

C++ Software Design @ O'Reilly

# 2. C++ Software Design

---

Klaus Iglberger  
January, 23th-24th, 2023

# Content

---

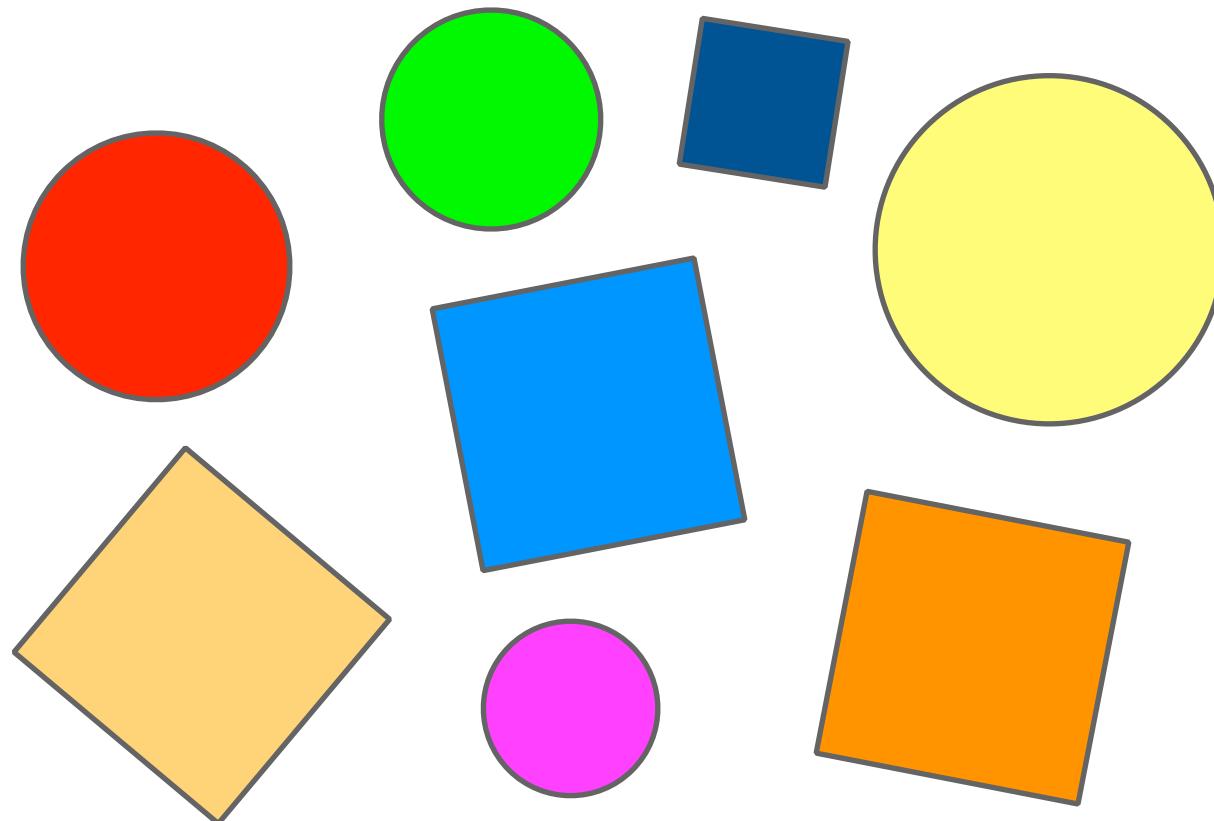
1. Motivation
2. Design Pattern Overview
3. Visitor
4. Strategy
5. Policy-Based Design
6. Interlude
7. Prototype
8. Adapter
9. Bridge
10. External Polymorphism
11. Type Erasure

## 2.1. Motivation

---

# Our Toy Problem: Drawing Shapes

---



# An Example

---

**Task (2\_Modern\_Cpp\_Design\_Patterns/Procedural):** Evaluate the given design with respect to changeability and extensibility.

# 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

```
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

```
};

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 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() )  
        {  
            case Circle:  
                draw( *s );  
                break;  
            case Square:  
                draw( *s );  
                break;  
            case Triangle:  
                draw( *s );  
                break;  
            default:  
                break;  
        }  
    }  
}
```

# A Procedural Solution

```
};

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 ) );
}
```

# A Procedural Solution

```
    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 );
}
```

# 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

```
};

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 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 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 ) );
```

# The Expert's Advice

---

*"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 Problem

---

There is one constant in software development and that is ...

# Change

# The Problem

---

The truth in our industry:

**Software must be  
adaptable to frequent  
changes**

# The Problem

---

The truth in our industry:

**Software must be  
adaptable to frequent  
changes**

# The Problem

---

What is the core problem of adaptable software  
and software development in general?

# Dependencies

## The Problem

---

# Dependencies ...

- ... complicate **changes/modifications**
- ... impede the **testability** of software
  - ... obstruct **modularity**
  - ... increase **build times**

# The Expert's Opinion

---

*"Dependency is the key problem in software development at all scales."*

*(Kent Beck, TDD by Example)*

# Guidelines

---

**Guideline:** When designing software (modules, classes, functions, ...) try to minimize coupling between software components.

# The SOLID Principles

---

**Single-Responsibility Principle (SRP)**

**Open-Closed Principle (OCP)**

**Liskov Substitution Principle (LSP)**

**Interface Segregation Principle (ISP)**

**Dependency Inversion Principle (DIP)**

# The SOLID Principles

---

**S**ingle-Responsibility Principle (SRP)

**O**pen-Closed Principle (OCP)

**L**iskov Substitution Principle (LSP)

**D**ependency Inversion Principle (DIP)

**I**nterface Segregation Principle (ISP)



**Robert C. Martin**

# The SOLID Principles

---

**Single-Responsibility Principle (SRP)**

**Open-Closed Principle (OCP)**

**Liskov Substitution Principle (LSP)**

**Interface Segregation Principle (ISP)**

**Dependency Inversion Principle (DIP)**



**Robert C. Martin**

**Michael Feathers**

# An Example

---

**Task (2\_Modern\_Cpp\_Design\_Patterns/ObjectOriented):** Evaluate the given design with respect to changeability and extensibility.

# An Example

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

    virtual void translate( Vector2D const& ) = 0;
    virtual void rotate( double const& ) = 0;
    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 translate( Vector2D const& ) override;
    void rotate( double const& ) override;
    void draw() const override;

    // ...

private:
    double radius;
```

# An Example

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

    virtual void translate( Vector2D const& ) = 0;
    virtual void rotate( double const& ) = 0;
    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 translate( Vector2D const& ) override;
    void rotate( double const& ) override;
    void draw() const override;

    // ...

private:
    double radius;
```

# An Example

```
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 translate( Vector2D const& ) override;  
    void rotate( double const& ) override;  
    void draw() const override;  
  
    // ...  
  
private:  
    double radius;  
    // ... Remaining data members  
};  
  
class Square : public Shape  
{  
public:  
    explicit Square( double s )  
        : side{ s }  
        , // ... Remaining data members  
    {}  
};
```

# An Example

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

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

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

    void translate( Vector2D const& ) override;
    void rotate( double const& ) override;
    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 Example

```
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();
    }
}

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 Example

```
{  
    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 Example

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

    virtual void translate( Vector2D const& ) = 0;
    virtual void rotate( double const& ) = 0;
    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 translate( Vector2D const& ) override;
    void rotate( double const& ) override;
    void draw() const override;

    // ...

private:
    double radius;
```

## 2.2. Design Pattern Overview

---

# What is Software Design/Architecture?

---

# What is Software Design/Architecture?

---

*“..., I’ll assert that there is no difference between [architecture and design]. None at all.”*

*(Robert C. Martin, Clean Architecture)*

# What is Software Design/Architecture?

---

*"The goal of software architecture is to minimise the human resources required to build and maintain the required system."*

*(Robert C. Martin, Clean Architecture)*

# What is Software Design/Architecture?

---

**Software Design** is the art of managing interdependencies between software components.  
It aims at minimizing (technical) **dependencies** and introduces the necessary **abstractions** and compromises.

*(Klaus Iglberger)*

# What is Software Design/Architecture?

---

**Software Design** is the art of managing  
dependencies and abstractions.

# What is a Design Pattern?

---

A design pattern ...

- ... has a **name**;
- ... has an **intent**;
- ... aims at reducing **dependencies**;
- ... provides some sort of **abstraction**;
- ... has been **proven to work over the years**.

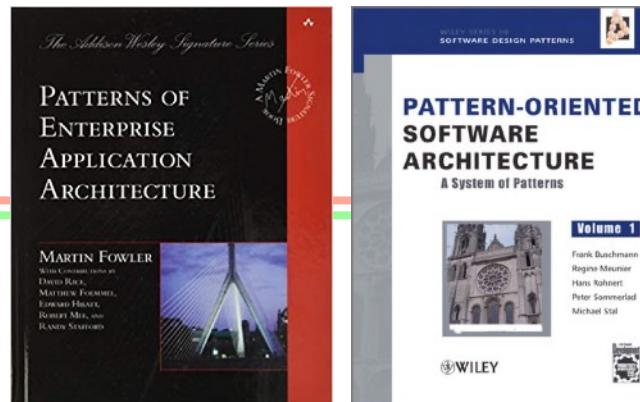
A design pattern is **not** ...

- ... limited to **object-oriented programming**;
- ... limited to **dynamic polymorphism**;
- ... a **language specific idiom**.

## 2. C++ Software Design - Design Pattern Overview

### Software Architecture

- ⦿ How are big entities depending on each other?
- ⦿ Design decisions that are harder to change
- ⦿ Architectural patterns
- ⦿ Examples:
  - ⦿ Client-Server Architecture
  - ⦿ Microservices
  - ⦿ MVC, ...



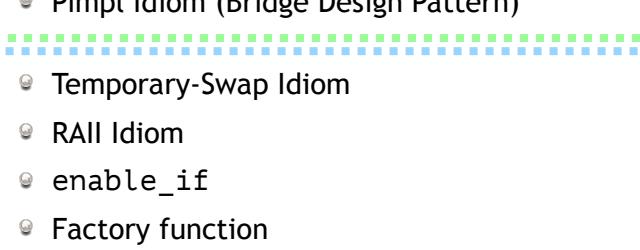
### Software Design

- ⦿ How are small entities depending on each other?
- ⦿ Design decisions that are easier to change
- ⦿ Design patterns
- ⦿ Examples:
  - ⦿ GoF Patterns: Visitor, Strategy, Observer, ...
  - ⦿ External Polymorphism
  - ⦿ ...



### Implementation Details

- ⦿ How is a design implemented?
- ⦿ Which features are used?
- ⦿ Implementation patterns
- ⦿ Examples:
  - ⦿ new, malloc, ...
  - ⦿ Exception Safety, Performance, ...
  - ⦿ ...



# The Value of a Name



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.”

# The Value of a Name



Me

“I think we should create a system that allows us to extend the operations without the need to modify existing types again and again.”



You

“I don’t know. Rather than new operations. I would expect new types to be added frequently. So I prefer a solution that allows me to add types easily. But in order to reduce coupling to the implementation details, which is to be expected, I would suggest a way to extract implementation details from existing types by introducing a variation point.”



Me

“Yes, you may have a point there. But since we’ll have to extend operations fairly often, we probably should consider designing the system in such a way that we can build on and reuse a given implementation easily.”

# The Attitude Toward Design Patterns

---



1 month ago

Really? Design Patterns in 2021?



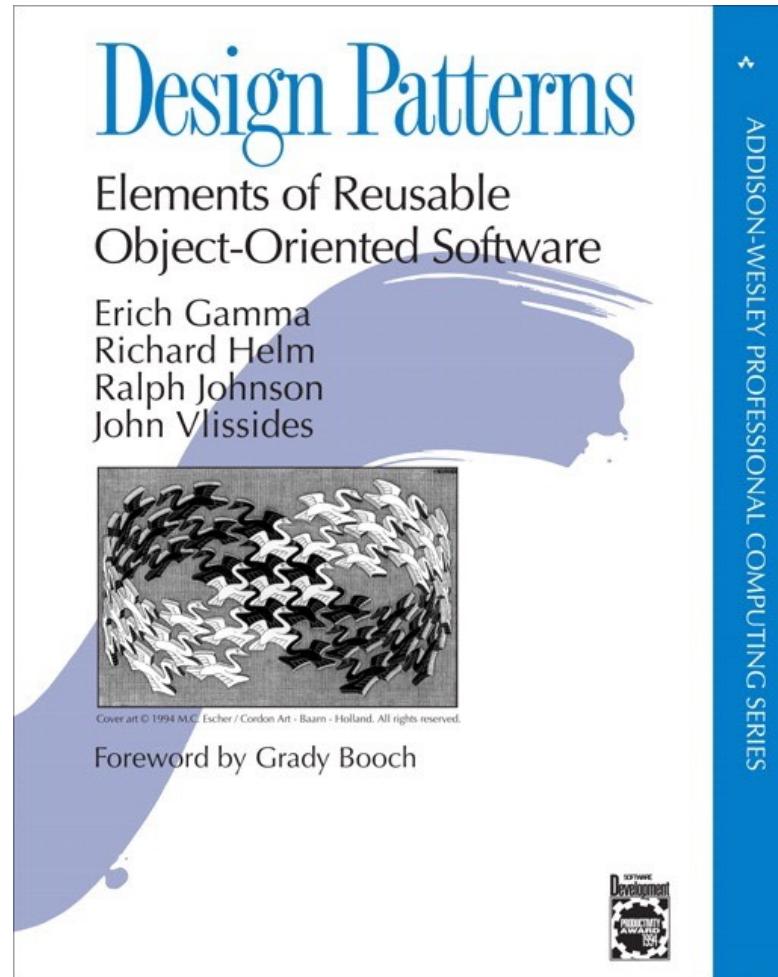
REPLY

# Guidelines

---

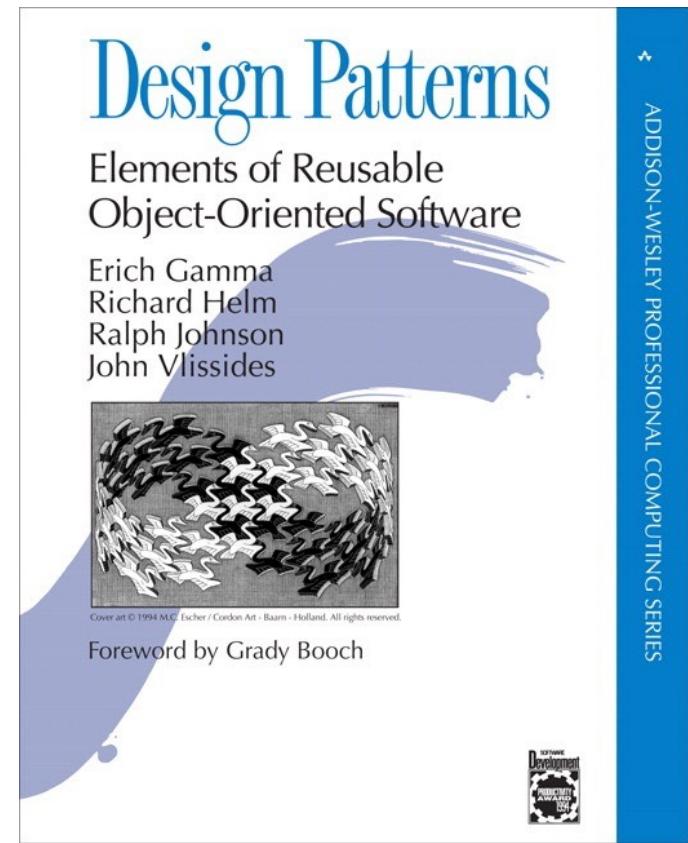
**Guideline:** Design Patterns are Everywhere!

