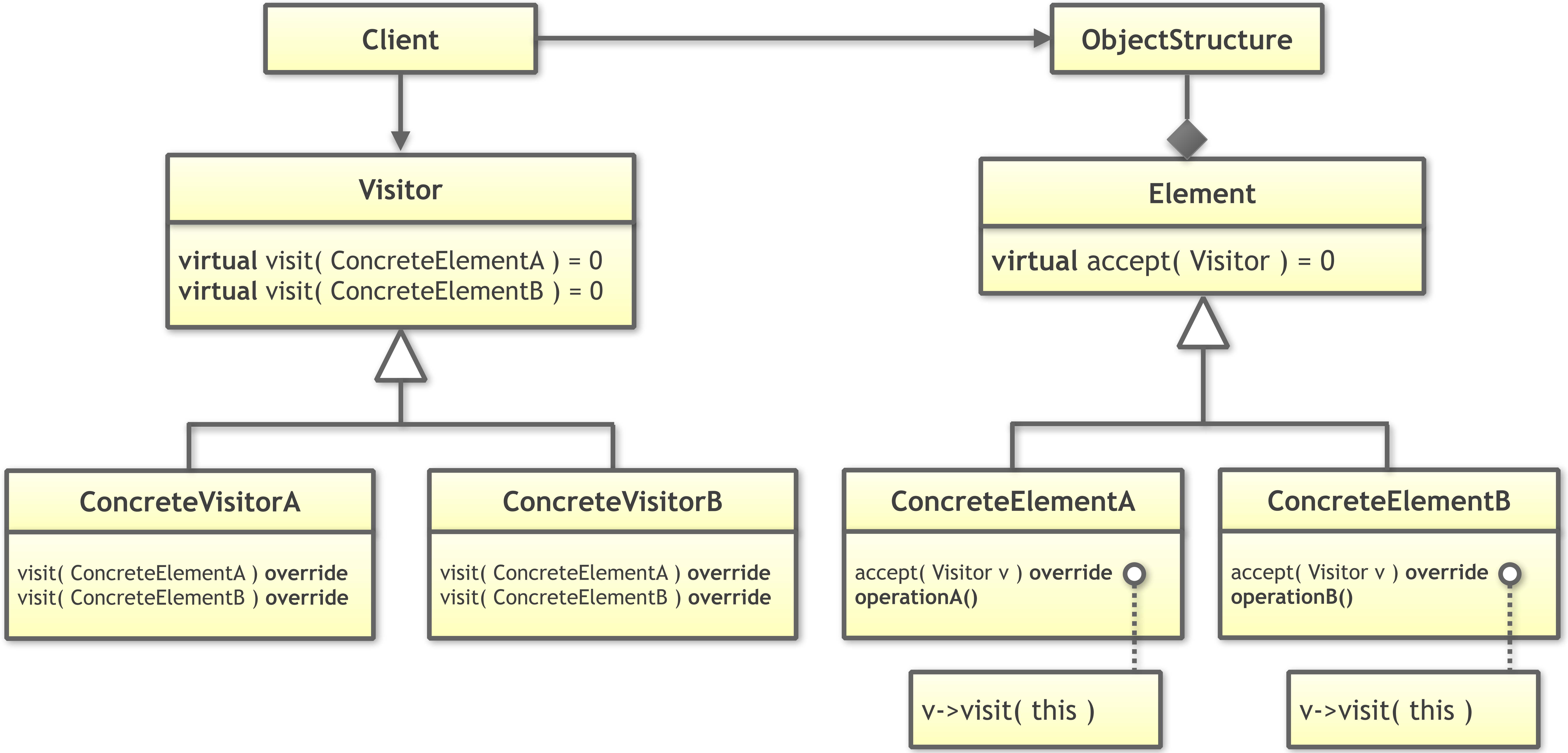
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides

Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.

Foreword by Grady Booch

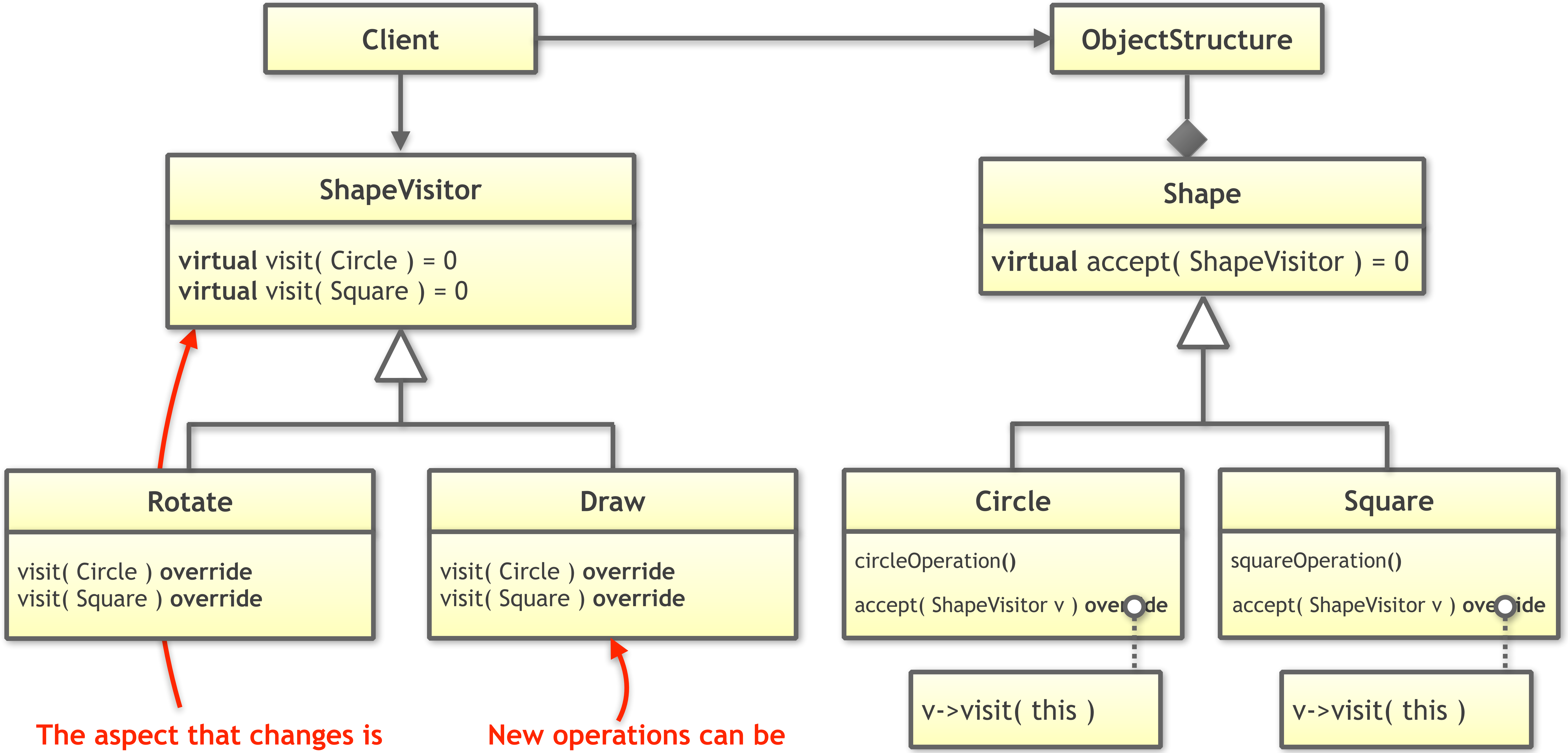
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

The Classic Visitor Design Pattern



Client
ObjectStructure
Visitor
virtual visit(ConcreteElementA) = 0
virtual visit(ConcreteElementB) = 0
Element
virtual accept(Visitor) = 0
ConcreteVisitorA
visit(ConcreteElementA) override
visit(ConcreteElementB) override
ConcreteVisitorB
visit(ConcreteElementA) override
visit(ConcreteElementB) override
ConcreteElementA
accept(Visitor v) override
operationA()
ConcreteElementB
accept(Visitor v) override
operationB()
v->visit(this)
v->visit(this)