# Terminology



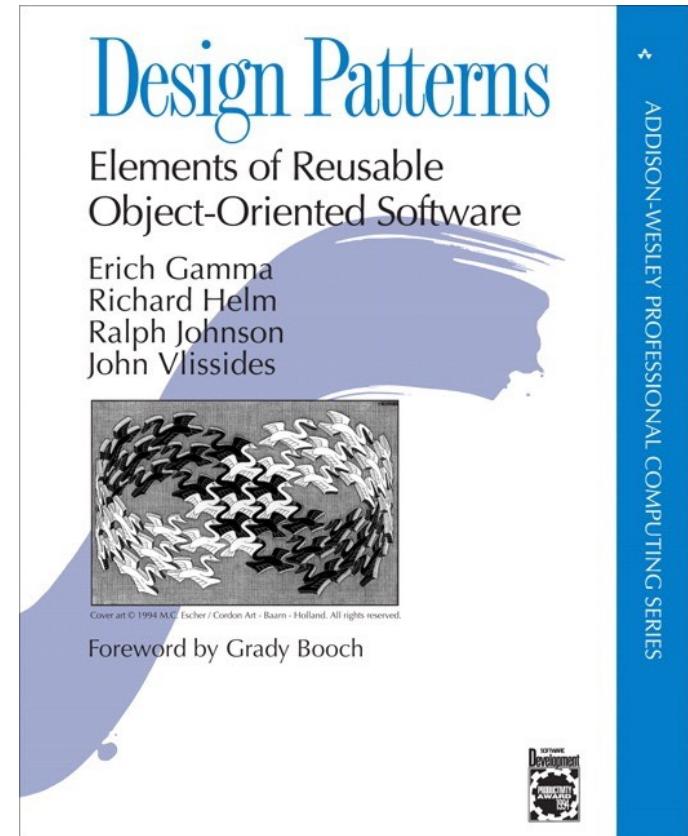
# GoF Design Patterns

- Creational Patterns (5)
- Structural Patterns (7)
- Behavioral Patterns (11)



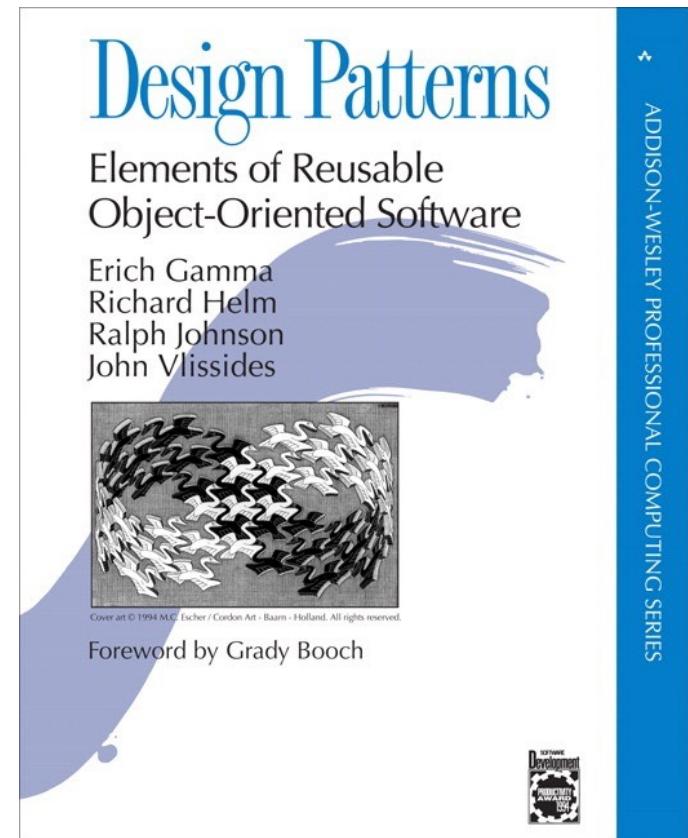
# Creational Patterns

- Abstract Factory
- Builder
- Factory Method
- Prototype
- Singleton



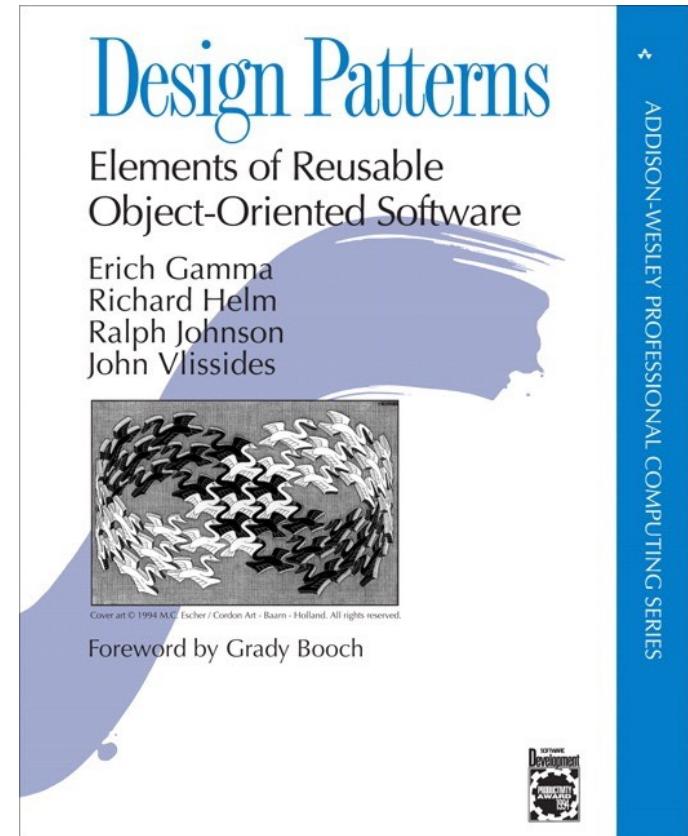
# Creational Patterns

- Abstract Factory
- Builder
- Factory Method
- Prototype
- Singleton



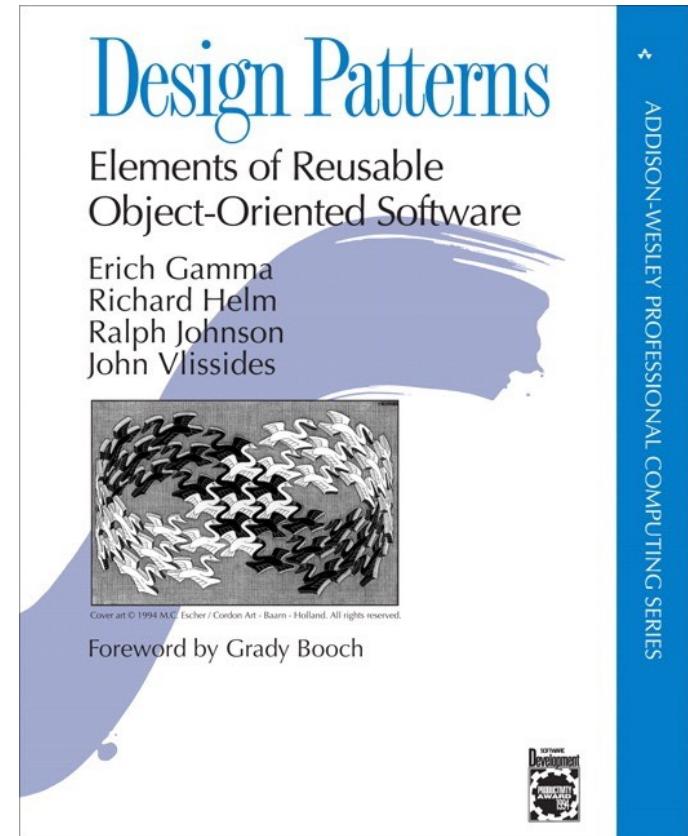
# Structural Patterns

- Adaptor
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy



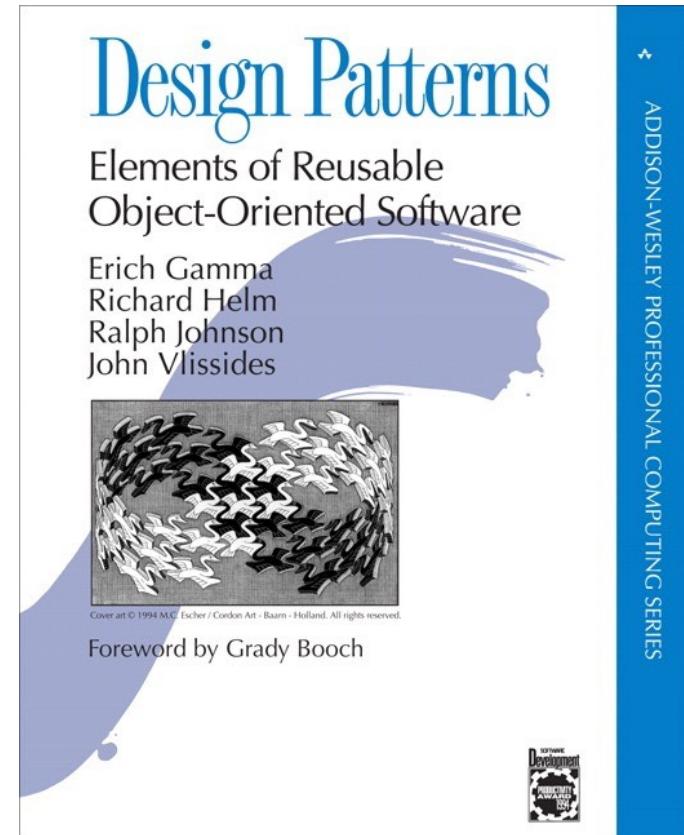
# Structural Patterns

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy



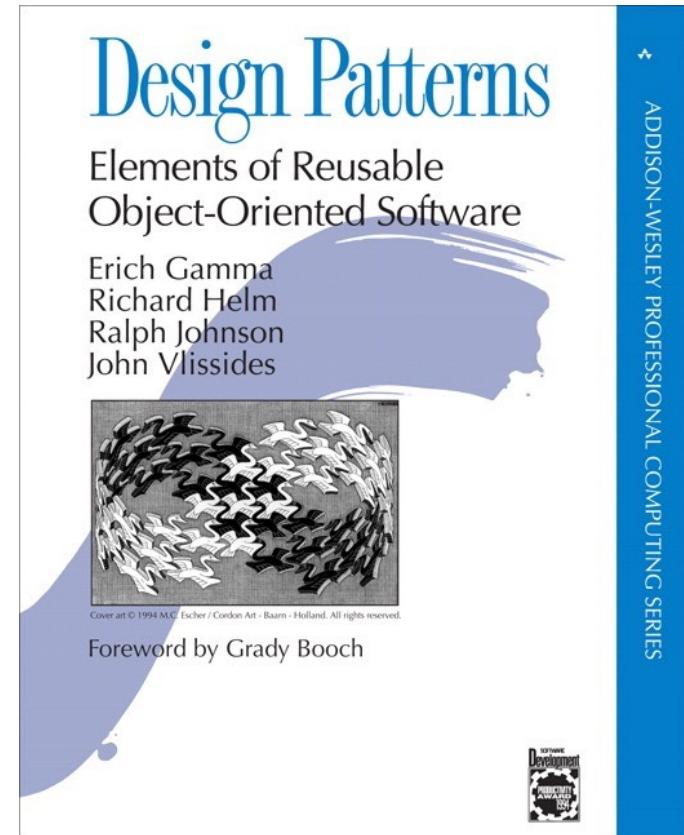
# Behavioral Patterns

- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor



# Behavioral Patterns

- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor



# Modern C++ Design Patterns

---

- Policy-Based Design
- CRTP
- Expression Templates
- Type Erasure

## 2.3. Visitor

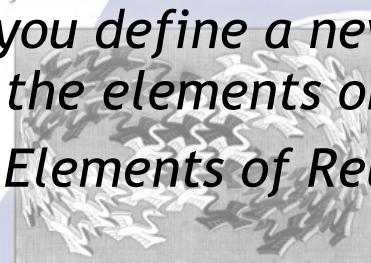
---

# The Intent

## Design Patterns

Elements of Reusable  
Object-Oriented Software

Erich Gamma  
Richard Helm  
Ralph Johnson  
John Vlissides



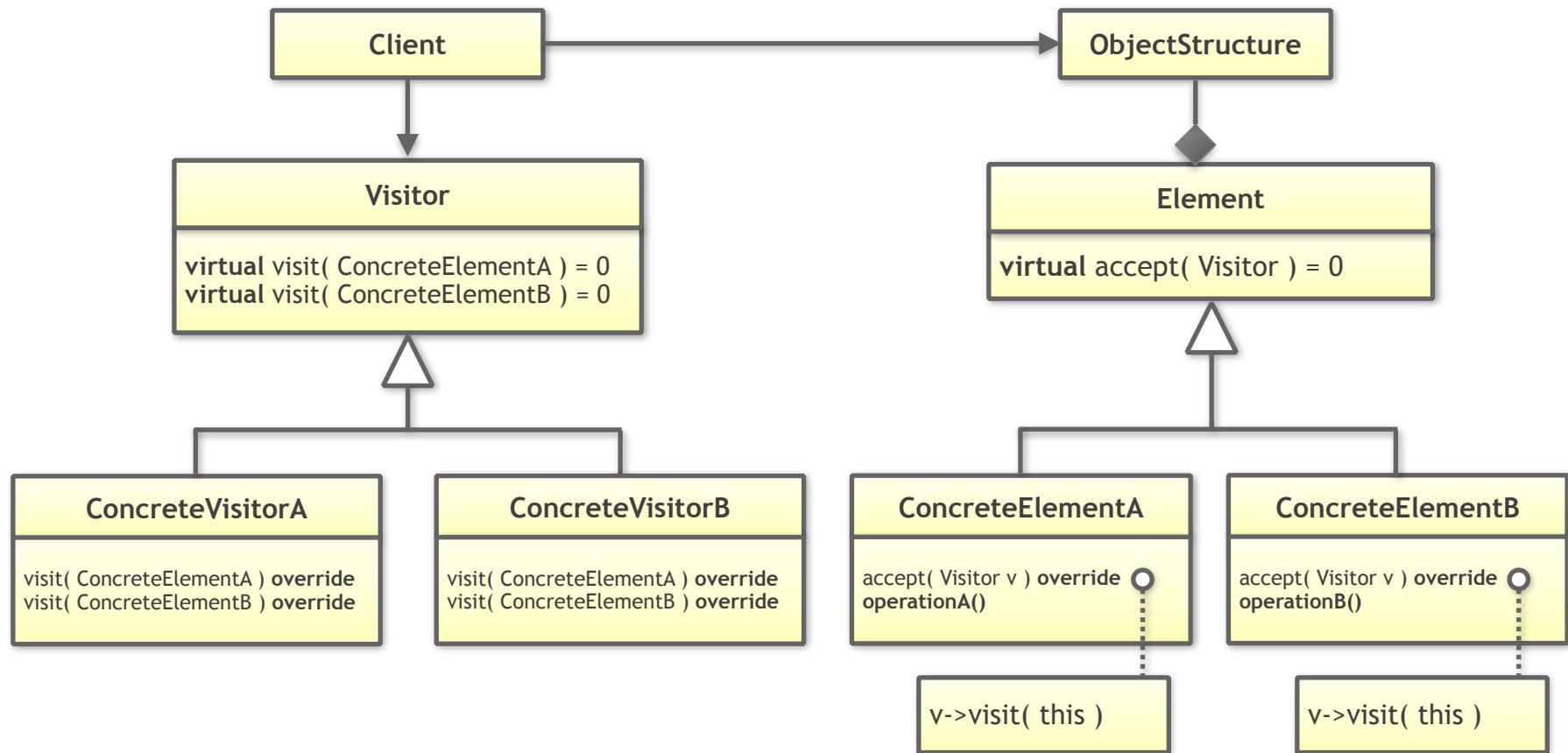
Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

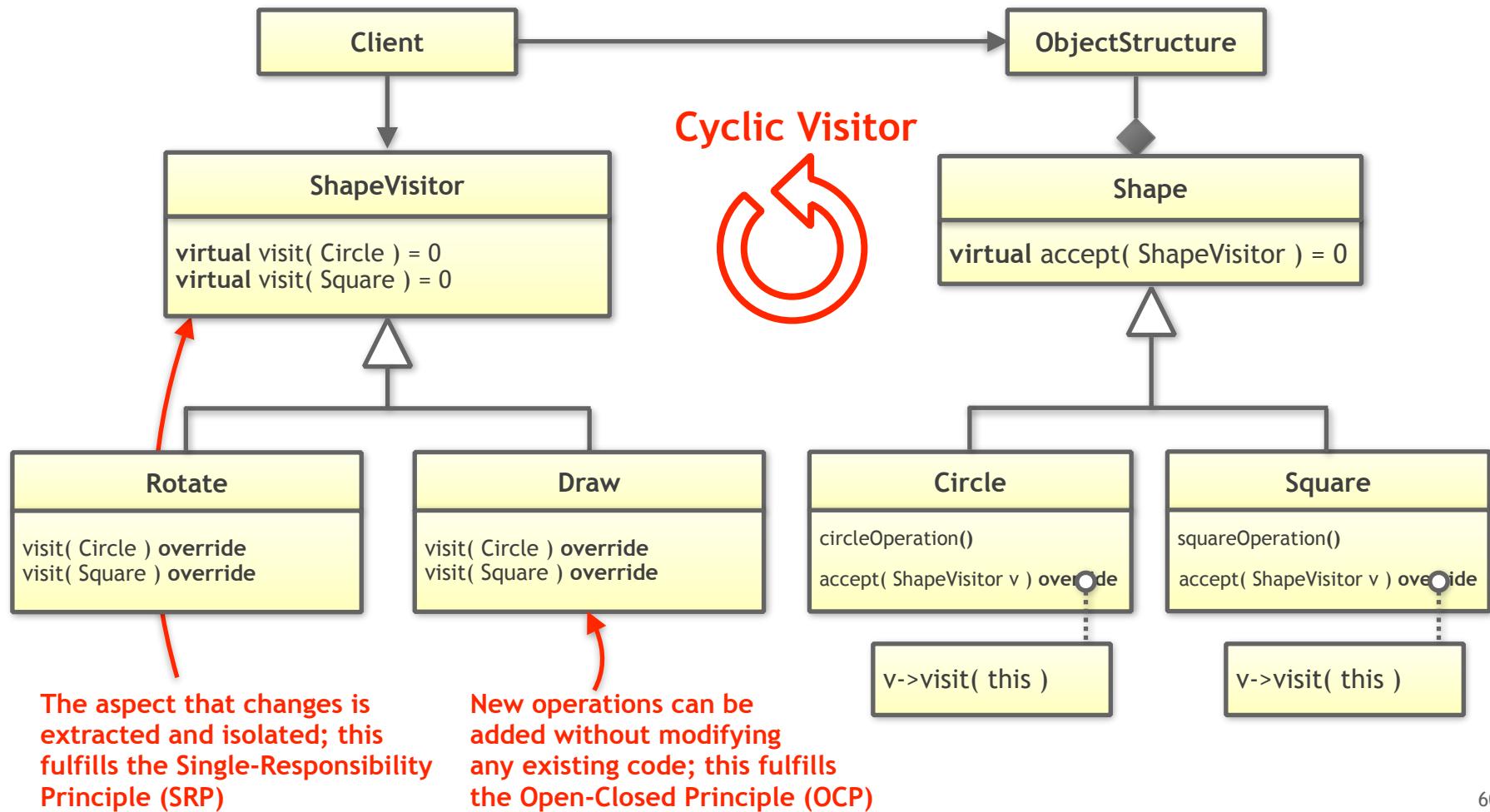
*"Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates."*

*(GoF, Design Patterns: Elements of Reusable Object-Oriented Software)*

# The Classic Visitor Design Pattern



# The Classic Visitor Design Pattern



# The Classic Visitor Design Pattern

---

**Task ([C\\_Modern\\_Cpp\\_Design\\_Patterns/Visitor](#)):** Refactor the classic Visitor solution by a value semantics based solution. Note that the general behavior should remain unchanged.

# 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 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

```
// ... 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

```
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 ) );
}
```

# 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 );  
}
```

# 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;
```

# 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;
```

# A “Modern C++” Solution

```
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 override;  
    void operator()( Square const& ) const override;  
};  
  
using Shape = std::variant<Circle,Square>;
```

# A “Modern C++” Solution

```
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 “Modern C++” Solution

```
{  
    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 );  
}
```

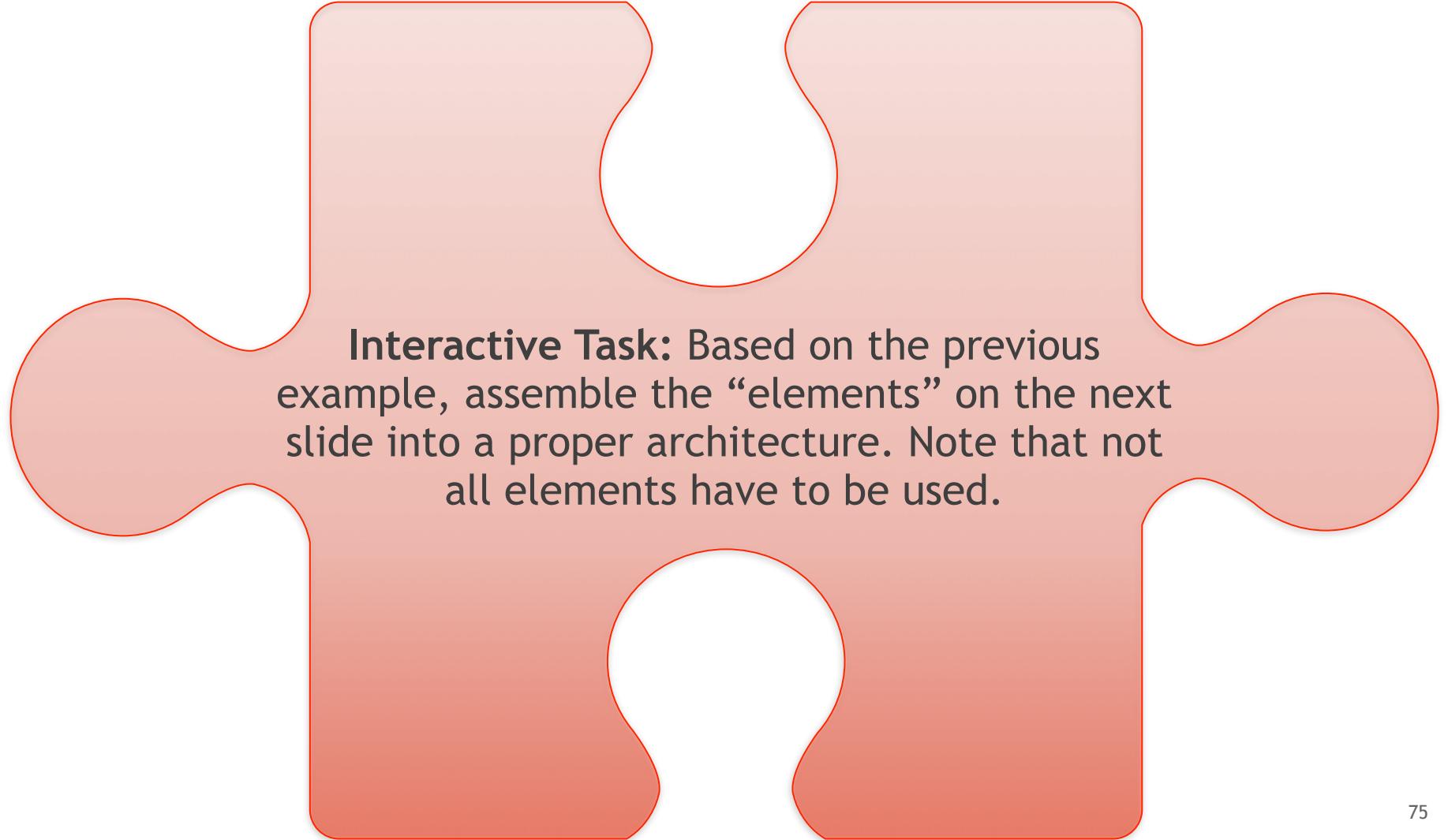
# std::variant - Implementation Details

```
template< typename T, typename V >
constexpr auto make_func() {
    return + []( const char* b, V v ) {
        const auto& x = *reinterpret_cast<const T*>(b);
        v(x);
    };
}

template< typename... Ts, typename V >
void foo( std::size_t i, const char* b, V v ) {
    static constexpr std::array<void(*)(const char*, V v ),  

        sizeof...(Ts)> table = { make_func<Ts,V>()... };
    table[i](b,v);
}
```

# An Architectural Puzzle



**Interactive Task:** Based on the previous example, assemble the “elements” on the next slide into a proper architecture. Note that not all elements have to be used.

# An Architectural Puzzle

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

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

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

```
class ShapeVisitor
{
public:
    virtual ~ShapeVisitor() {}

    virtual void draw( const Circle& circle ) const = 0;
    virtual void draw( const Square& square ) const = 0;
};
```

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

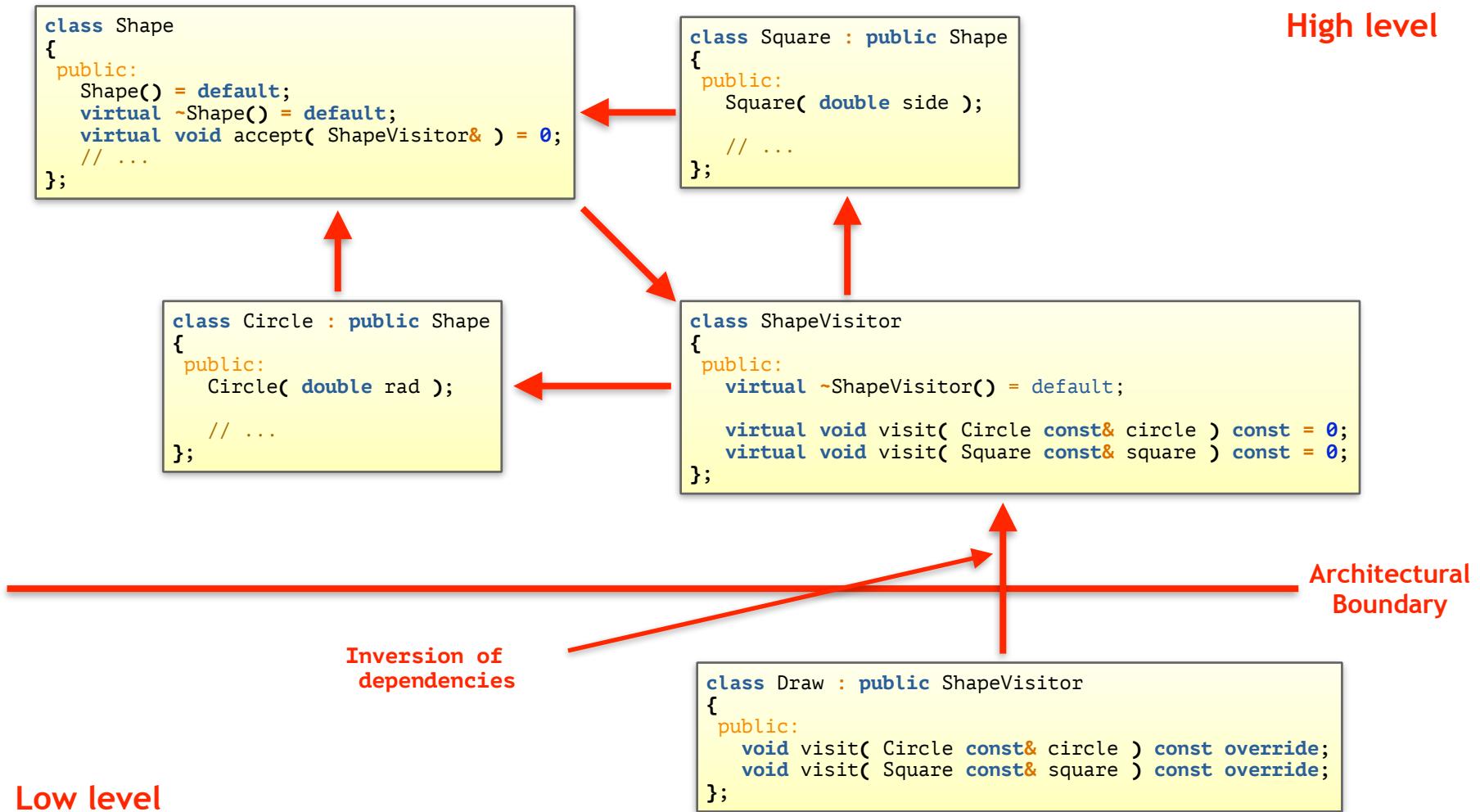
Architectural  
Boundary

Low level



High level

# An Architectural Puzzle (Solution)



# Comparison with std::variant

High level

```
class Circle
{
public:
    Circle( double rad );
    // ...
};
```

```
class Square
{
public:
    Square( double side );
    // ...
};
```

Architectural Boundary

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

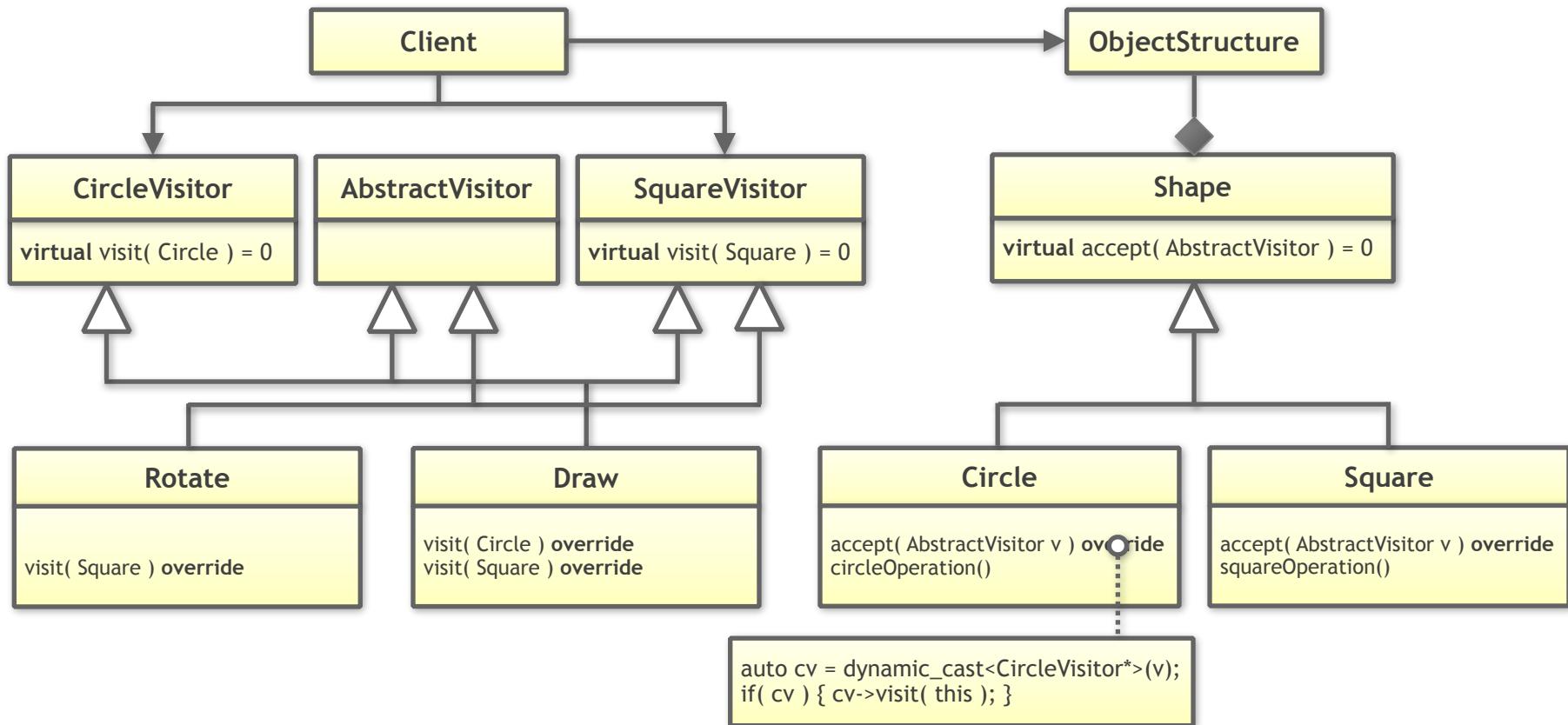
Automatic inversion  
of dependencies

Architectural Boundary

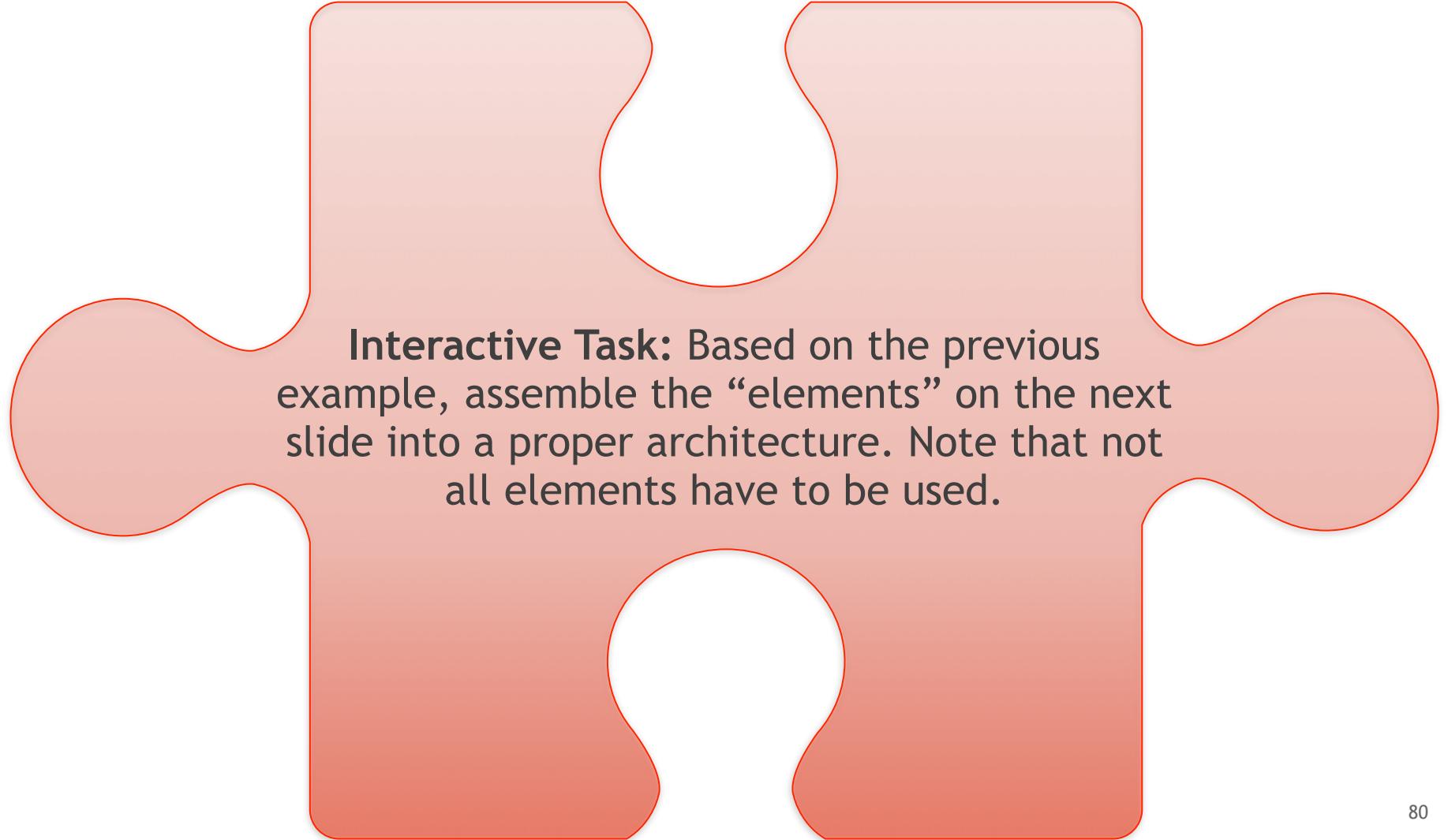
```
class Draw
{
public:
    void operator()( Circle const& circle ) const;
    void operator()( Square const& square ) const;
};
```

Low level

# The Acyclic Visitor Design Pattern



# An Architectural Puzzle



**Interactive Task:** Based on the previous example, assemble the “elements” on the next slide into a proper architecture. Note that not all elements have to be used.

# The Architectural of the Acyclic Visitor

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

```
template< typename T >
class Visitor
{
public:
    virtual ~Visitor() = default;
    virtual void visit( const T& ) const = 0;
};
```

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

```
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;
};
```

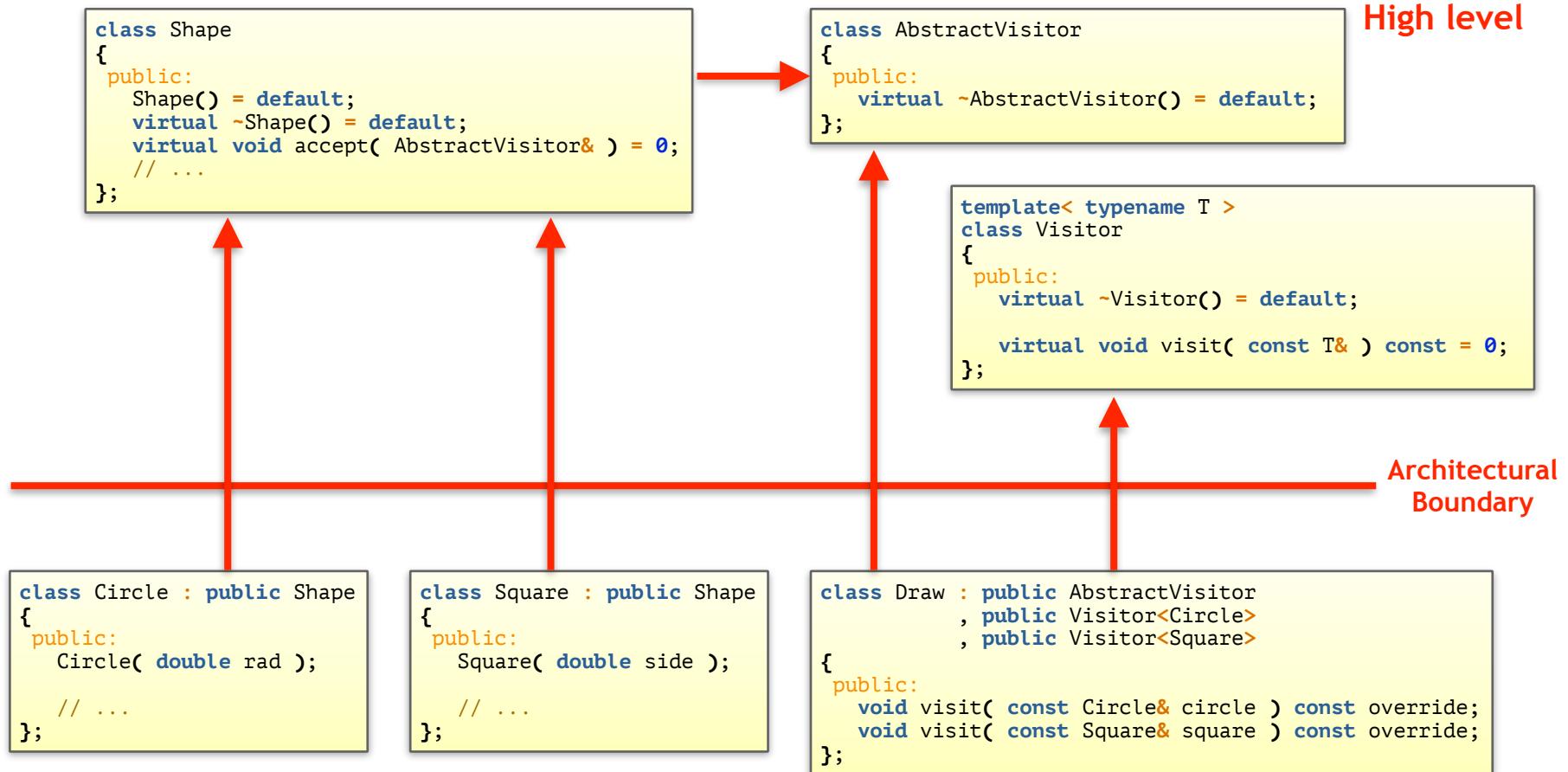
Architectural  
Boundary

Low level



High level

# The Architectural of the Acyclic Visitor



Low level

# An Acyclic Visitor-Based Solution

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

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

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

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

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

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

# An Acyclic Visitor-Based Solution

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

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

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

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

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

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

# An Acyclic Visitor-Based Solution

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

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

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

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

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

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

# An Acyclic Visitor-Based Solution

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

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

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

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

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

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

# An Acyclic Visitor-Based Solution

```
};

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

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

    void accept( const AbstractVisitor& v ) override {
        if( auto cv = dynamic_cast<const Visitor<Circle*>">(&v) ) {
            cv->visit( *this );
        }
    }

    // ...