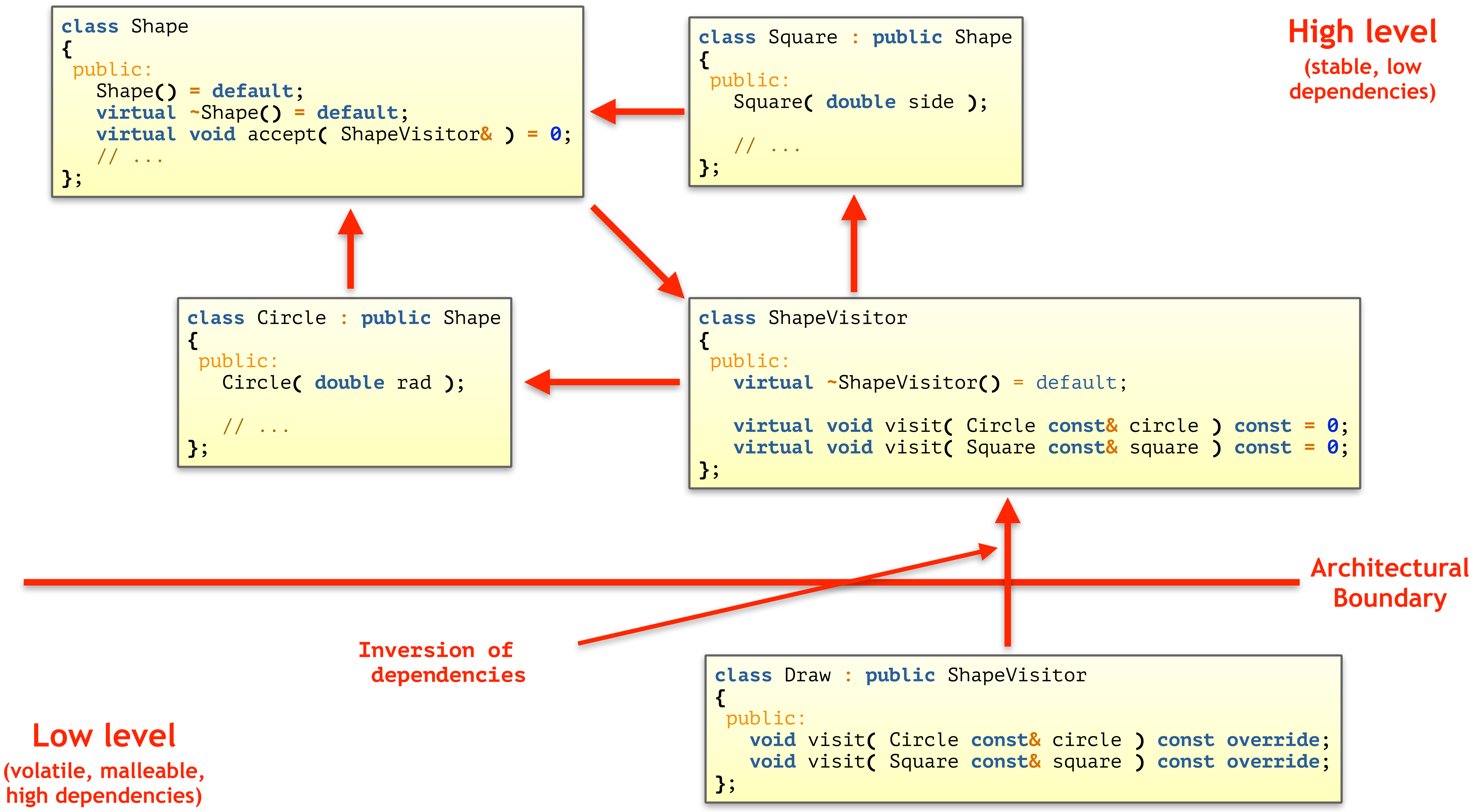
The Classic Visitor Design Pattern



The aspect that changes is extracted and isolated; this fulfills the Single-Responsibility Principle (SRP)

New operations can be added without modifying any existing code; this fulfills the Open-Closed Principle (OCP)

The Classic Visitor Design Pattern



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

```
class Square : public Shape
{
 public:
   Square( double side );

   // ...
};
```

High level

(stable, low dependencies)

```
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;
};
```

Architectural Boundary

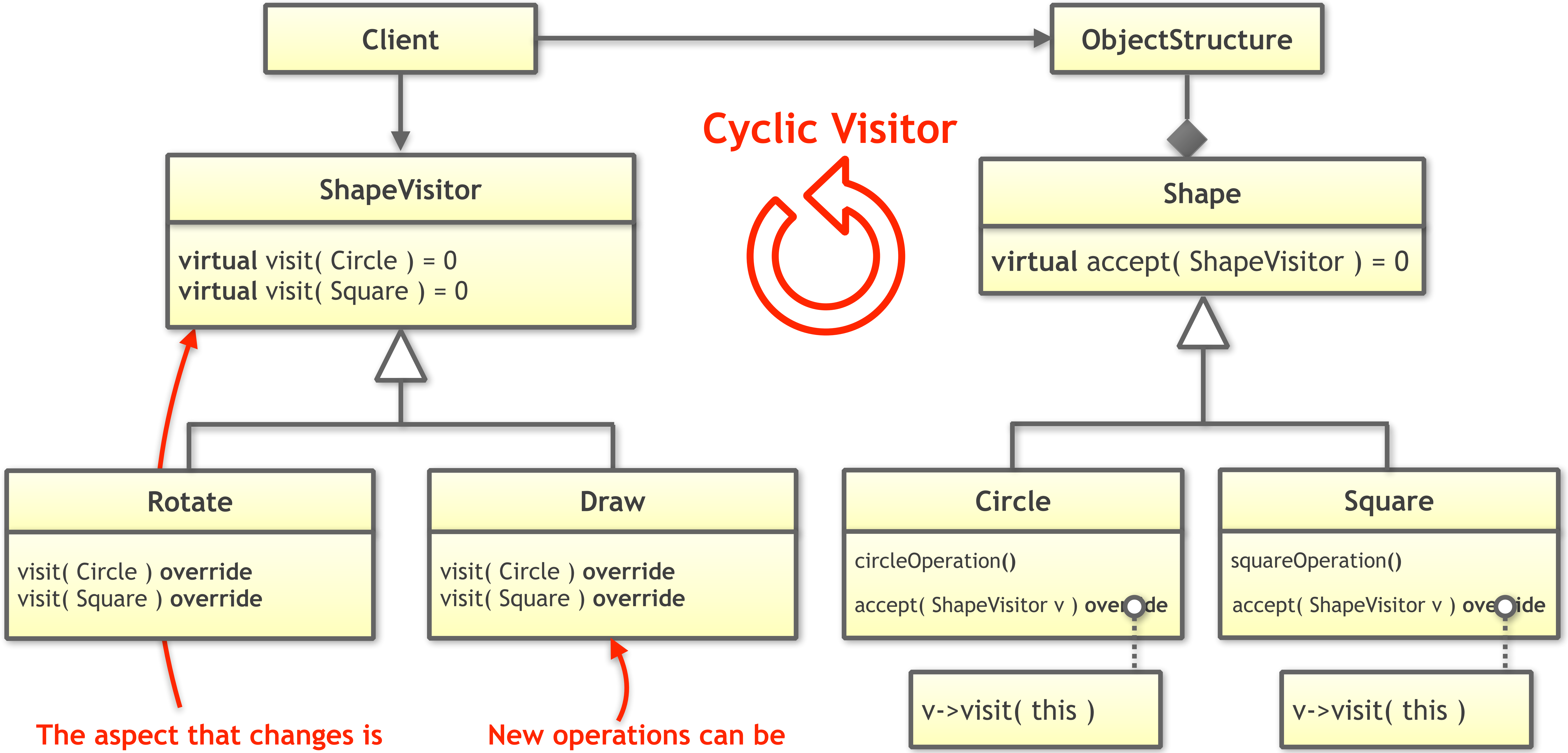
Inversion of dependencies

Low level

(volatile, malleable, high dependencies)

```
class Draw : public ShapeVisitor
{
 public:
   void visit( Circle const& circle ) const override;
   void visit( Square const& square ) const override;
};
```

The Classic Visitor Design Pattern



Client

ObjectStructure

Cyclic Visitor

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

circleOperation**()**

accept(ShapeVisitor v) **override**

v->visit(this)

Square

squareOperation**()**

accept(ShapeVisitor v) **override**

v->visit(this)

The aspect that changes is extracted and isolated; this fulfills the Single-Responsibility Principle (SRP)

New operations can be added without modifying any existing code; this fulfills the Open-Closed Principle (OCP)

A Visitor-Based Solution

```
class Circle;
class Square;

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

   virtual void visit( Circle const& ) const = 0;
   virtual void visit( Square const& ) const = 0;
};


class Shape
{
 public:
   Shape() = default;
   virtual ~Shape() = default;

   virtual void accept( ShapeVisitor const& ) = 0;
};


class Circle : public Shape
{
 public:
   explicit Circle( double rad )
      : radius{ rad }
      , // ... Remaining data members
   {}
```

A Visitor-Based Solution

```
class Circle;
class Square;

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

   virtual void visit( Circle const& ) const = 0;
   virtual void visit( Square const& ) const = 0;
};


class Shape
{
 public:
   Shape() = default;
   virtual ~Shape() = default;

   virtual void accept( ShapeVisitor const& ) = 0;
};


class Circle : public Shape
{
 public:
   explicit Circle( double rad )
      : radius{ rad }
      , // ... Remaining data members
   {}
```

A Visitor-Based Solution

```
class Circle;
class Square;

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

   virtual void visit( Circle const& ) const = 0;
   virtual void visit( Square const& ) const = 0;
};


class Shape
{
 public:
   Shape() = default;
   virtual ~Shape() = default;

   virtual void accept( ShapeVisitor const& ) = 0;
};


class Circle : public Shape
{
 public:
   explicit Circle( double rad )
      : radius{ rad }
      , // ... Remaining data members
   {}
```

A Visitor-Based Solution

```
   virtual ~Shape() = default;

   virtual void accept( ShapeVisitor const& ) = 0;
};


class Circle : public Shape
{
 public:
   explicit Circle( double rad )
      : radius{ rad }
      , // ... Remaining data members
   {}

   double getRadius() const noexcept;
   // ... getCenter(), getRotation(), ...

   void accept( ShapeVisitor const& ) override;

   // ...

 private:
   double radius;
   // ... Remaining data members
};


class Square : public Shape
{
 public:
   explicit Square( double s )
      : side{ s }
      , // ... Remaining data members
   {}
```

A Visitor-Based Solution

```
 private:
   double radius;
   // ... Remaining data members
};


class Square : public Shape
{
 public:
   explicit Square( double s )
      : side{ s }
      , // ... Remaining data members
   {}

   double getSide() const noexcept;
   // ... getCenter(), getRotation(), ...

   void accept( ShapeVisitor const& ) override;

   // ...

 private:
   double side;
   // ... Remaining data members
};


class Draw : public ShapeVisitor
{
 public:
   void visit( Circle const& ) const override;
   void visit( Square const& ) const override;
};
```

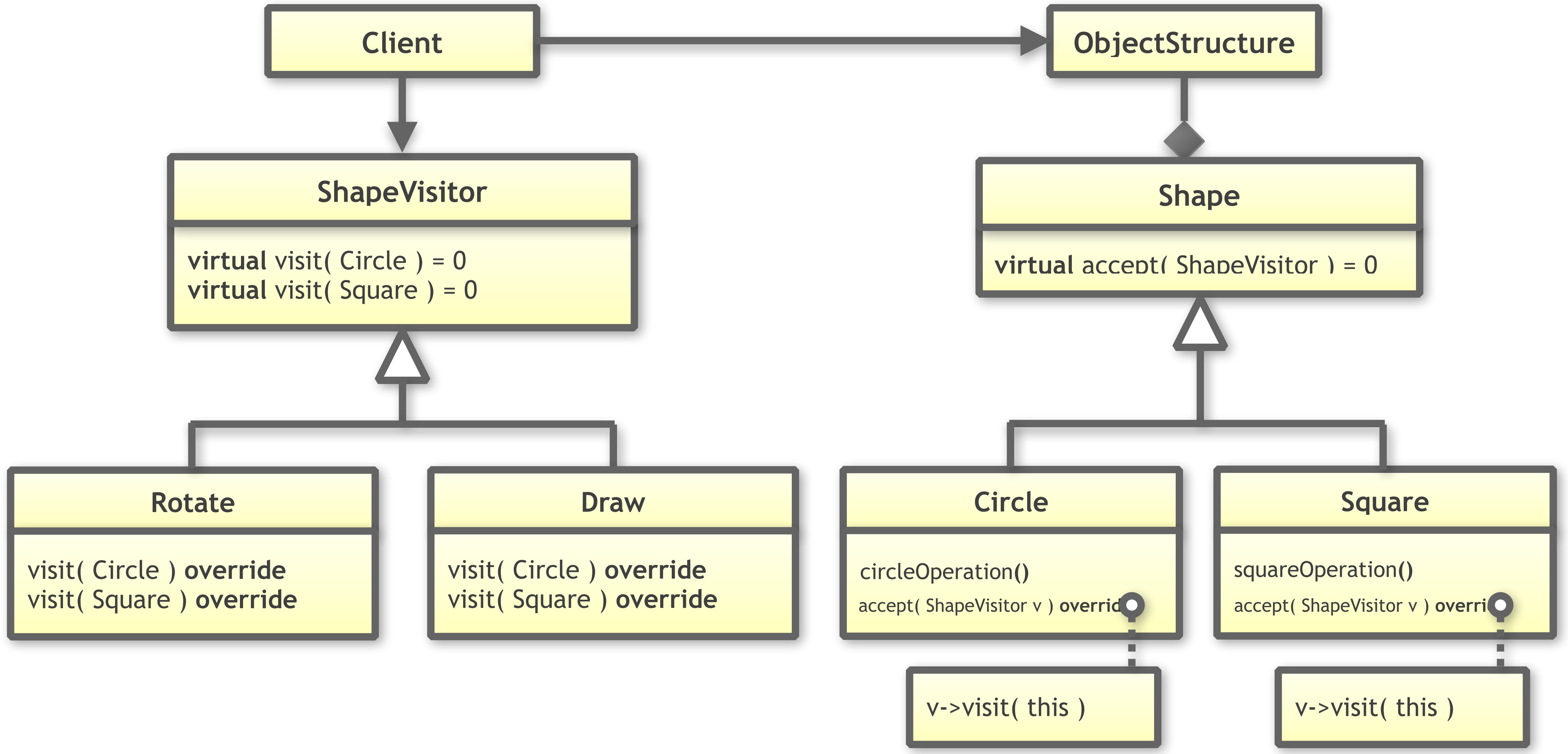
A Visitor-Based Solution

```
 private:
   double side;
   // ... Remaining data members
};


class Draw : public ShapeVisitor
{
 public:
   void visit( Circle const& ) const override;
   void visit( Square const& ) const override;
};


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


int main()
{
   using Shapes = std::vector<std::unique_ptr<Shape>>;

   // Creating some shapes
   Shapes shapes;
   shapes.emplace_back( std::make_unique<Circle>( 2.0 ) );
   shapes.emplace_back( std::make_unique<Square>( 1.5 ) );
   shapes.emplace_back( std::make_unique<Circle>( 4.2 ) );
```

A Visitor-Based Solution

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


int main()
{
   using Shapes = std::vector<std::unique_ptr<Shape>>;

   // Creating some shapes
   Shapes shapes;
   shapes.emplace_back( std::make_unique<Circle>( 2.0 ) );
   shapes.emplace_back( std::make_unique<Square>( 1.5 ) );
   shapes.emplace_back( std::make_unique<Circle>( 4.2 ) );

   // Drawing all shapes
   drawAllShapes( shapes );
}
```

The Classic 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

circleOperation**()**
accept(ShapeVisitor v) **overrid**

Square

squareOperation**()**
accept(ShapeVisitor v) **overri**

v->visit(this)

v->visit(this)

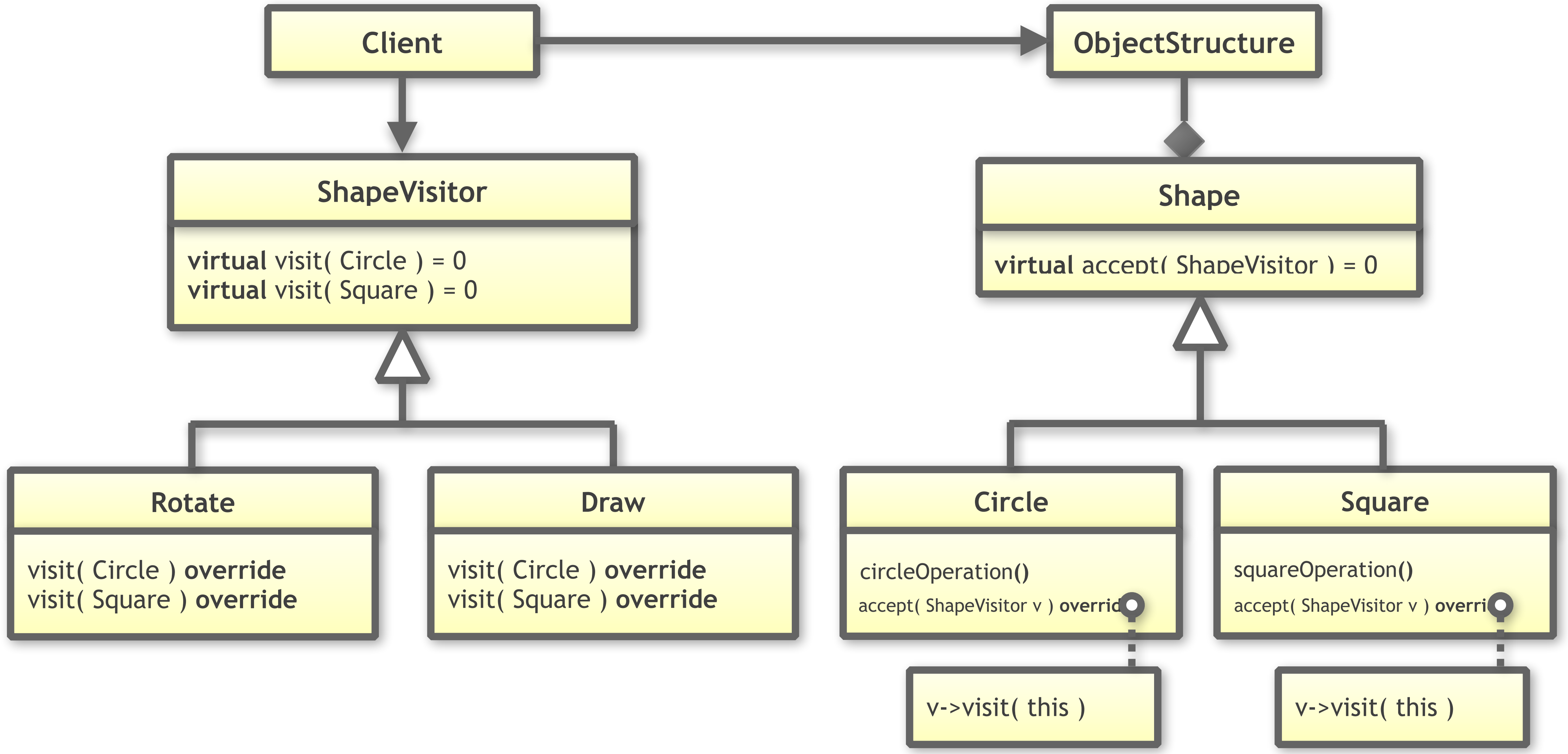
Advantages:

- Allows the non-intrusive addition of operations (OCP)
- Isolates the implementation details of operations (SRP)

Disadvantages:

- Impedes the addition of new types (shapes)
- Restricts operations to the public interface of types
- Negatively affects performance (two virtual functions)

The Classic Visitor Design Pattern



Implementation-specific disadvantages:

- Base class required (intrusive!)
- Promotes heap allocation
- Requires memory management

But there is a modern solution …

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

A “Modern C++” Solution