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

class Square : public Shape
{
public:
```

# An Acyclic Visitor-Based Solution

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

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

    void accept( const AbstractVisitor& v ) override {
        if( auto sv = dynamic_cast<const Visitor<Square>*>(&v) ) {
            sv->visit( *this );
        }
    }

    // ...

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

class Draw : public AbstractVisitor
, public Visitor<Circle>
, public Visitor<Square>
```

# An Acyclic Visitor-Based Solution

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

class Draw : public AbstractVisitor
            , public Visitor<Circle>
            , public Visitor<Square>
{
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 ) );
}
```

# An Acyclic 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

---

The classic Visitor design pattern ...

- ... requires a base class (dependency);
- ... promotes heap allocation;
- ... requires memory management.

Using std::variant instead of the classic Visitor design pattern ...

- ... simplifies code (a lot!);
- ... facilitates comprehension;
- ... reduces dependencies.

# std::variant – Advantages/Disadvantages

---

Use std::variant if ...

- ... you have a **closed set** of known types;
- ... you want to **extend functionality**, not types;
- ... you don't need to abstract from the **concrete types**;
- ... you require **maximum performance**.

Don't use std::variant if ...

- ... you have an **open set** of types;
- ... you want to **extend types**, not functionality;
- ... you want to use the abstraction across **architectural boundaries**;
- ... you want to **hide implementation details** about the alternatives;
- ... you need an **abstraction of the operations** (e.g. via base pointer);
- ... performance is not the **primary concern**.

## 2. C++ Software Design - Visitor

YouTube DE Search

### C++'s evolution priorities

Historical

The slide features two large triangles side-by-side. The left triangle, labeled 'Historical', is blue and oriented downwards, containing the words 'add things', 'fix things', and 'simplify'. The right triangle, labeled 'A future worth considering?', is yellow and oriented upwards, containing the words 'add things', 'fix things', and 'simplify'. Arrows point from 'add things' and 'fix things' in the blue triangle down towards 'simplify' in the yellow triangle.

5

Herb Sutter

De-fragmenting C++:  
Making Exceptions and RTTI  
More Affordable and Usable  
("Simplifying C++" #6 of N)

Video Sponsorship Provided By:

ansatz

CC BY SA

# std::variant – Return Values and Parameters

---

**Task (C\_Modern\_Cpp\_Design\_Patterns/Variant):** Implement the translate() and area() operations for the given Shape variant. Hint: the area of a circle is `radius*radius*M_PI`, the area of a square is `side*side`.

# Guidelines

---

**Guideline:** Use the Visitor design pattern to enable the extension of operations for a closed set of types.

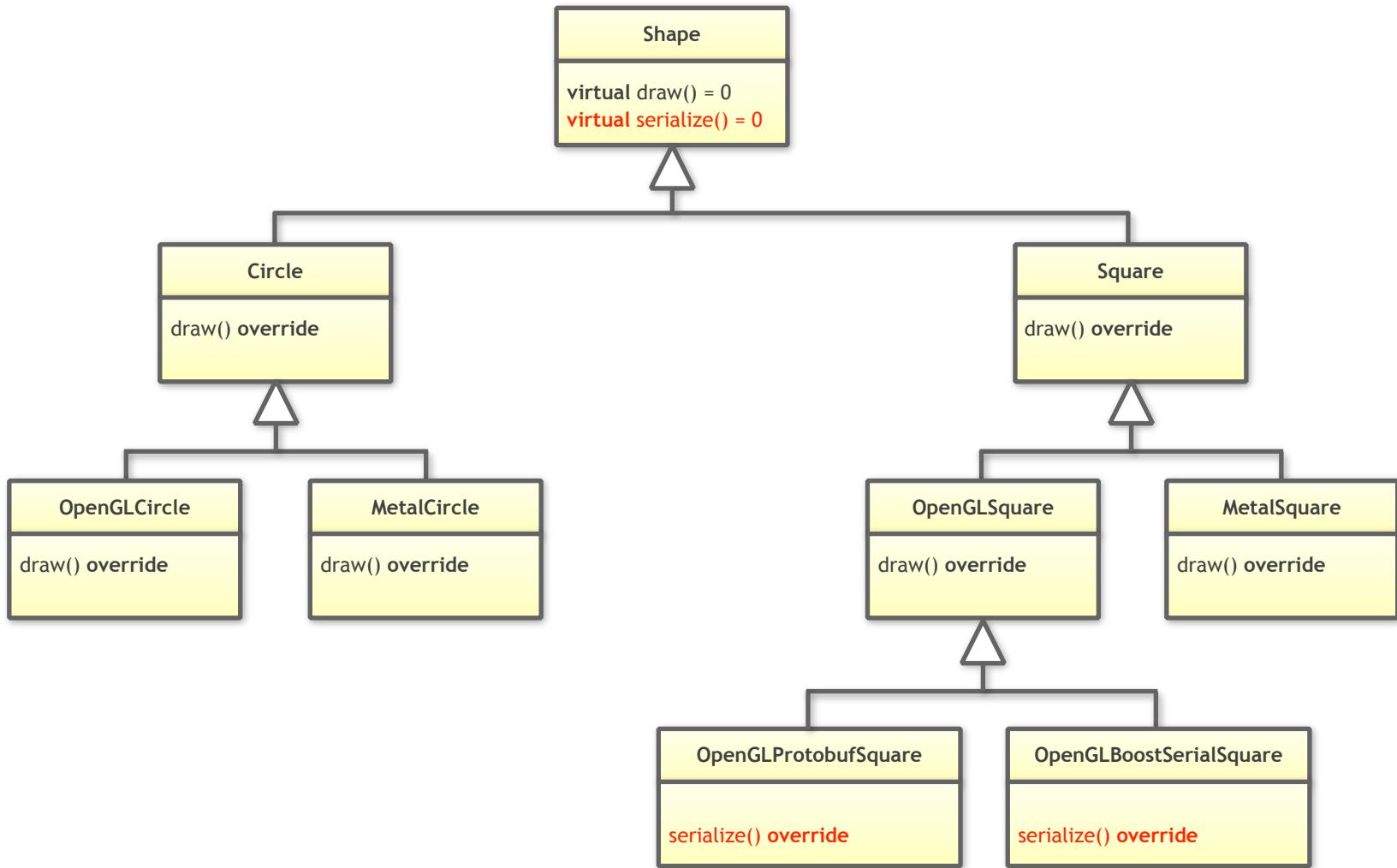
**Guideline:** Avoid the over-/abuse of inheritance

**Guideline:** Prefer multi-paradigm solutions.

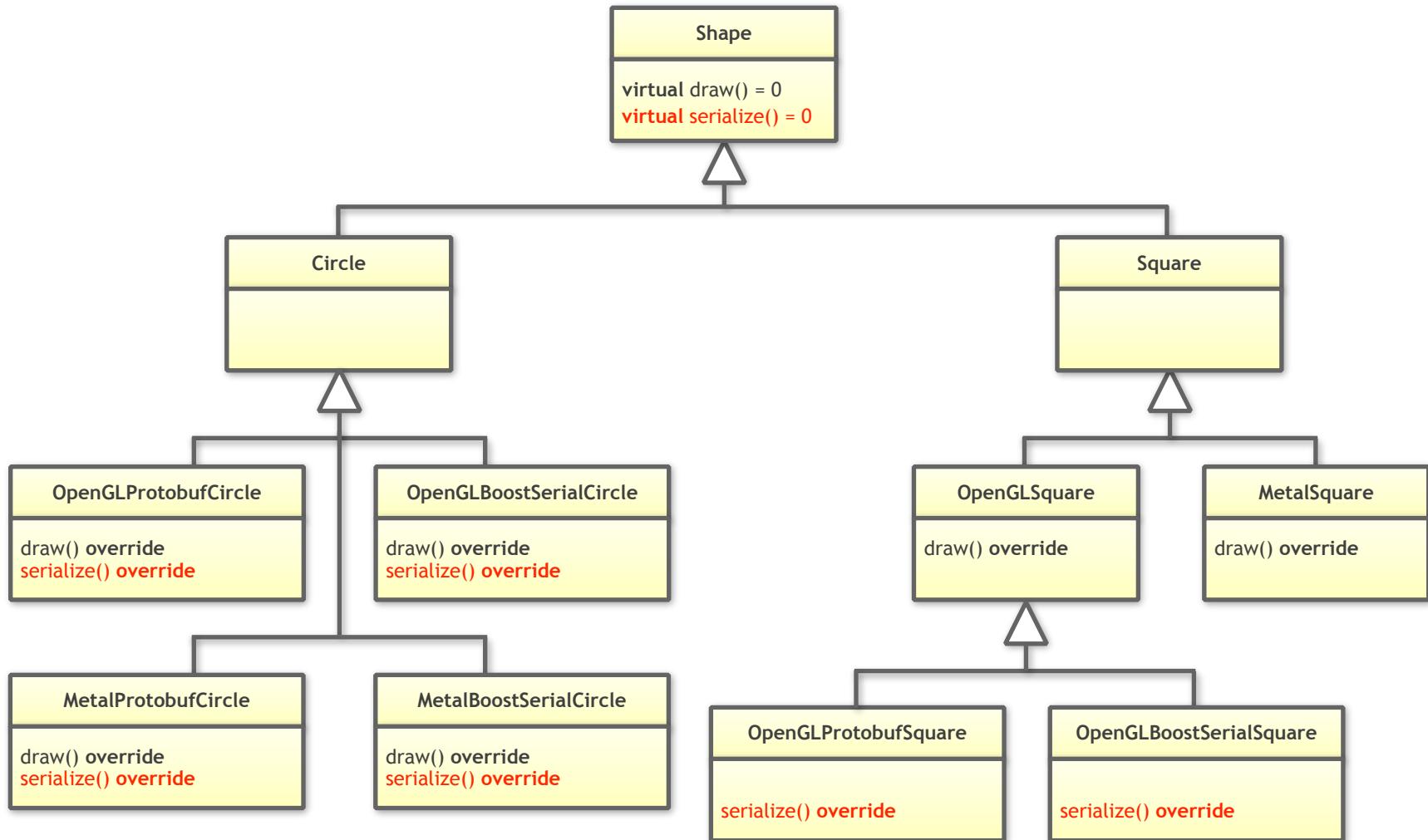
## 2.4. Strategy

---

# A Design Challenge



# A Design Challenge



# A Design Challenge

---

```
class OpenGLProtobufCircle : public Circle
{
public:
    // ...
    virtual void draw( Screen& s, /*...*/ ) const;
    virtual void serialize( ByteStream& bs, /*...*/ ) const;
    // ...
};
```

These functions easily result in coupling/dependencies. How do we extract the logic?

Using inheritance to solve our problem easily leads to ...

- ⌚ ... many derived classes;
- ⌚ ... ridiculous class names;
- ⌚ ... deep inheritance hierarchies;
- ⌚ ... duplication between similar implementations (DRY);
- ⌚ ... impeded maintenance;
- ⌚ ... (almost) impossible extensions (OCP);
- ⌚ ...

# A Design Challenge

---

*“Inheritance is Rarely the Answer.”*

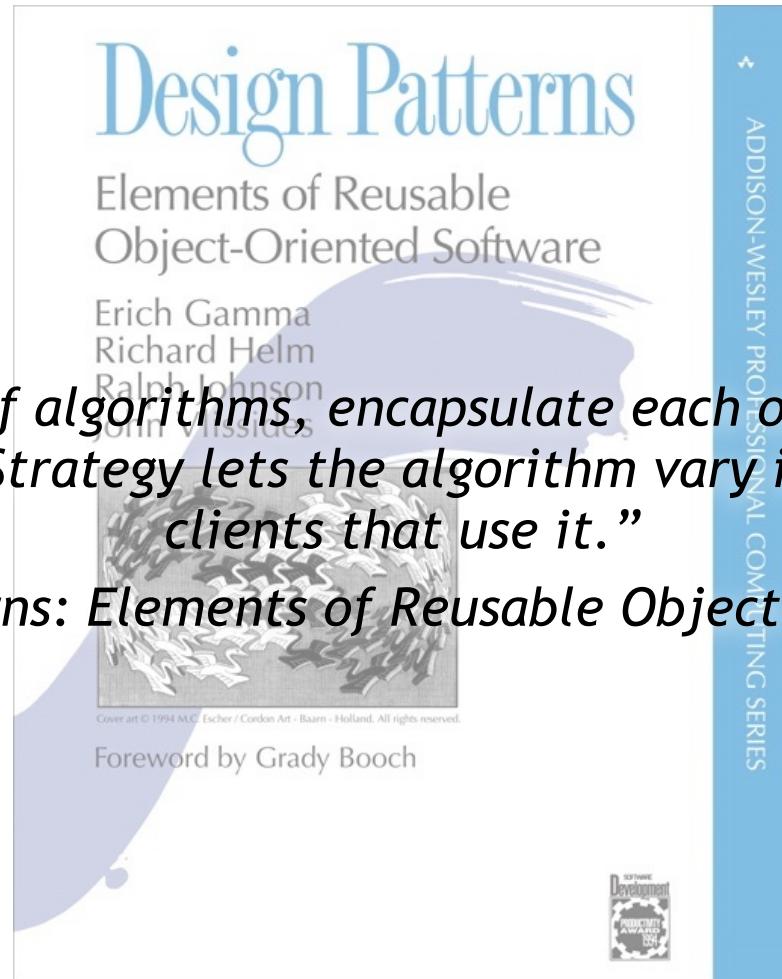
*(Andrew Hunt, David Thomas, The Pragmatic Programmer)*

# Guidelines

---

**Guideline:** Prefer containment (composition/aggregation) to inheritance.

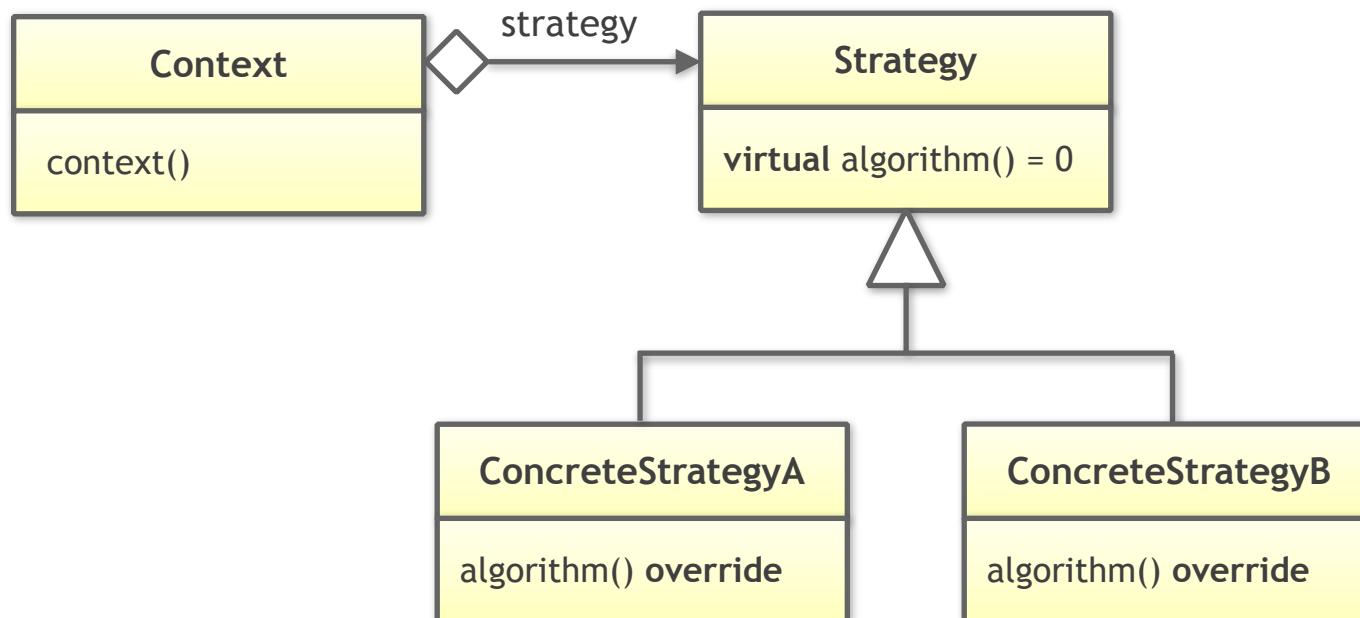
# The Intent



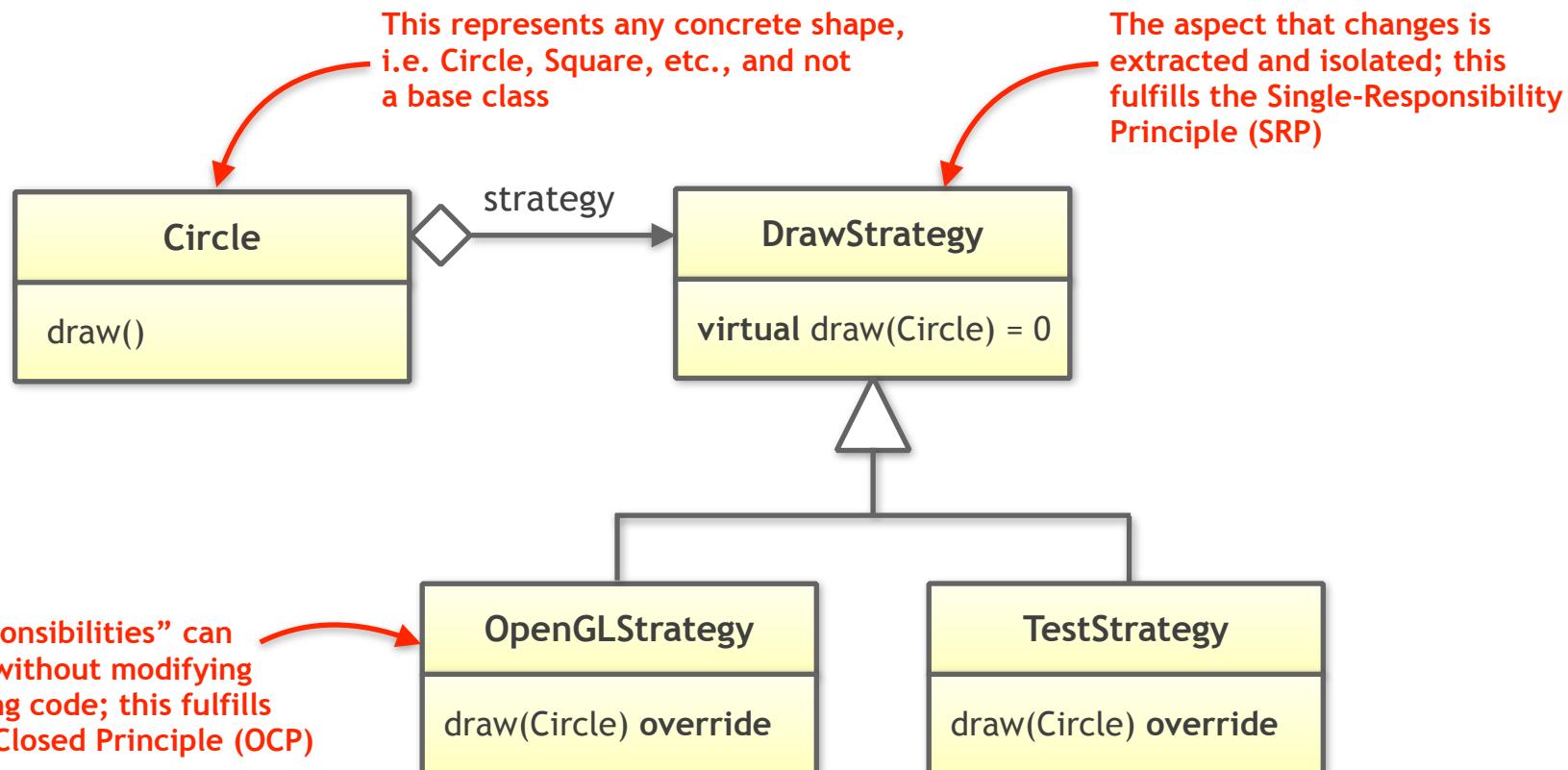
*"Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it."*

*(GoF, Design Patterns: Elements of Reusable Object-Oriented Software)*

# The Classic Strategy Design Pattern



# The Classic Strategy Design Pattern



# An Example from the Standard Library

---

Alternatively we could use static polymorphism:

```
template< typename OP >
void doSomething( OP strategy );
```

This form of the strategy pattern is used in the standard library:

```
std::vector<int> numbers{ 1, 2, 3, 4, 5, 6, 7 };
std::accumulate( begin(numbers), end(numbers), 0
                , std::plus<>{} );
```

# The Classic Strategy Design Pattern

---

**Task (C\_Modern\_Cpp\_Design\_Patterns/Strategy):** Refactor the classic Strategy solution by a value semantics based solution. Note that the general behavior should remain unchanged.

# A Strategy-Based Solution

```
class Circle;
class Square;

class DrawStrategy
{
public:
    virtual ~DrawStrategy() {}

    virtual void draw( const Circle& circle ) const = 0;
    virtual void draw( const Square& square ) const = 0;
};

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

    virtual void translate( Vector2D const& ) = 0;
    virtual void rotate( double const& ) = 0;
    virtual void draw() const = 0;
};

class Circle : public Shape
{
public:
    explicit Circle( double rad, std::unique_ptr<DrawStrategy> ds )
        : radius{ rad }
        // Remaining data members
};
```

# A Strategy-Based Solution

```
class Circle;
class Square;

class DrawStrategy
{
public:
    virtual ~DrawStrategy() {}

    virtual void draw( const Circle& circle ) const = 0;
    virtual void draw( const Square& square ) const = 0;
};

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

    virtual void translate( Vector2D const& ) = 0;
    virtual void rotate( double const& ) = 0;
    virtual void draw() const = 0;
};

class Circle : public Shape
{
public:
    explicit Circle( double rad, std::unique_ptr<DrawStrategy> ds )
        : radius{ rad }
        // Remaining data members
};
```

# A Strategy-Based Solution

```
{  
public:  
    virtual ~DrawStrategy() {}  
  
    virtual void draw( const Circle& circle ) const = 0;  
    virtual void draw( const Square& square ) const = 0;  
};  
  
class Shape  
{  
public:  
    Shape() = default;  
    virtual ~Shape() = default;  
  
    virtual void translate( Vector2D const& ) = 0;  
    virtual void rotate( double const& ) = 0;  
    virtual void draw() const = 0;  
};  
  
class Circle : public Shape  
{  
public:  
    explicit Circle( double rad, std::unique_ptr<DrawStrategy> ds )  
        : radius{ rad }  
        , // ... Remaining data members  
        , drawing{ std::move(ds) }  
    {}  
  
    double getRadius() const noexcept;  
    // ... setCenter(), setRotation(), ...  
};
```

# A Strategy-Based Solution

```
class Circle : public Shape
{
public:
    explicit Circle( double rad, std::unique_ptr<DrawStrategy> ds )
        : radius{ rad }
        , // ... Remaining data members
        , drawing{ std::move(ds) }
    {}

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

    void translate( Vector2D const& ) override;
    void rotate( double const& ) override;
    void draw() const override;

    // ...

private:
    double radius;
    // ... Remaining data members
    std::unique_ptr<DrawStrategy> drawing;
};

class Square : public Shape
{
public:
```

# A Strategy-Based Solution

```
class Square : public Shape
{
public:
    explicit Square( double s, std::unique_ptr<DrawStrategy> ds )
        : side{ s }
        , // ... Remaining data members
        , drawing{ std::move(ds) }
    {}

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

    void translate( Vector2D const& ) override;
    void rotate( double const& ) override;
    void draw() const override;

    // ...

private:
    double side;
    // ... Remaining data members
    std::unique_ptr<DrawStrategy> drawing;
};

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

# A Strategy-Based Solution

```
// ...

private:
    double side;
    // ... Remaining data members
    std::unique_ptr<DrawStrategy> drawing;
};

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

class OpenGLStrategy : public DrawStrategy
{
public:
    virtual ~OpenGLStrategy() {}

    void draw( Circle const& circle ) const override;
    void draw( Square const& square ) const override;
};

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

# A Strategy-Based Solution

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

class OpenGLStrategy : public DrawStrategy
{
public:
    virtual ~OpenGLStrategy() {}

    void draw( Circle const& circle ) const override;
    void draw( Square const& square ) const override;
};

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

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

# A Strategy-Based Solution

```
virtual ~OpenGLStrategy() {}

void draw( Circle const& circle ) const override;
void draw( Square const& square ) const override;
};

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

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

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

# A Strategy-Based Solution

```
class Circle;
class Square;

class DrawStrategy
{
public:
    virtual ~DrawStrategy() {}

    virtual void draw( const Circle& circle ) const = 0;
    virtual void draw( const Square& square ) const = 0;
};

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

    virtual void translate( Vector2D const& ) = 0;
    virtual void rotate( double const& ) = 0;
    virtual void draw() const = 0;
};

class Circle : public Shape
{
public:
    explicit Circle( double rad, std::unique_ptr<DrawStrategy> ds )
        : radius{ rad }
        // Remaining data members
};
```

# A Strategy-Based Solution

```
class Circle;
class Square;

class DrawCircleStrategy
{
public:
    virtual ~DrawCircleStrategy() {}

    virtual void draw( const Circle& circle ) const = 0;
};

class DrawSquareStrategy
{
public:
    virtual ~DrawSquareStrategy() {}

    virtual void draw( const Square& square ) const = 0;
};

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

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

# A Strategy-Based Solution

```
};
```

```
class Circle : public Shape
{
public:
    explicit Circle( double rad, std::unique_ptr<DrawCircleStrategy> ds )
        : radius{ rad }
        , // ... Remaining data members
        , drawing{ std::move(ds) }
    {}

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

    void translate( Vector2D const& ) override;
    void rotate( double const& ) override;
    void draw() const override;

    // ...

private:
    double radius;
    // ... Remaining data members
    std::unique_ptr<DrawCircleStrategy> drawing;
};
```

```
class Square : public Shape
{
public:
```

# A Strategy-Based Solution

```
class Square : public Shape
{
public:
    explicit Square( double s, std::unique_ptr<DrawSquareStrategy> ds )
        : side{ s }
        , // ... Remaining data members
        , drawing{ std::move(ds) }
    {}

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

    void translate( Vector2D const& ) override;
    void rotate( double const& ) override;
    void draw() const override;

    // ...

private:
    double side;
    // ... Remaining data members
    std::unique_ptr<DrawSquareStrategy> drawing;
};

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

# A Strategy-Based Solution

```
{  
    s->draw()  
}  
}  
  
  
class OpenGLCircleStrategy : public DrawCircleStrategy  
{  
public:  
    virtual ~OpenGLStrategy() {}  
  
    void draw( Circle const& circle ) const override;  
};  
  
class OpenGLSquareStrategy : public DrawSquareStrategy  
{  
public:  
    virtual ~OpenGLStrategy() {}  
  
    void draw( Square const& square ) const override;  
};  
  
  
int main()  
{  
    using Shapes = std::vector<std::unique_ptr<Shape>>;  
  
    // Creating some shapes  
    Shapes shapes;  
    shapes.emplace_back( std::make_unique<Circle>( 2.0  
                                                , std::make_unique<OpenGLCircleStrategy>() ) );
```

# A Strategy-Based Solution

```
virtual ~OpenGLStrategy() {}

void draw( Square const& square ) const override;
};

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

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

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

## std::function

---

- Is a wrapper around a callable with a specific signature
- The canonical example for value semantics
- Not intrusive!
- You can have local variables of type std::function (including function parameters) as well as members (not forced for them to be pointers or references)
- std::function may allocate as a fallback mechanism

# std::function

---

```
#include <functional>

void foo(int i) {
    std::cout << "foo: " << i << '\n';
}

int main()
{
    std::function<void(int)> f{};

    f = [](int i){ std::cout << "lambda: " << i << '\n'; };

    auto g = f; // Value semantics: Creates a deep copy

    f = foo;

    f( 1 );
    g( 2 );
}
```

# std::function

---

Note that function is available in ...

C++11:

```
#include <functional>
using Callback = std::function<void()>;
```

C++11

Boost:

```
#include <boost/function.hpp>
typedef boost::function<void()> Callback;
```

C++03

# A “Modern C++” Solution

```
class Circle;
class Square;

using DrawCircleStrategy = std::function<void(Circle const&)>;
using DrawSquareStrategy = std::function<void(Square const&)>;

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

    virtual void translate( Vector2D const& ) = 0;
    virtual void rotate( double const& ) = 0;
    virtual void draw() const = 0;
};

class Circle : public Shape
{
public:
    explicit Circle( double rad, DrawCircleStrategy ds )
        : radius{ rad }
        , // ... Remaining data members
        , drawing{ std::move(ds) }
    {}

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

# A “Modern C++” Solution

```
class Circle;
class Square;

using DrawCircleStrategy = std::function<void(Circle const&)>;
using DrawSquareStrategy = std::function<void(Square const&)>;

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

    virtual void translate( Vector2D const& ) = 0;
    virtual void rotate( double const& ) = 0;
    virtual void draw() const = 0;
};

class Circle : public Shape
{
public:
    explicit Circle( double rad, DrawCircleStrategy ds )
        : radius{ rad }
        , // ... Remaining data members
        , drawing{ std::move(ds) }
    {}

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

# A “Modern C++” Solution

```
class Circle;
class Square;

using DrawCircleStrategy = std::function<void(Circle const&)>;
using DrawSquareStrategy = std::function<void(Square const&)>;

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

    virtual void translate( Vector2D const& ) = 0;
    virtual void rotate( double const& ) = 0;
    virtual void draw() const = 0;
};

class Circle : public Shape
{
public:
    explicit Circle( double rad, DrawCircleStrategy ds )
        : radius{ rad }
        , // ... Remaining data members
        , drawing{ std::move(ds) }
    {}

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

# A “Modern C++” Solution

```
};
```

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

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

    void translate( Vector2D const& ) override;
    void rotate( double const& ) override;
    void draw() const override;

    // ...

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

```
class Square : public Shape
{
```

# A “Modern C++” Solution

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

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

    void translate( Vector2D const& ) override;
    void rotate( double const& ) override;
    void draw() const override;

    // ...

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

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

# A “Modern C++” Solution

```
{  
    for( auto const& s : shapes )  
    {  
        s->draw()  
    }  
}  
  
class OpenGLCircleStrategy  
{  
public:  
    void operator()( Circle const& circle ) const;  
};  
  
class OpenGLSquareStrategy  
{  
public:  
    void operator()( Square const& square ) const;  
};  
  
int main()  
{  
    using Shapes = std::vector<std::unique_ptr<Shape>>;  
  
    // Creating some shapes  
    Shapes shapes;  
    shapes.emplace_back( std::make_unique<Circle>( 2.0  
                                                , OpenGLCircleStrategy{} ) );  
    shapes.emplace_back( std::make_unique<Square>( 1.5  
                                                , OpenGLSquareStrategy{} ) );  
    shapes.emplace_back( std::make_unique<Circle>( 1.2  
                                                , OpenGLCircleStrategy{} ) );  
}
```

# A “Modern C++” Solution

```
{  
public:  
    void operator()( Square const& square ) const;  
};  
  
int main()  
{  
    using Shapes = std::vector<std::unique_ptr<Shape>>;  
  
    // Creating some shapes  
    Shapes shapes;  
    shapes.emplace_back( std::make_unique<Circle>( 2.0  
                                                , OpenGLCircleStrategy{} ) );  
    shapes.emplace_back( std::make_unique<Square>( 1.5  
                                                , OpenGLSquareStrategy{} ) );  
    shapes.emplace_back( std::make_unique<Circle>( 4.2  
                                                , OpenGLCircleStrategy{} ) );  
  
    // Drawing all shapes  
    drawAllShapes( shapes );  
}
```

# A “Modern C++” Solution

```
{  
    for( auto const& s : shapes )  
    {  
        s->draw()  
    }  
}  
  
class OpenGLCircleStrategy  
{  
public:  
    void operator()( Circle const& circle ) const;  
};  
  
class OpenGLSquareStrategy  
{  
public:  
    void operator()( Square const& square ) const;  
};  
  
int main()  
{  
    using Shapes = std::vector<std::unique_ptr<Shape>>;  
  
    // Creating some shapes  
    Shapes shapes;  
    shapes.emplace_back( std::make_unique<Circle>( 2.0  
                                                , OpenGLCircleStrategy{} ) );  
    shapes.emplace_back( std::make_unique<Square>( 1.5  
                                                , OpenGLSquareStrategy{} ) );  
    shapes.emplace_back( std::make_unique<Shape>( 1.2  
                                                , OpenGLCircleStrategy{} ) );  
}
```

# A “Modern C++” Solution

# A “Modern C++” Solution

```
{  
    for( auto const& s : shapes )  
    {  
        s->draw()  
    }  
}  
  
void draw( Circle const& circle ) const;  
void draw( Square const& square ) const;  
  
struct Draw  
{  
    template< typename T >  
    void operator()( T const& drawable ) const {  
        draw( drawable );  
    }  
};  
  
int main()  
{  
    using Shapes = std::vector<std::unique_ptr<Shape>>;  
  
    // Creating some shapes  
    Shapes shapes;  
    shapes.emplace_back( std::make_unique<Circle>( 2.0, Draw{} ) );  
    shapes.emplace_back( std::make_unique<Square>( 1.5, Draw{} ) );  
    shapes.emplace_back( std::make_unique<Circle>( 4.2, Draw{} ) );  
  
    // Drawing all  
}
```

# A “Modern C++” Solution

```
template< typename T >
void operator()( T const& drawable ) const {
    draw( drawable );
}

int main()
{
    using Shapes = std::vector<std::unique_ptr<Shape>>;
    // Creating some shapes
    Shapes shapes;
    shapes.emplace_back( std::make_unique<Circle>( 2.0, Draw{} ) );
    shapes.emplace_back( std::make_unique<Square>( 1.5, Draw{} ) );
    shapes.emplace_back( std::make_unique<Circle>( 4.2, Draw{} ) );

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

# A “Modern C++” Solution

```
{  
    for( auto const& s : shapes )  
    {  
        s->draw()  
    }  
}  
  
void draw( Circle const& circle ) const;  
void draw( Square const& square ) const;  
  
struct Draw  
{  
    template< typename T >  
    void operator()( T const& drawable ) const {  
        draw( drawable );  
    }  
};  
  
int main()  
{  
    using Shapes = std::vector<std::unique_ptr<Shape>>;  
  
    // Creating some shapes  
    Shapes shapes;  
    shapes.emplace_back( std::make_unique<Circle>( 2.0, Draw{} ) );  
    shapes.emplace_back( std::make_unique<Square>( 1.5, Draw{} ) );  
    shapes.emplace_back( std::make_unique<Circle>( 4.2, Draw{} ) );  
  
    // Drawing all shapes  
    for( auto const& s : shapes )  
    {  
        s->draw()  
    }  
}
```

# A “Modern C++” Solution

# A “Modern C++” Solution

```
class Circle;
class Square;

using DrawCircleStrategy = std::function<void(Circle const&)>;
using DrawSquareStrategy = std::function<void(Square const&)>;

struct Draw
{
    template< typename T >
    void operator()( T const& drawable ) const {
        draw( drawable );
    }
};

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

    virtual void translate( Vector2D const& ) = 0;
    virtual void rotate( double const& ) = 0;
    virtual void draw() const = 0;
};

class Circle : public Shape
{
public:
    explicit Circle( double rad, DrawCircleStrategy ds = DrawCircleStrategy() )
        : radius{ rad }, drawStrategy{ std::move(ds) } { }

    void draw() const override {
        drawStrategy( *this );
    }

    void translate( Vector2D const& v ) override {
        center += v;
    }

    void rotate( double angle ) override {
        angle_ += angle;
    }

private:
    double radius;
    Vector2D center;
    double angle_ = 0;
    DrawCircleStrategy drawStrategy;
};
```

# A “Modern C++” Solution

```
class Circle : public Shape
{
public:
    explicit Circle( double rad, DrawCircleStrategy ds = Draw{} )
        : radius{ rad }
        , // ... Remaining data members
        , drawing{ std::move(ds) }
    {}

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

    void translate( Vector2D const& ) override;
    void rotate( double const& ) override;
    void draw() const override;

    // ...