```
class Circle
{
 public:
   explicit Circle( double rad )
      : radius{ rad }
      , // ... Remaining data members
   {}

   double getRadius() const noexcept;
   // ... getCenter(), getRotation(), ...

 private:
   double radius;
   // ... Remaining data members
};


class Square
{
 public:
   explicit Square( double s )
      : side{ s }
      , // ... Remaining data members
   {}

   double getSide() const noexcept;
   // ... getCenter(), getRotation(), ...

 private:
   double side;
   // ... Remaining data members
```

A “Modern C++” Solution

```
class Circle
{
 public:
   explicit Circle( double rad )
      : radius{ rad }
      , // ... Remaining data members
   {}

   double getRadius() const noexcept;
   // ... getCenter(), getRotation(), ...

 private:
   double radius;
   // ... Remaining data members
};


class Square
{
 public:
   explicit Square( double s )
      : side{ s }
      , // ... Remaining data members
   {}

   double getSide() const noexcept;
   // ... getCenter(), getRotation(), ...

 private:
   double side;
   // ... Remaining data members
```

No base class required!

No accumulation of dependencies via member functions!

A “Modern C++” Solution

```
   // ... getCenter(), getRotation(), ...

 private:
   double radius;
   // ... Remaining data members
};


class Square
{
 public:
   explicit Square( double s )
      : side{ s }
      , // ... Remaining data members
   {}

   double getSide() const noexcept;
   // ... getCenter(), getRotation(), ...

 private:
   double side;
   // ... Remaining data members
};


class Draw
{
 public:
   void operator()( Circle const& ) const;
   void operator()( Square const& ) const;
};


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

A “Modern C++” Solution

```
   // ... getCenter(), getRotation(), ...

 private:
   double side;
   // ... Remaining data members
};


class Draw
{
 public:
   void operator()( Circle const& ) const;
   void operator()( Square const& ) const;
};


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

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


int main()
{
   using Shapes = std::vector<Shape>;

   // Creating some shapes
   Shapes shapes;
   shapes.emplace_back( Circle{ 2.0 } );
```

No base class required!

Operations can be non-intrusively be added (OCP)

A “Modern C++” Solution

```
   // ... getCenter(), getRotation(), ...

 private:
   double side;
   // ... Remaining data members
};


class Draw
{
 public:
   void operator()( Circle const& ) const;
   void operator()( Square const& ) const;
};


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

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


int main()
{
   using Shapes = std::vector<Shape>;

   // Creating some shapes
   Shapes shapes;
   shapes.emplace_back( Circle{ 2.0 } );
```

A shape is a value, representing either a circle or a square

A “Modern C++” Solution

```
   // ... getCenter(), getRotation(), ...

 private:
   double side;
   // ... Remaining data members
};


class Draw
{
 public:
   void operator()( Circle const& ) const;
   void operator()( Square const& ) const;
};


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

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


int main()
{
   using Shapes = std::vector<Shape>;

   // Creating some shapes
   Shapes shapes;
   shapes.emplace_back( Circle{ 2.0 } );
```

The function expects
a vector of values

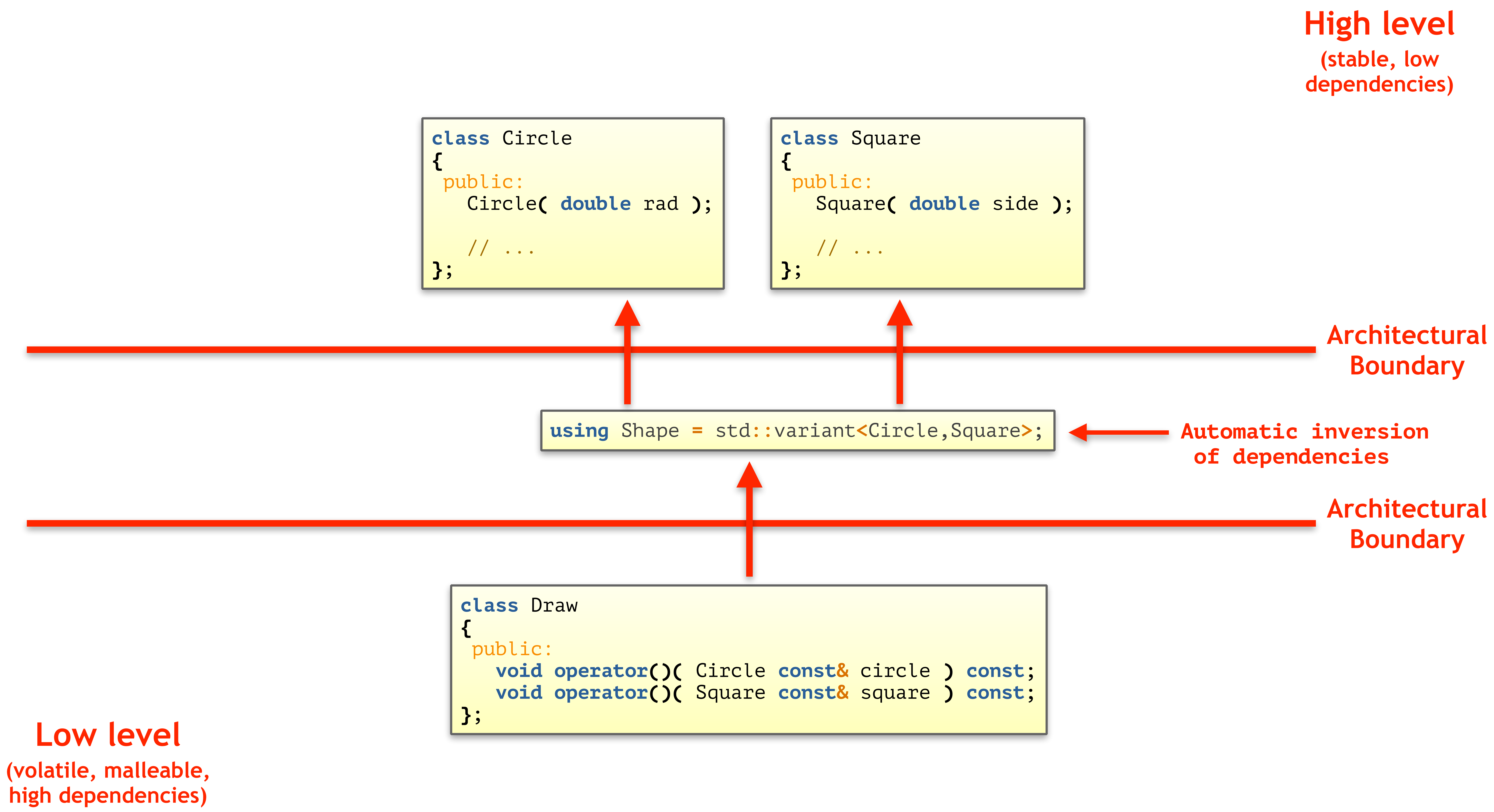
A “Modern C++” Solution

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


int main()
{
   using Shapes = std::vector<Shape>;

   // Creating some shapes
   Shapes shapes;
   shapes.emplace_back( Circle{ 2.0 } );
   shapes.emplace_back( Square{ 1.5 } );
   shapes.emplace_back( Circle{ 4.2 } );

   // Drawing all shapes
   drawAllShapes( shapes );
}
```

No pointers, no allocations, but values …

… and only values, making
the code soooo much
simpler.

A “Modern C++” Solution



High level

(stable, low dependencies)

```
class Circle
{
 public:
   Circle( double rad );

   // ...
};
```

```
class Square
{
 public:
   Square( double side );