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

class Square : public Shape
{
public:
```

# A “Modern C++” Solution

```
class Square : public Shape
{
public:
    explicit Square( double s, DrawSquareStrategy ds = Draw{} )
        : side{ s }
        , // ... Remaining data members
        , drawing{ std::move(ds) }
    {}

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

    void translate( Vector2D const& ) override;
    void rotate( double const& ) override;
    void draw() const override;

    // ...

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

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

# A “Modern C++” Solution

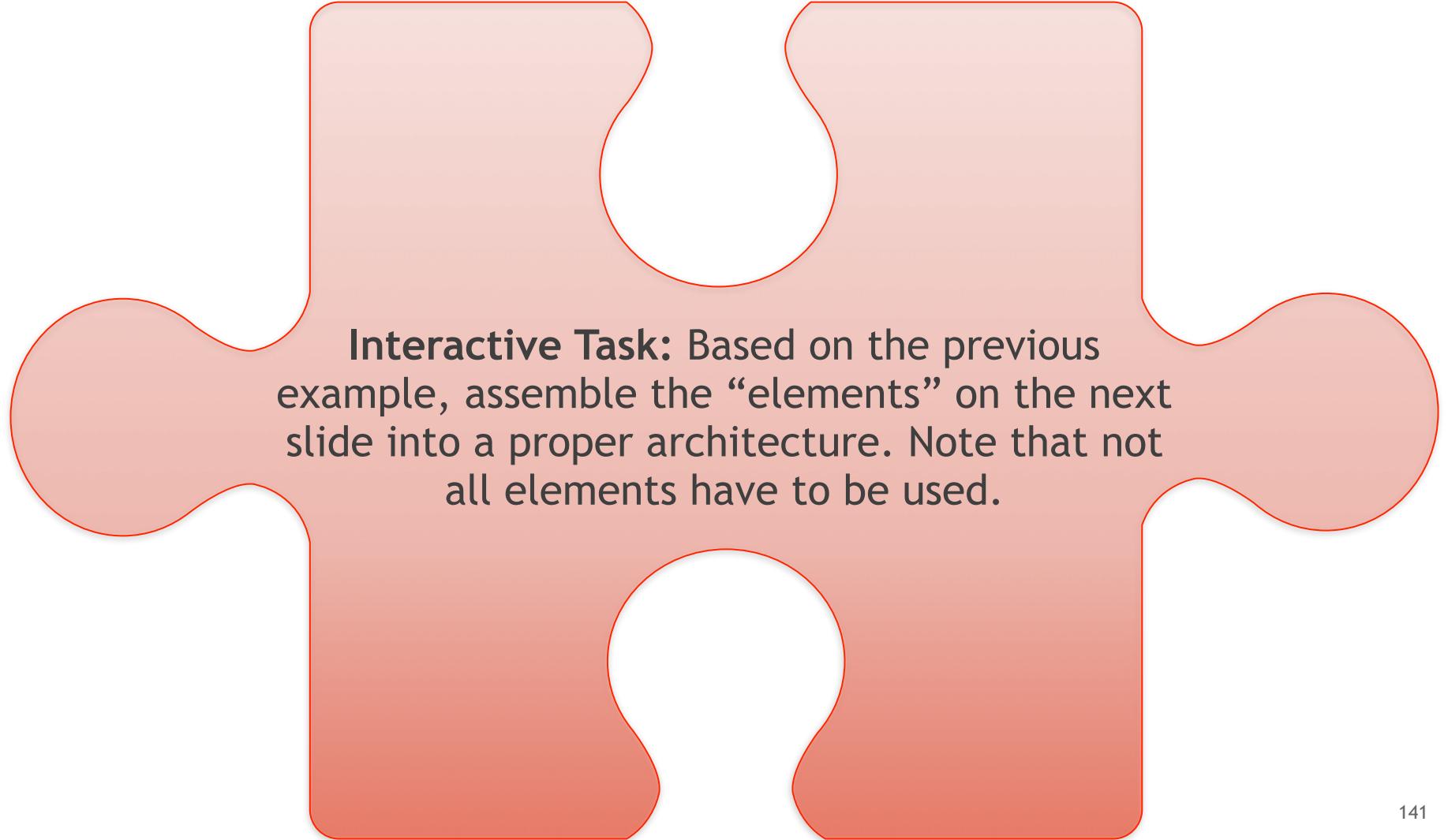
---

```
void draw( Circle const& circle ) const;
void draw( Square const& square ) const;

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 Architectural Puzzle



**Interactive Task:** Based on the previous example, assemble the “elements” on the next slide into a proper architecture. Note that not all elements have to be used.

# An Architectural Puzzle

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

```
class Circle : public Shape
{
public:
    explicit Circle( double rad, std::unique_ptr<DrawCircleStrategy> strategy );
    // ...
};
```

Architectural Boundary

```
class Circle;
class DrawCircleStrategy
{
public:
    virtual ~DrawCircleStrategy() {}
    virtual void draw( const Circle& circle ) const = 0;
};
```

```
class OpenGLCircleStrategy : public DrawCircleStrategy
{
public:
    void draw( const Circle& circle ) const override;
};
```

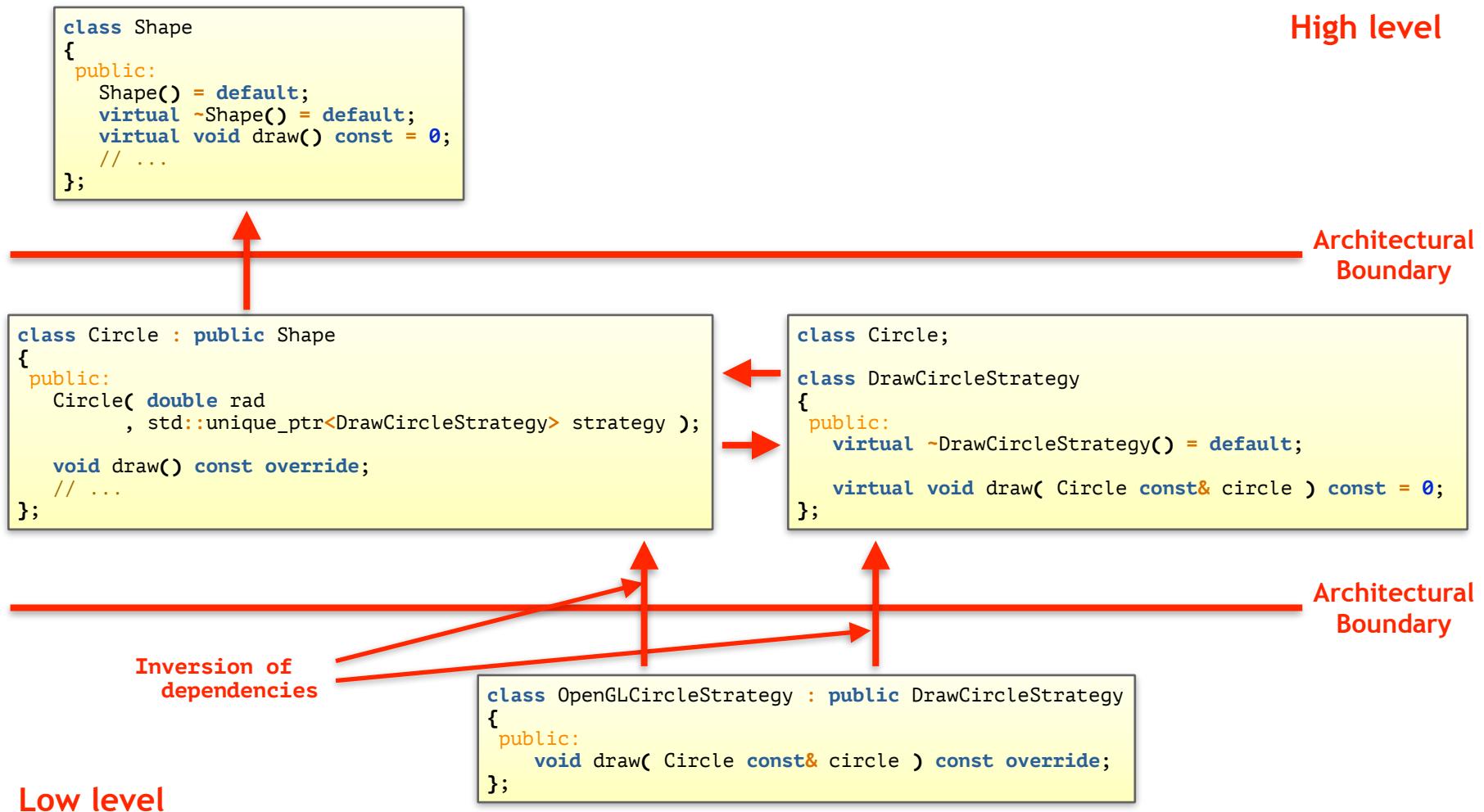
Architectural Boundary

Low level



High level

# An Architectural Puzzle (Solution)



# An Architectural Puzzle (Solution)

---

```
//---- <DrawStrategy.h> ----

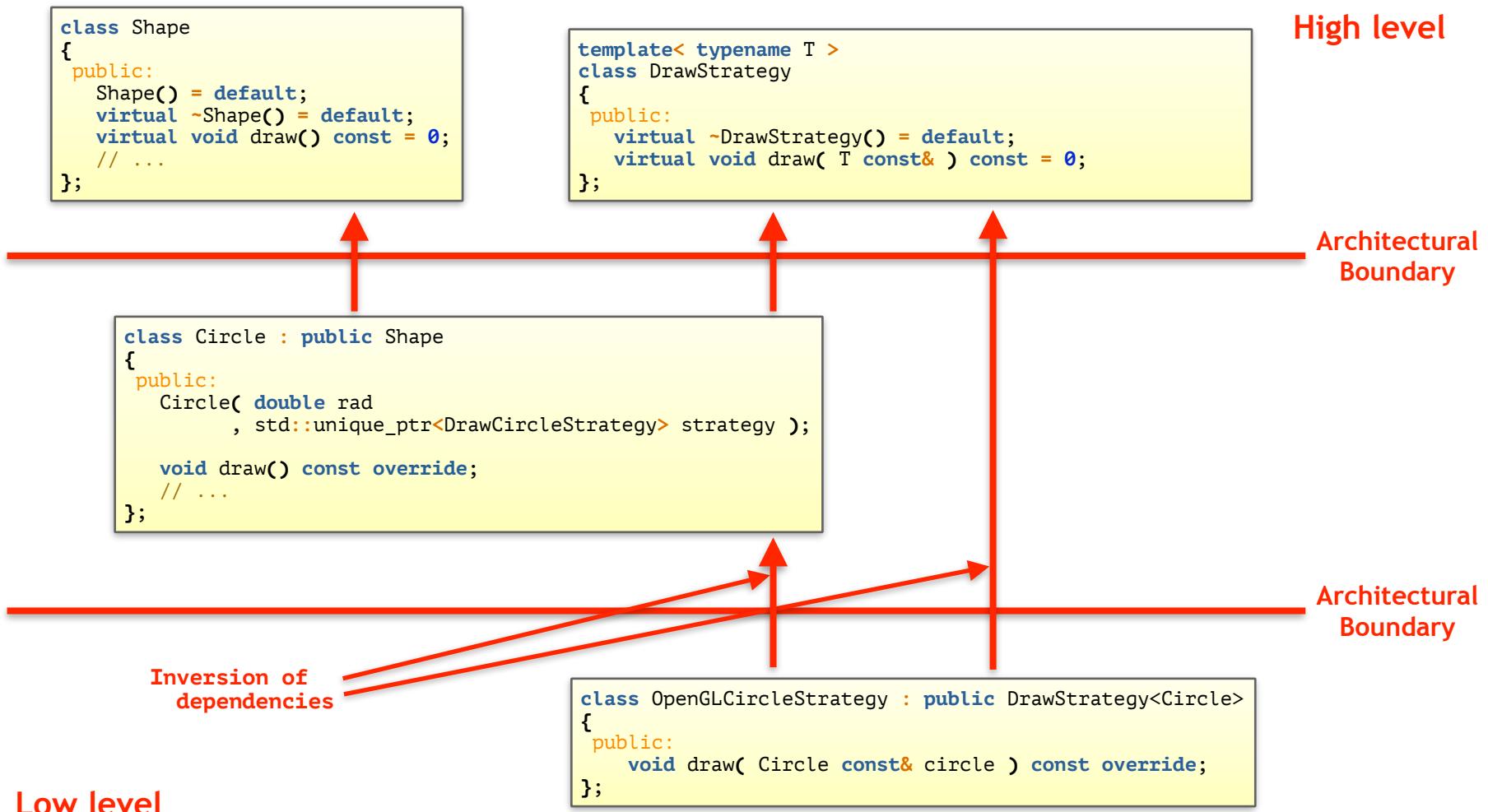
template< typename T >
class DrawStrategy
{
    public:
        virtual ~DrawStrategy() = default;
        virtual void draw( T const& circle ) const = 0;
};

//---- <OpenGLCircleStrategy.h> ----

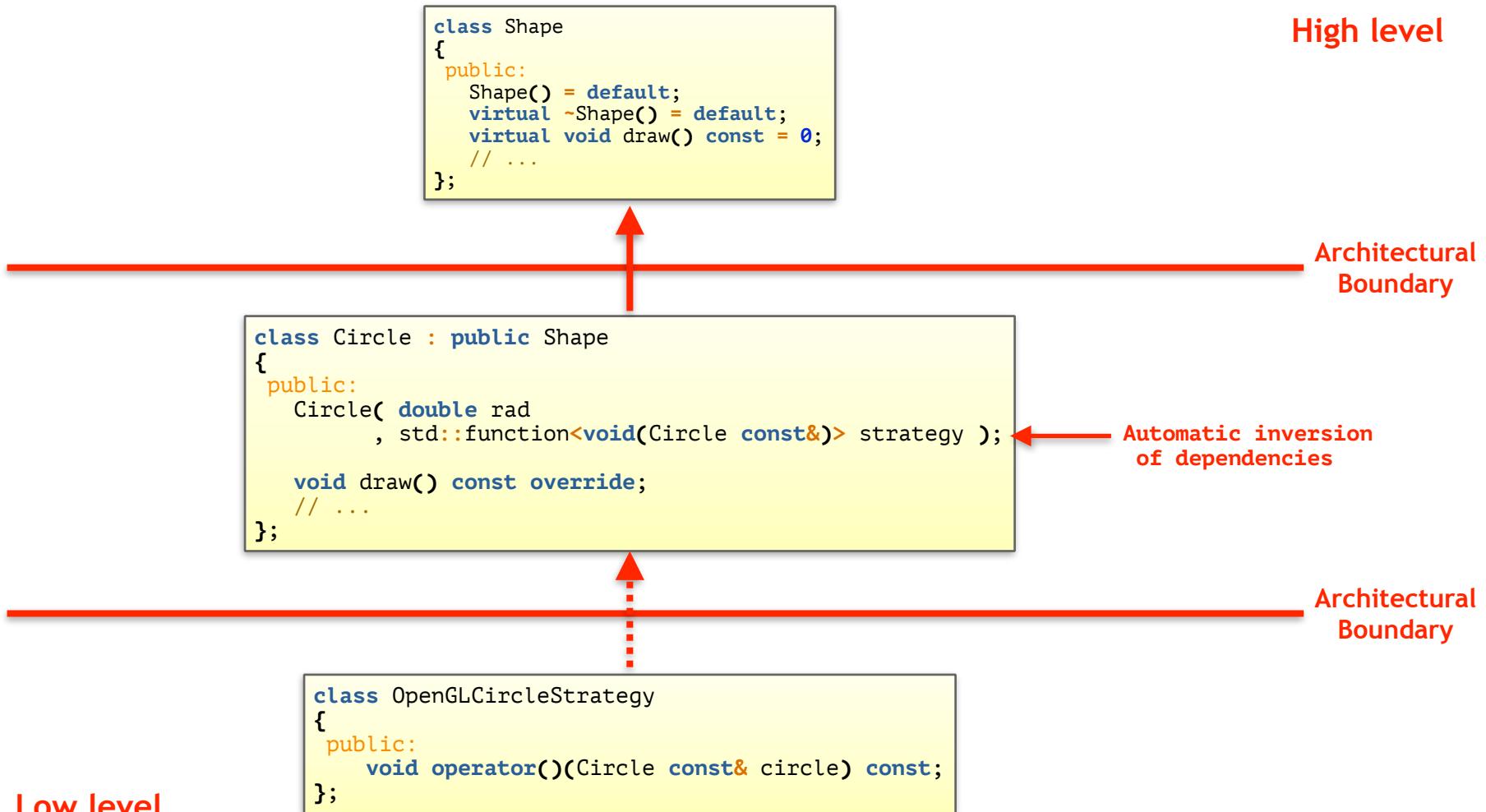
#include <Circle.h>
#include <DrawStrategy.h>
// ... Potentially some OpenGL-related header