   // ...
};
```

Architectural Boundary

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

Automatic inversion of dependencies

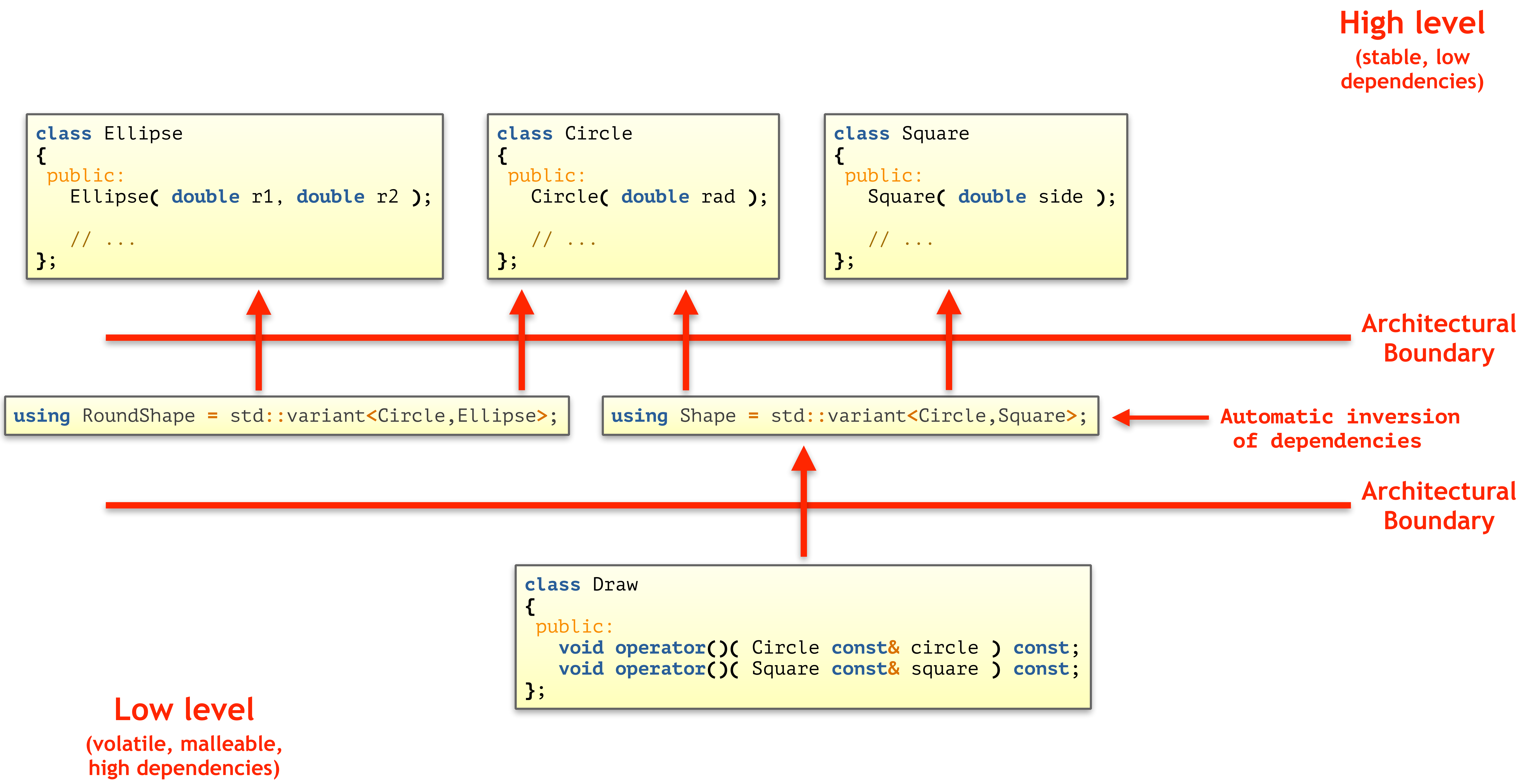
Architectural Boundary

```
class Draw
{
 public:
   void operator()( Circle const& circle ) const;
   void operator()( Square const& square ) const;
};
```

Low level

(volatile, malleable, high dependencies)

A “Modern C++” Solution



High level

(stable, low dependencies)

```cpp
class Ellipse
{
 public:
   Ellipse( double r1, double r2 );

   // ...
};
```

```cpp
class Circle
{
 public:
   Circle( double rad );

   // ...
};
```

```cpp
class Square
{
 public:
   Square( double side );

   // ...
};
```

Architectural Boundary

```cpp
using RoundShape = std::variant<Circle,Ellipse>;
```

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

Automatic inversion of dependencies

Architectural Boundary

```cpp
class Draw
{
 public:
   void operator()( Circle const& circle ) const;
   void operator()( Square const& square ) const;
};
```

Low level

(volatile, malleable, high dependencies)

Evaluation of the Modern Visitor Style

This style of programming has many advantages:

- There is **no inheritance** hierarchy (non-intrusive)
- **No cyclic dependency** (implementation flexibility)
- The code is **so much simpler** (KISS)
- There are **no virtual functions**
- There are **no pointers** or indirections
- There is **no manual dynamic memory** allocation
- There is **no need to manage lifetime**
- There is **no lifetime-related issue** (no need for smart pointers)
- The **performance** is better

These are the advantages of value semantics!

Performance Comparison

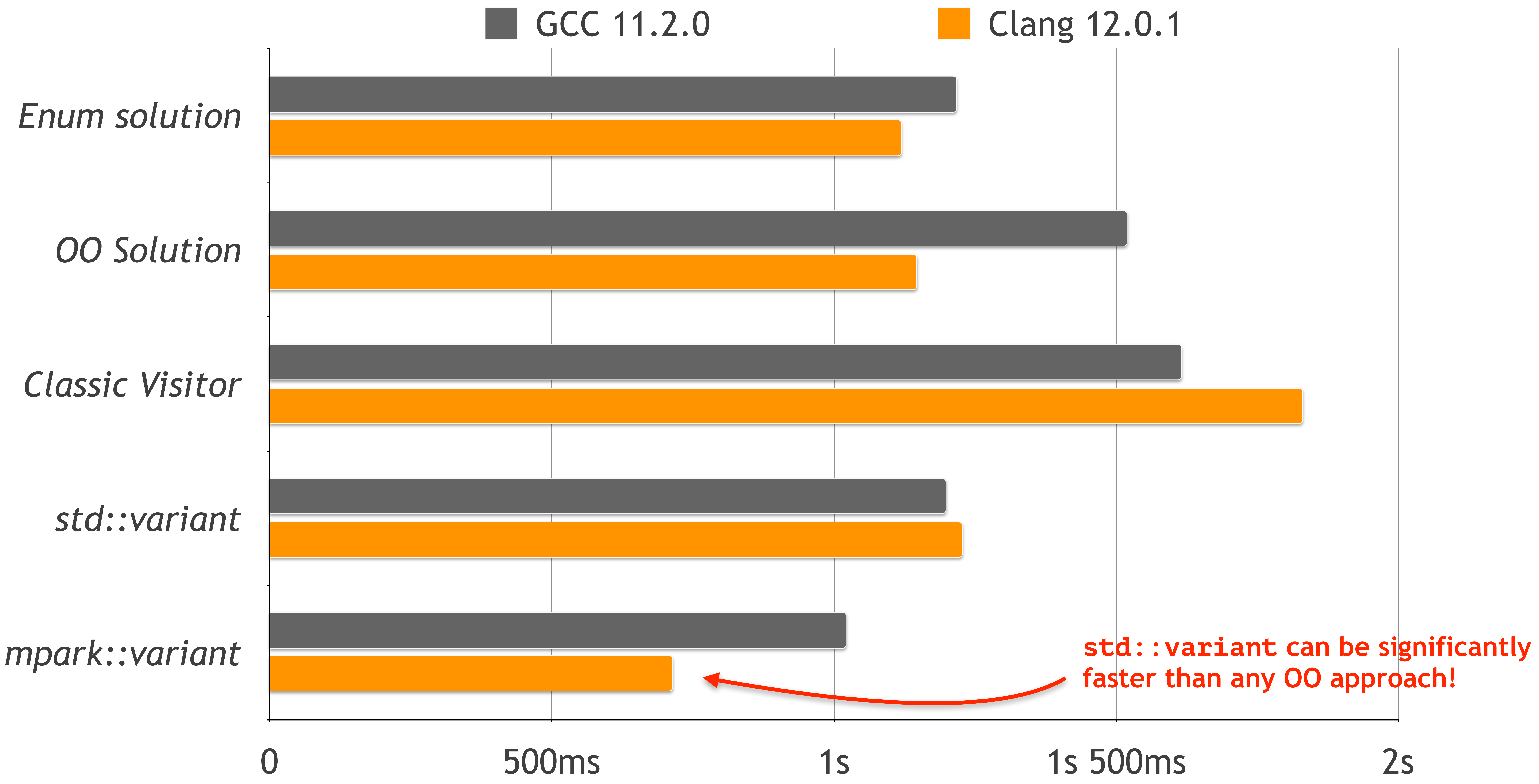
Performance … *sigh*

Do you promise to not take the following results too seriously and as qualitative results only?

Performance Comparison

- Using four different kinds of shape: circles, squares, ellipses and rectangles
- Using 10000 randomly generated shapes
- Performing 25000 `translate()` operations each
- Benchmarks with GCC-11.2.0 and Clang-12.0.1
- 8-core Intel Core i7 with 3.8 Ghz, 64 GB of main memory

Performance Comparison



`std::variant` can be significantly faster than any OO approach!

Why is `std::variant` so fast?

How does `std::visit()` work?

The Secret behind `std::visit()`

Product
Team
Enterprise
Explore
Marketplace
Pricing
Search
/
Sign in
Sign up
Public
Notifications
Fork 75
Star 558
Code
Issues 7
Pull requests 1
Actions
Security
Insights
master
Go to file
Code

mpark / variant

6 branches 8 tags

ax3l GCC<=4.9: Exceptions (#76) ... ✓ 23cb94f on 17 Aug 2021 450 commits

3rdparty	Updated googletest submodule and pointed it to abseil.	4 years ago
cmake	Rename `LICENSE_1_0.txt` to `LICENSE.md`.	5 years ago
include/mpark	GCC<=4.9: Exceptions (#76)	12 months ago
support	Updated Wandbox script.	4 years ago
test	Implement P0608.	3 years ago
.appveyor.yml	Fixed `MPARK_CPP14_CONSTEXPR` for MSVC 19.15.	4 years ago
.clang-format	Implemented initial version of `MPark.Variant`.	6 years ago
.gitignore	Ignored Google Benchmark submodule in `3rdparty/benchmark`.	4 years ago
.gitmodules	Updated googletest submodule and pointed it to abseil.	4 years ago

About

C++17 `std::variant` for C++11/14/17

mpark.github.io/variant

cpp cpp14 cpp11 cpp17 polymorphism discriminated-unions variant cpp20

Readme

BSL-1.0 license

558 stars

30 watching

75 forks

Releases 8

The Secret behind `std::visit()`

```
template <std::size_t B, typename F, typename V, typename... Vs>
MPARK_ALWAYS_INLINE static constexpr R dispatch(
    F &&f, typename ITs::type &&... visited_vs, V &&v, Vs &&... vs)
{
#define MPARK_DISPATCH(I)  \
  // ...