class OpenGLCircleStrategy : public DrawStrategy<Circle>
{
    public:
        void draw( Circle const& circle ) const override;
};
```

# An Architectural Puzzle (Solution)



# Comparison with std::function



# The Classic Strategy Design Pattern

---

The classic Strategy design pattern ...

- ... requires a base class (dependency);
- ... promotes heap allocation;
- ... requires memory management.

Using std::function instead of the classic Strategy design pattern ...

- ... simplifies code;
- ... facilitates comprehension;
- ... reduces dependencies.

# std::function – Advantages/Disadvantages

---

Use std::function if ...

- ... you want to abstract and to **decouple a single function**;
- ... performance is not the **primary concern**.

Don't use std::function if ...

- ... you need to **decouple several cohesive functions**;
- ... you require **maximum performance**.

# Guidelines

---

**Guideline:** Use the Strategy pattern to encapsulate interchangeable behaviours and use delegation to decide which behavior to use.

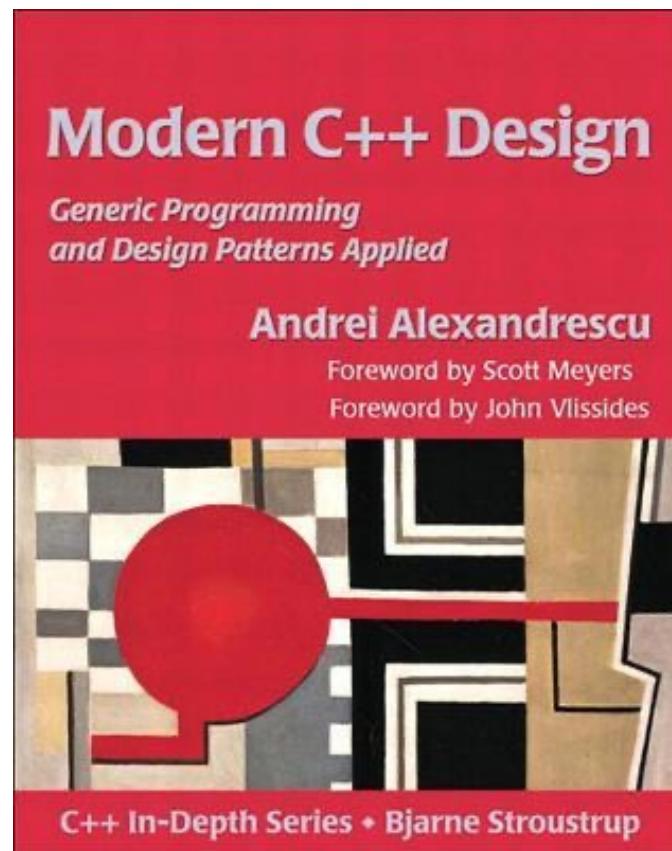
**Guideline:** Prefer containment (composition/aggregation) to inheritance.

## 2.5. Policy-Based Design

---

# The Origin of Policy-Based Design

---



## Example: STL Containers

---

- Every STL container has a template parameter for an allocator
- The associative containers have template parameters for several more properties
- By means of these template parameters, important behavior is customizable

```
template< class T, class Allocator = std::allocator<T> >
class vector;
```

```
template< class T, class Allocator = std::allocator<T> >
class list;
```

```
template< class Key
        , class Hash = std::hash<Key>
        , class KeyEqual = std::equal_to<Key>
        , class Allocator = std::allocator<Key> >
class unordered_set;
```

## Example: Smart Pointers

---

- The cleanup behavior of `std::unique_ptr` can be customized by means of a template parameter
- By default `std::default_delete` is used, which calls `delete`

```
template<
    typename T,
    typename Deleter = std::default_delete<T>>
class unique_ptr;
```

# Example: Smart Pointers

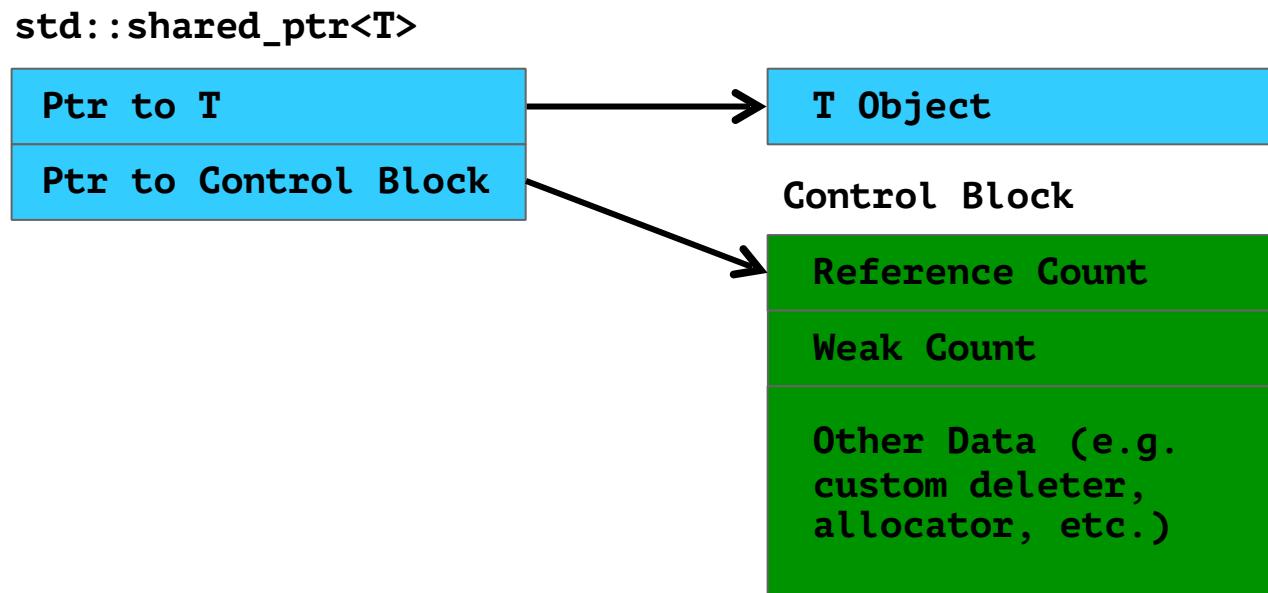
---

```
struct ChattyDelete
{
    template< typename T >
    void operator()( T* ptr ) const noexcept
    {
        std::cout << "Deleting ptr (" << ptr << ")\n";
        delete ptr;
    }
};

int main()
{
    std::unique_ptr<int, ChattyDelete> uptr( new int(42) );

    std::shared_ptr<int> sptr( new int(42), ChattyDelete{} );
}
```

# Example: Smart Pointers



# Guidelines

---

**Guideline:** Use Policy-Based Design as a compile time equivalent of the Strategy design pattern.

## 2.6. Interlude

---

# The Expert's Attitude

GoingNative 2013 Inheritance Is The Base Class of Evil

Inheritance Is The Base Class of Evil  
Sean Parent | Principal Scientist

© 2013 Adobe Systems Incorporated. All Rights Reserved.

0:37 / 24:19

68

A screenshot of a video player interface. The main content area displays a presentation slide with a dark blue header containing the text "GoingNative 2013 Inheritance Is The Base Class of Evil". Below the header, there is a red square icon with a white letter "A". The main title of the slide is "Inheritance Is The Base Class of Evil" and the author is listed as "Sean Parent | Principal Scientist". The background of the slide features abstract, overlapping shapes in shades of red, orange, and grey. At the bottom of the slide, a copyright notice reads "© 2013 Adobe Systems Incorporated. All Rights Reserved.". The video player interface includes a progress bar at the bottom left showing "0:37 / 24:19", and various control icons for play, volume, and settings at the bottom right. The number "68" is visible in the bottom right corner of the video frame.

# The Expert's Attitude



A large slide with a white background and a teal header and footer. The main feature is the text "{OOP}" in a large, bold, black font. In the bottom right corner of the white area, there is a block of text: "Presentation by Jon Kalb", "CppCon 2019, 2019-09-17", and "thanks Andrei, Herb, & Scott". The footer contains standard presentation control icons.

{OOP}

Back to Basics: Object-Oriented Programming

Presentation by Jon Kalb  
CppCon 2019, 2019-09-17  
thanks Andrei, Herb, & Scott

# The Expert's Attitude

Cppcon | 2019  
The C++ Conference  
cppcon.org

Jon Kalb

Back to Basics:  
Object-Oriented  
Programming

Video Sponsorship Provided By:  
ansatz

▶ ▶ 🔍 1:27 / 59:58

THE NEW STACK Ebooks ▾ Podcasts ▾ Events Newsletter

Architecture ▾ Development ▾ Operations ▾

CULTURE / DEVELOPMENT

## Why Are So Many Developers Hating on Object-Oriented Programming?

21 Aug 2019 12:00pm, by David Cassel

# The Expert's Attitude

Can a browser engine be successful with data-oriented design?

STOYAN NIKOLOV

OOP is dead, long live Data-oriented design

CppCon.org

3

CppCon 2018 | @stoyannk

▶ ▶ 🔍 2:49 / 1:00:45

cc HD

THE C++ CONFERENCE • BELLEVUE, WASHINGTON

cppcon | 2018

# The Expert's Attitude

---

*”... [Programming by difference] fell out of favor in the 1990s when many people in the OO community noticed that inheritance can be rather problematic if it is overused.”*

*(Michael C. Feathers, Working Effectively with Legacy Code)*

# The Expert's Attitude

---

- Why Are So Many Developers Hating on Object-Oriented Programming (David Cassel) (<https://thenewstack.io/why-are-so-many-developers-hating-on-object-oriented-programming/>)
- The Forgotten History of OOP (Eric Elliott) (<https://medium.com/javascript-scene/the-forgotten-history-of-oop-88d71b9b2d9f>)
- If everyone hates it, why is OOP still so widely spread? (Medi Madelen Gwosdz) (<https://stackoverflow.blog/2020/09/02/if-everyone-hates-it-why-is-oop-still-so-widely-spread/>)
- Uncle Bob SOLID principles (<https://www.youtube.com/watch?v=zHiWqnTWsn4&t=2264s>)

# Why is it so bad?

---

- Does not harmonize with the philosophy of the STL (value semantics)
- Fails to model many sub typing relations (i.e. it is hard)

# Overuse of Inheritance

---

**Task:** Implement the `max()` algorithm by means of inheritance.

```
class Comparable
{
public:
    virtual ~Comparable() = default;
    virtual bool lessThan( const Comparable& other ) = 0;
    // ...
};

// ...

const Comparable& max( const Comparable& a, const Comparable& b );
```

- Types like `int` and `std::string` cannot be compared
- The interface allows to compare different types
  - How should we deal with the comparison of different types?
- For every comparison we have to call a virtual function
- Do we use a `dynamic_cast`?

# Overuse of Inheritance

---

```
template< typename T >
const T& max( const T& a, const T& b );
```

- Any types can be compared (even `int` and `std::string`)
- The interface allows to compare only values of the same type
  - No error handling for comparing different types
- No virtual function call, no `dynamic_cast`, but inlining

# Guidelines

---

**Guideline:** Remember that inheritance is about behaviour, not about data!

# Why is it so bad?

---

- Does not harmonize with the philosophy of the STL
- Fails to model many sub typing relations (i.e. it is hard)
- Inheritance creates a very tight coupling (second only to friendship)
  - Intrusive: Forces us to inherit, or ...
  - Verbose: Forces us to wrap perfectly good types to conform to a hierarchy
- Adding functions requires modifications (violation of the OCP)
  - May cause contradictions between SRP and OCP
- Inheritance introduces overhead:
  - Heap allocation (memory management), leads to pointers (nullptr, dangling pointers, ...)

# Dynamic Allocation

---

```
class Animal { virtual ~Animal() = default; };

class Cat : public Animal { /*...*/ };
class Dog : public Animal { /*...*/ };

Animal make_animal();

std::vector<Animal> v{};
```

# Dynamic Allocation

---

```
class Animal { virtual ~Animal() = default; };

class Cat : public Animal { /*...*/ };
class Dog : public Animal { /*...*/ };

Animal make_animal();

std::vector<Animal> v{};
```

# Disclaimer

In this C++ training, no animals were hurt, tortured, or sliced!

# Dynamic Allocation

---

```
class Animal { virtual ~Animal() = default; };

class Cat : public Animal { /*...*/ };
class Dog : public Animal { /*...*/ };

std::unique_ptr<Animal> make_animal();

std::vector<std::unique_ptr<Animal>> v{};
```

# Why is it so bad?

---

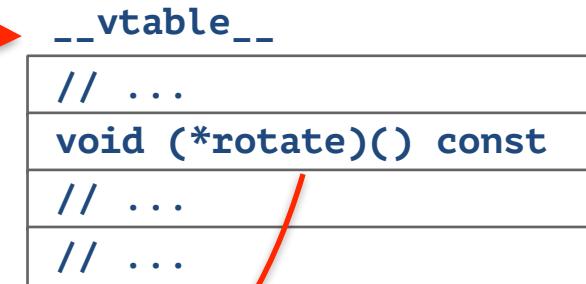
- Does not harmonize with the philosophy of the STL
- Fails to model many sub typing relations (i.e. it is hard)
- Inheritance creates a very tight coupling (second only to friendship)
  - Intrusive: Forces us to inherit, or ...
  - Verbose: Forces us to wrap perfectly good types to conform to a hierarchy
- Adding functions requires modifications (violation of the OCP)
  - May cause contradictions between SRP and OCP
- Inheritance introduces overhead:
  - Heap allocation (memory management), leads to pointers (nullptr, dangling pointers)
  - Virtual functions (no inlining)

# Virtual Dispatch

```
struct Shape {  
    void translate();  
    virtual void rotate();  
    // ...  
  
    __VTable__* __vptr__;  
};
```

```
struct Circle : public Shape {  
    void translate();  
    void rotate() override;  
    // ...  
};
```

```
int main()  
{  
    Circle circle{};  
    Shape& shape = derived;  
    shape.rotate();  
    circle.rotate();  
}
```



# Why is it so bad?

---

- Does not harmonize with the philosophy of the STL
- Fails to model many sub typing relations (i.e. it is hard)
- Inheritance creates a very tight coupling (second only to friendship)
  - Intrusive: Forces us to inherit, or ...
  - Verbose: Forces us to wrap perfectly good types to conform to a hierarchy
- Adding functions requires modifications (violation of the OCP)
  - May cause contradictions between SRP and OCP
- Inheritance introduces overhead:
  - Heap allocation (memory management), leads to pointers (nullptr, dangling pointers)
  - Virtual functions (no inlining)
  - RTTI

## 2. C++ Software Design - Interlude



# Why is it so bad?

---

- ...
- Forces us to use pointers and references
  - One extra indirection (runtime overhead)
  - Allocations: memory fragmentation, synchronisation, may fail
  - Shallow vs. deep copy
  - Disables local reasoning
  - Incentives sharing, which complicates lifetime management

# Value Types

---

```
class A
{
public:
    A( int m, int c ) : mult( m ), offset( c ) {}

    int foo( int x ) const
    {
        return mult * x + offset;
    }

private:
    int mult;
    int offset;
};
```

# Value Types

---

- Automatic Memory Management (no garbage collection required)
- Exception-safe
- Does not change
- No side effects
- Can use the same value in multiple threads (lock-free)
- Deterministic
- Pure
- Regular

→ Value Semantics

# Pointers and References

---

```
class A
{
public:
    A( int const& m, int* c ) : mult( m ), offset( c ) {}

    int foo( int x ) const
    {
        return mult * x + *offset;
    }

private:
    int const& mult;
    int* offset;
};
```

# Pointers and References

---

- Do I have to delete offset?
- Does someone else have write access to mult and offset?
- Do I need a mutex?
- Is it deterministic?
- It is a bug hive

→ Reference Semantics

# Custom Types

---