#define MPARK_DEFAULT(I)  \
  // ...

  switch (v.index()) {
    case B + 0: return MPARK_DISPATCH(B + 0);
    case B + 1: return MPARK_DISPATCH(B + 1);
    case B + 2: return MPARK_DISPATCH(B + 2);
    case B + 3: return MPARK_DISPATCH(B + 3);
    case B + 4: return MPARK_DISPATCH(B + 4);
    case B + 5: return MPARK_DISPATCH(B + 5);
    case B + 6: return MPARK_DISPATCH(B + 6);
    case B + 7: return MPARK_DISPATCH(B + 7);
    case B + 8: return MPARK_DISPATCH(B + 8);
    case B + 9: return MPARK_DISPATCH(B + 9);
    case B + 10: return MPARK_DISPATCH(B + 10);
    case B + 11: return MPARK_DISPATCH(B + 11);
    case B + 12: return MPARK_DISPATCH(B + 12);
    case B + 13: return MPARK_DISPATCH(B + 13);
    case B + 14: return MPARK_DISPATCH(B + 14);
    case B + 15: return MPARK_DISPATCH(B + 15);
    case B + 16: return MPARK_DISPATCH(B + 16);
    case B + 17: return MPARK_DISPATCH(B + 17);
    case B + 18: return MPARK_DISPATCH(B + 18);
```

- Dispatch may be based on `switch`
- “Good old” procedural programming
- … which is generated,
- … poses no maintenance issue, and
- … and can be significantly faster.

Amazing, isn’t it?

😍 🤩 🥳

Comparison of Visitor Implementations

Classic Visitor	Modern Visitor with `std::variant`
Intrusive (base class)	**Non-intrusive** (can be added on-the-fly)
Reference-semantics (based on references/pointers)	**Value-semantics** (based on values)
OOP style	**Procedural style**
Slow (many virtual functions, scattered memory access)	**Fast** (no virtual functions, contiguous memory access)

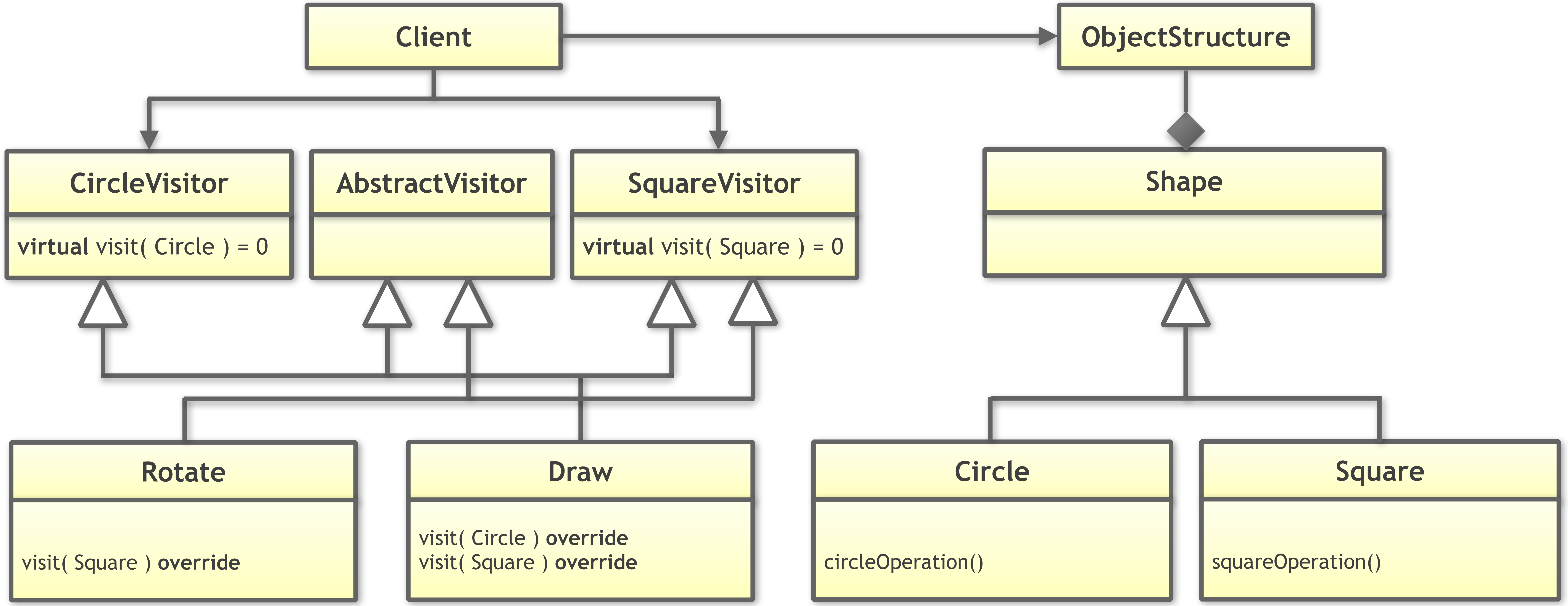
Potential Disadvantages of `std::variant`

- Use alternatives of approximately the same size
  - Revert to pointers (with a performance disadvantage)
  - Use the Proxy design pattern
  - Use the Bridge design pattern
- Be aware that std::variant reveals a lot of information (dependencies!)
  - Revert to pointers (with a performance disadvantage)

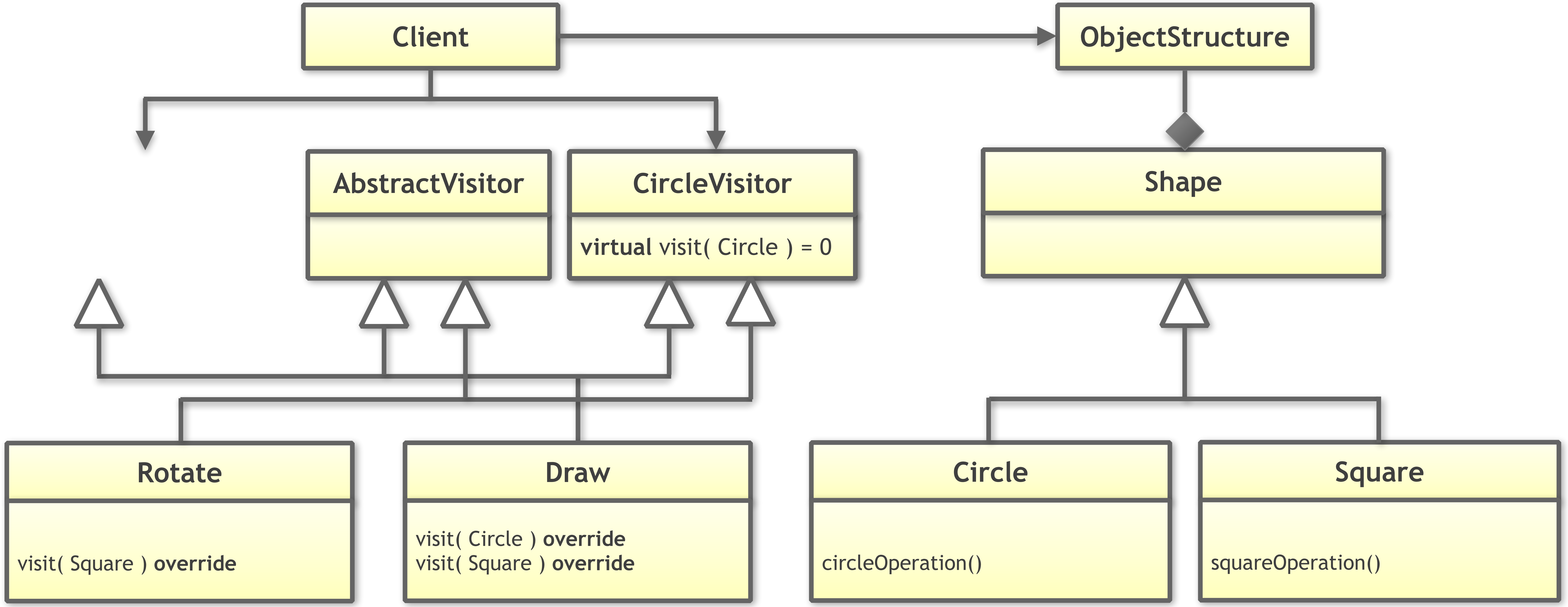
In Dynamic Polymorphism, you have to choose between adding types or operations.

Do I always have to choose between adding types or operations?

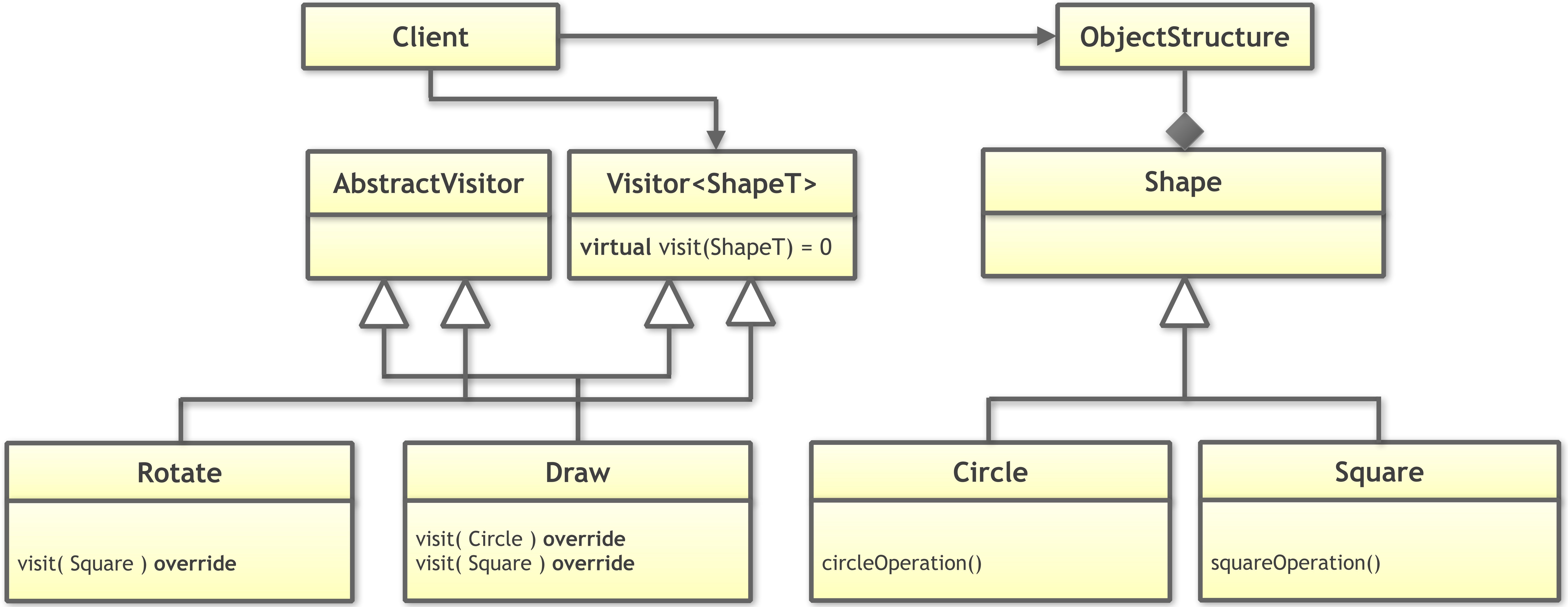
The Acyclic Visitor Design Pattern



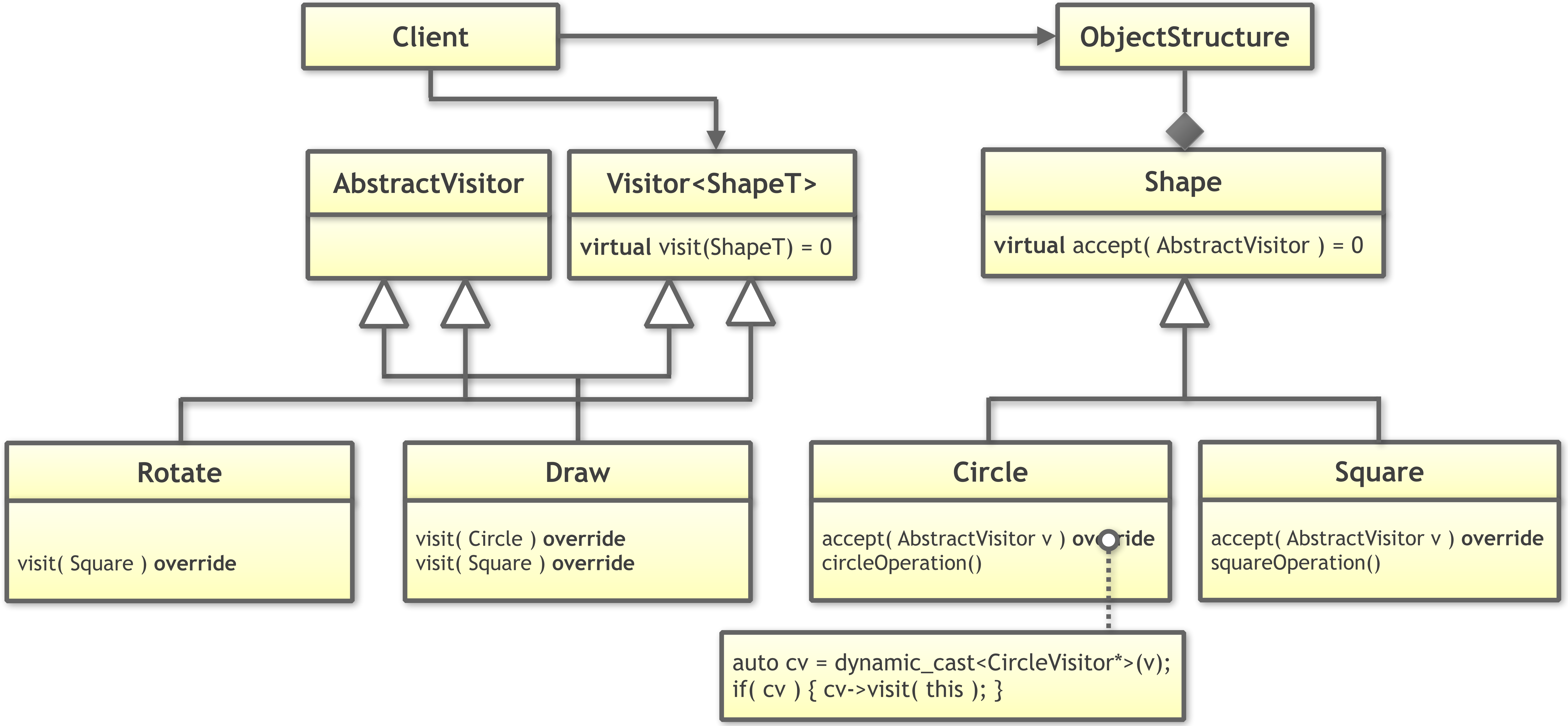
The Acyclic Visitor Design Pattern



The Acyclic Visitor Design Pattern

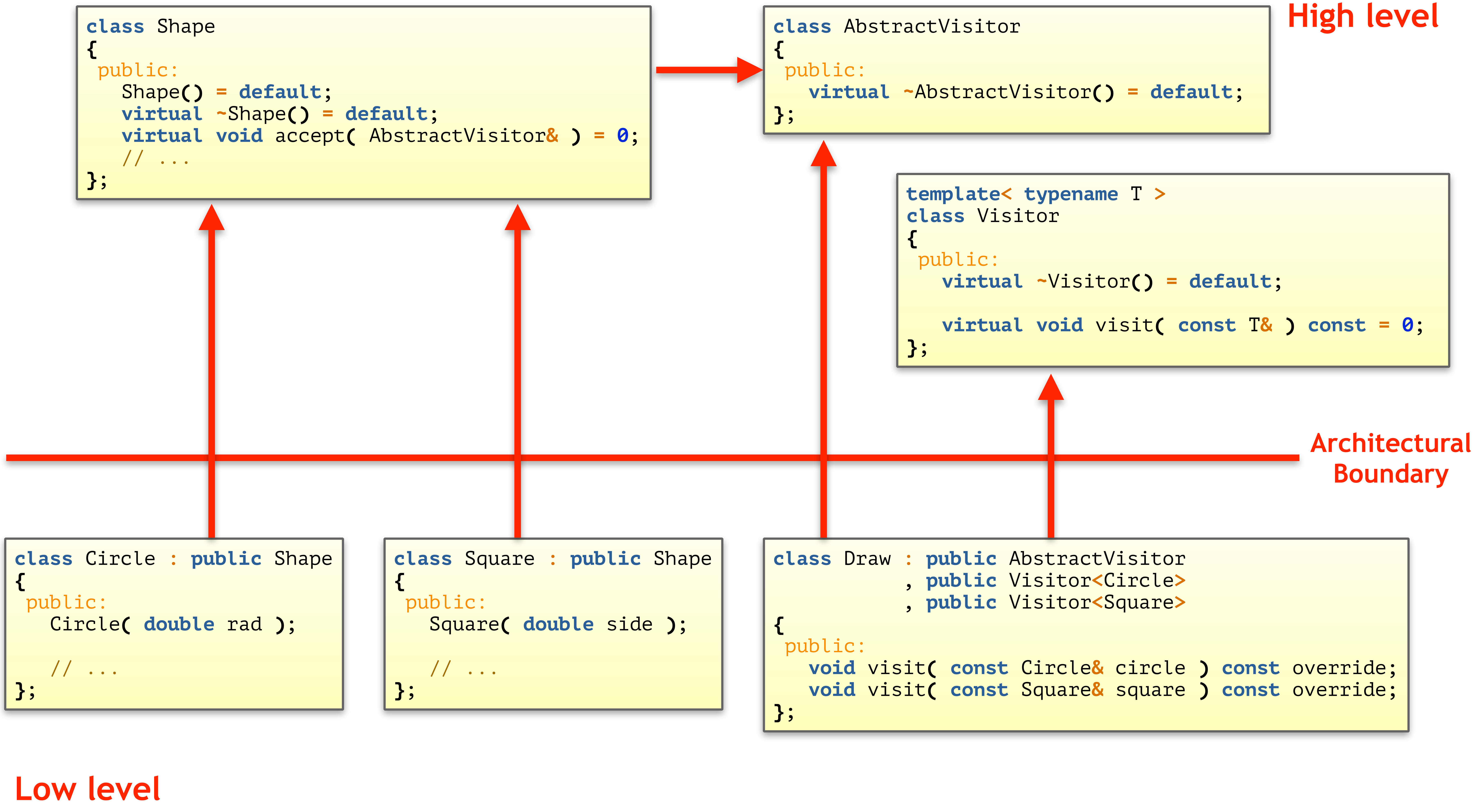


The Acyclic Visitor Design Pattern



```
auto cv = dynamic_cast<CircleVisitor*>(v);
if( cv ) { cv->visit( this ); }
```

The Acyclic Visitor Design Pattern



```
class Shape
{
 public:
   Shape() = default;
   virtual ~Shape() = default;
   virtual void accept( AbstractVisitor& ) = 0;
   // ...
};
```

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

High level

```
template< typename T >
class Visitor
{
 public:
   virtual ~Visitor() = default;

   virtual void visit( const T& ) const = 0;
};
```

Architectural Boundary

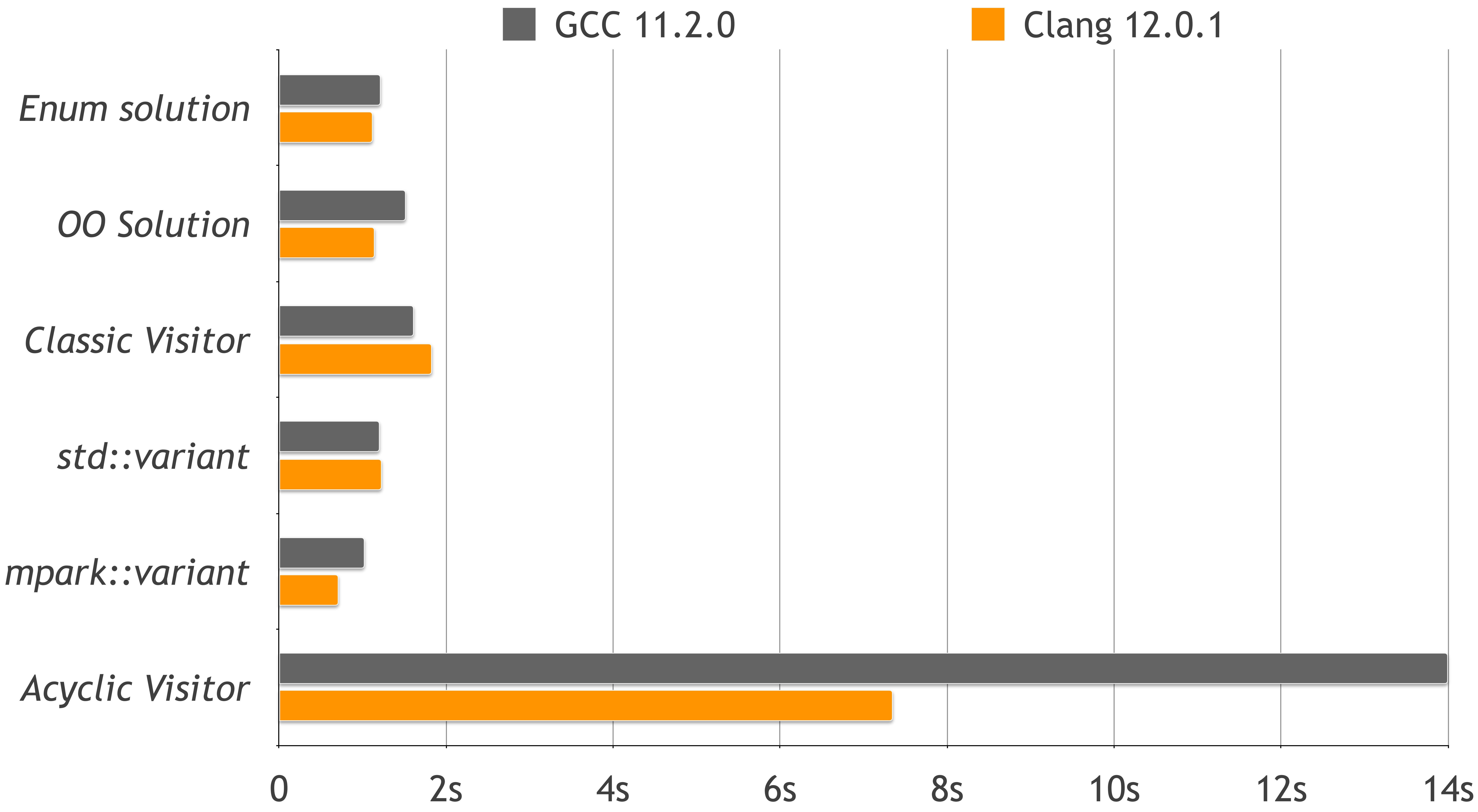
```
class Circle : public Shape
{
 public:
   Circle( double rad );

   // ...
};
```

```
class Square : public Shape
{
 public:
   Square( double side );

   // ...
};
```

```
class Draw : public AbstractVisitor
           , public Visitor<Circle>
           , public Visitor<Square>
{
 public:
   void visit( const Circle& circle ) const override;
   void visit( const Square& square ) const override;
};
```

Low level

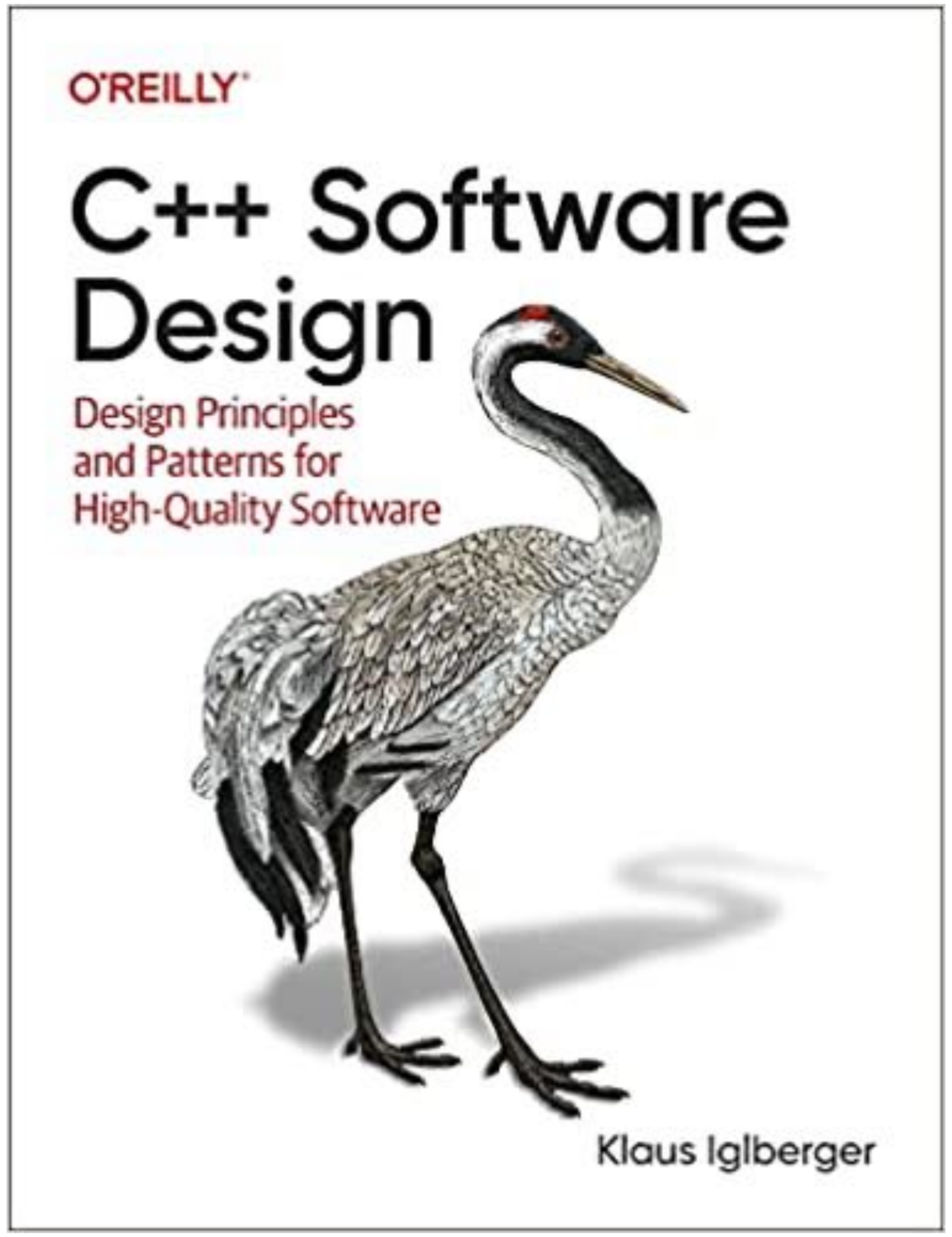
Performance Comparison



Summary

- The Visitor design pattern is the right choice if you want to add operations.
- The Visitor design pattern is the wrong choice if you want to add types.
- Prefer the value-semantics based implementation based on `std::variant`.
- Beware the performance of Acyclic Visitors.

Book Reference



O'REILLY

C++ Software Design

Design Principles and Patterns for High-Quality Software

Klaus Iglberger

www.oreilly.com

Breaking Dependencies

The Visitor Design Pattern

KLAUS IGLBERGER

Cppcon
The C++ Conference

2022
September 12th-16th