```
class A
{
public:
    A( Mult m, Offset c ) : mult( m ), offset( c ) {}

    int foo( int x ) const
    {
        return mult * x + offset;
    }

private:
    Mult mult;
    Offset offset;
};
```

# Custom Types

---

- Do Mult and Offset behave like int? → Good!
- Do Mult and Offset behave like int\* or int&? → Bad!

## 2. C++ Software Design - Interlude



# Guidelines

---

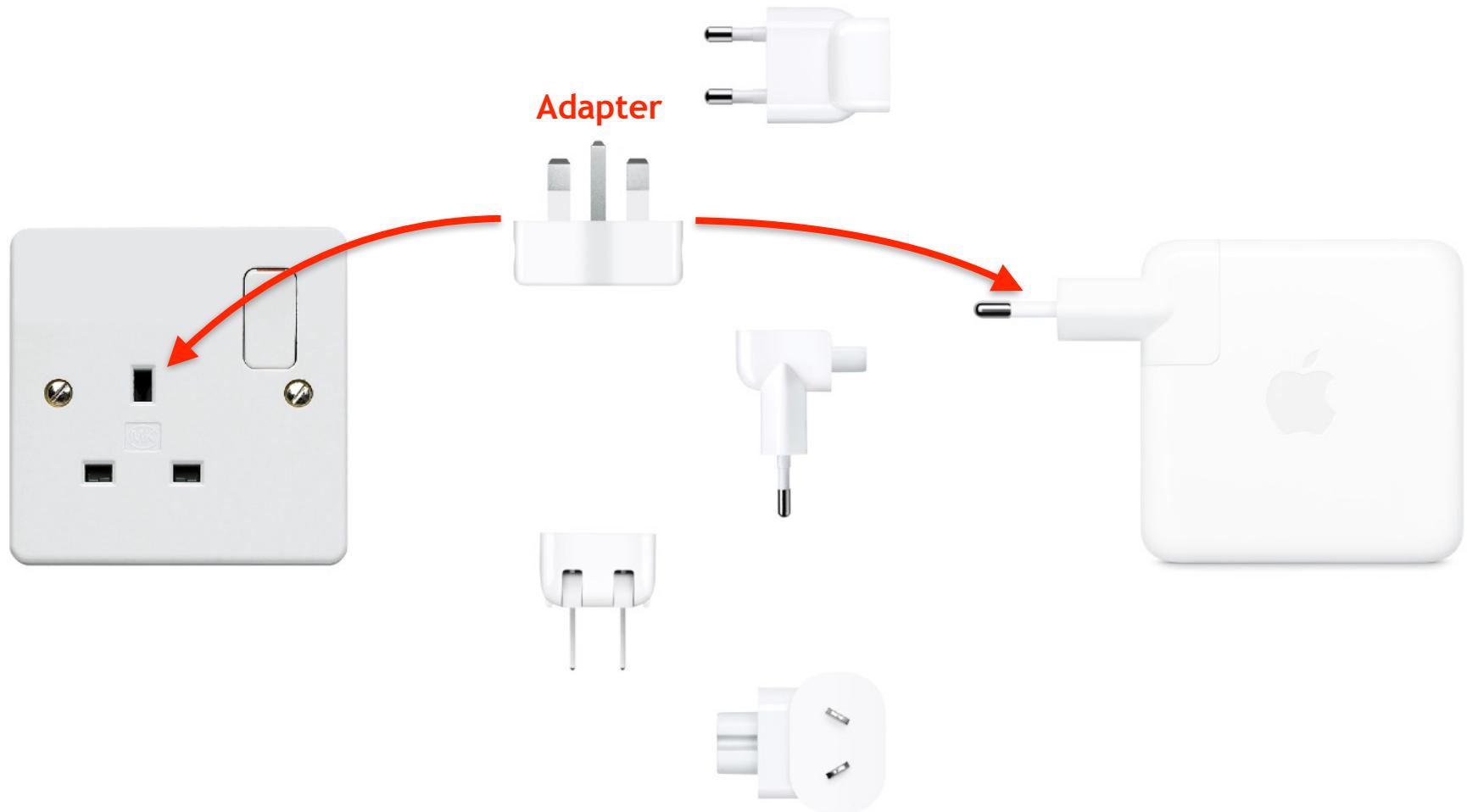
**Guideline:** Prefer value semantics over reference semantics!

**Guideline:** Try to reduce the use of pointers!

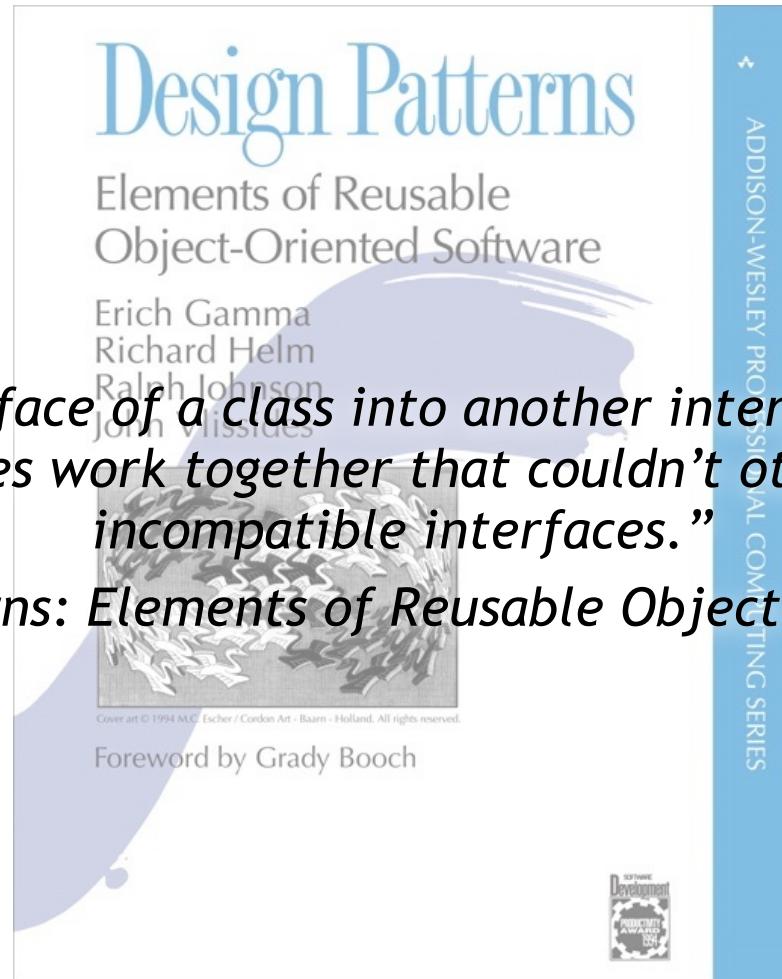
## 2.7. Adapter

---

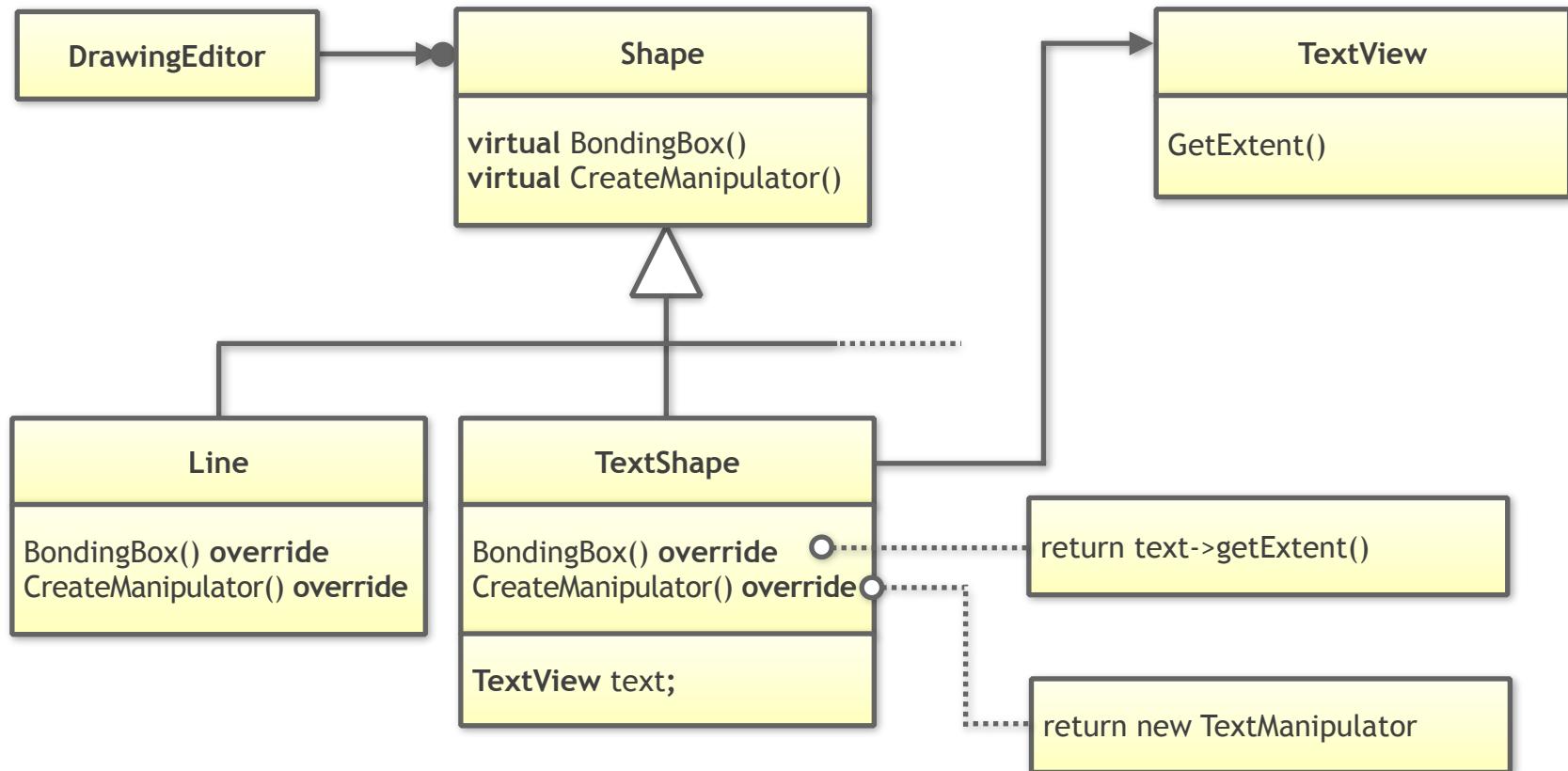
# The Classic Adapter Design Pattern



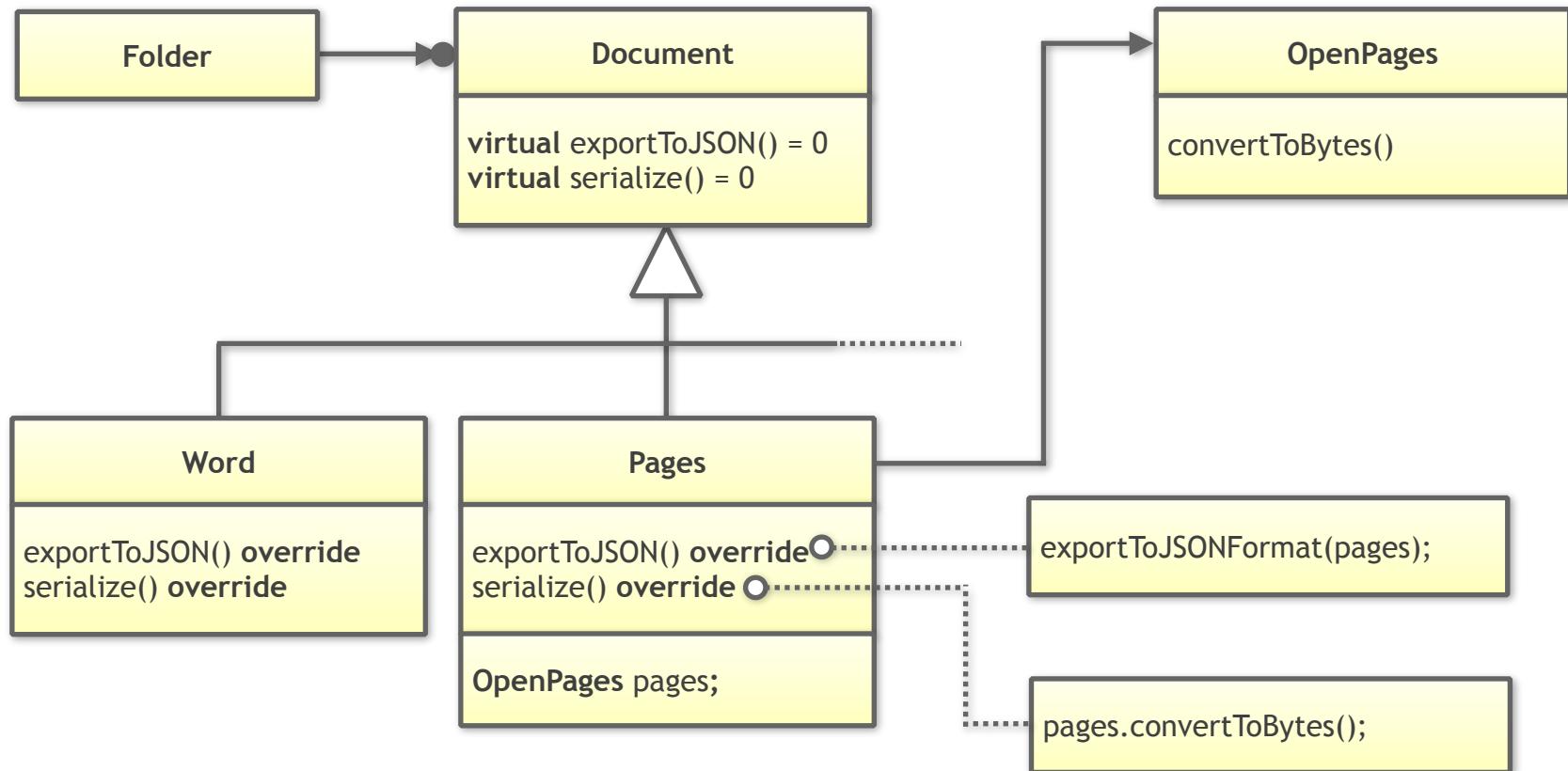
# The Intent



# The Classic Adapter Design Pattern



# The Classic Adapter Design Pattern



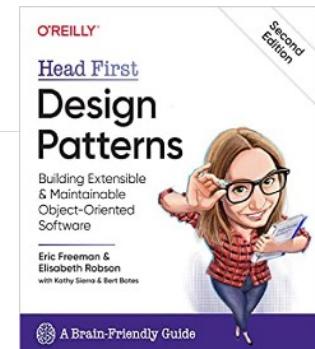
# Adapter Used For Duck-Typing

```
class Duck
{
public:
    virtual ~Duck() = default;
    virtual void quack() = 0;
    virtual void fly() = 0;
};

class MallardDuck : public Duck
{
public:
    void quack() override { /*...*/ }
    void fly() override { /*...*/ }
};

class Turkey
{
public:
    virtual ~Turkey() = default;
    virtual void gobble() = 0; // Turkeys don't quack, they gobble!
    virtual void fly() = 0;   // Turkeys can fly (a short distance)
};

class WildTurkey : public Turkey
{
public:
    void gobble() override { /* */ }
```



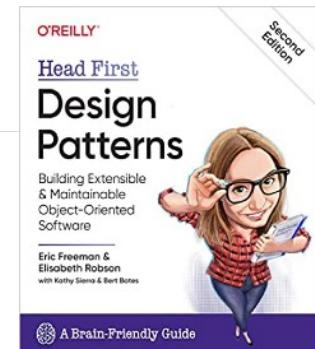
# Adapter Used For Duck-Typing

```
class Duck
{
public:
    virtual ~Duck() = default;
    virtual void quack() = 0;
    virtual void fly() = 0;
};

class MallardDuck : public Duck
{
public:
    void quack() override { /*...*/ }
    void fly() override { /*...*/ }
};

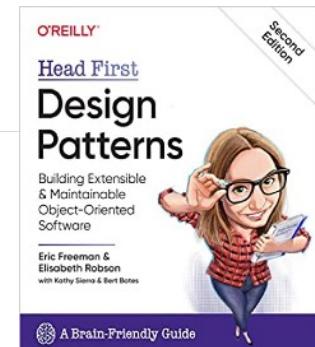
class Turkey
{
public:
    virtual ~Turkey() = default;
    virtual void gobble() = 0; // Turkeys don't quack, they gobble!
    virtual void fly() = 0;   // Turkeys can fly (a short distance)
};

class WildTurkey : public Turkey
{
public:
    void gobble() override { /* */ }
```



# Adapter Used For Duck-Typing

```
public:  
    void quack() override { /*...*/ }  
    void fly() override { /*...*/ }  
};  
  
class Turkey  
{  
public:  
    virtual ~Turkey() = default;  
    virtual void gobble() = 0; // Turkeys don't quack, they gobble!  
    virtual void fly() = 0; // Turkeys can fly (a short distance)  
};  
  
class WildTurkey : public Turkey  
{  
public:  
    void gobble() override { /*...*/ }  
    void fly() override { /*...*/ }  
};  
  
class TurkeyAdapter : public Duck  
{  
private:  
    std::unique_ptr<Turkey> turkey_;  
  
public:  
    explicit TurkeyAdapter( std::unique_ptr<Turkey>&& turkey )  
        : turkey_(std::move(turkey))
```



# Adapter Used For Duck-Typing

```

public:
    virtual ~Turkey() = default;
    virtual void gobble() = 0; // Turkeys don't quack, they gobble
    virtual void fly() = 0;   // Turkeys can fly (a short distance)
};

class WildTurkey : public Turkey
{
public:
    void gobble() override { /*...*/ }
    void fly() override { /*...*/ }
};

class TurkeyAdapter : public Duck
{
private:
    std::unique_ptr<Turkey> turkey_; ← Object Adapter
public:
    explicit TurkeyAdapter( std::unique_ptr<Turkey>&& turkey )
        : turkey_{std::move(turkey)} {}
    void quack() override { turkey_->gobble(); }
    void fly() override { turkey_->fly(); }
};

```



# Adapter Used For Duck-Typing

```
public:
    virtual ~Turkey() = default;
    virtual void gobble() = 0; // Turkeys don't quack, they gobble
    virtual void fly() = 0;   // Turkeys can fly (a short distance)
};
```

```
class WildTurkey : public Turkey
{
public:
    void gobble() override { /*...*/ }
    void fly() override { /*...*/ }
};
```

```
class TurkeyAdapter : public Duck, private WildTurkey
{
```

```
public:
    explicit TurkeyAdapter( /*WildTurkey arguments*/ )
        : WildTurkey{/*WildTurkey arguments*/}
    {}

    void quack() override { WildTurkey::gobble(); }
    void fly() override { WildTurkey::fly(); }
};
```

Class Adapter



## 5 Reasons to Use a Class Adapter

---

- You need to override a **virtual function** because you want to customize behavior or because the base class is abstract
- You need access to a **protected member function** (!) or constructor
- You need to **construct the used object before**, or destroy it after, another base subobject
- You need to share a **common virtual base class** or override the construction of a virtual base class
- You benefit substantially from the **empty base optimization (EBO)**

# The Classic Adapter Design Pattern

---

**Task (`C_Modern_Cpp_Design_Patterns/Adapter`):** Discuss the advantages and disadvantages of the given implementation of the classic Adapter design pattern.

# Examples from the STL

---

## std::stack

Defined in header `<stack>`

```
template<
    class T,
    class Container = std::deque<T>
> class stack;
```

The `std::stack` class is a container adapter that gives the programmer the functionality of a stack - specifically, a LIFO (last-in, first-out) data structure.

The class template acts as a wrapper to the underlying container - only a specific set of functions is provided. The stack pushes and pops the element from the back of the underlying container, known as the top of the stack.

## std::queue

Defined in header `<queue>`

```
template<
    class T,
    class Container = std::deque<T>
> class queue;
```

The `std::queue` class is a container adapter that gives the programmer the functionality of a queue - specifically, a FIFO (first-in, first-out) data structure.

The class template acts as a wrapper to the underlying container - only a specific set of functions is provided. The queue pushes the elements on the back of the underlying container and pops them from the front.

# Adapter

---

The Adapter design pattern ...

- ... helps to adapt an interface to an existing interface;
- ... separates interface from implementation (SRP);
- ... is non-intrusive;
- ... allows the addition of external/third-party types (OCP).

# Guidelines

---

**Guideline:** Use the Adapter design pattern to adapt any type to an existing interface.

**Guideline:** Beware the Liskov Substitution Principle when creating an adapter.

**Guideline:** Prefer containment (composition/aggregation) to inheritance.

## 2.8. Prototype

---

# Motivation

```
//---- <Animal.h> -----
class Animal
{
public:
    virtual ~Animal() = default;
    virtual void makeSound() const = 0;

    // ... more animal-specific functions
};

//---- <Sheep.h> -----
#include <Animal.h>

class Sheep : public Animal
{
public:
    void makeSound() const override { std::cout << "baa\n"; };

    // ... more animal-specific functions
};

//---- <Main.cpp> -----
#include <Sheep.h>
#include <cstdlib>
#include <memory>
```

# Motivation

```
class Sheep : public Animal
{
public:
    void makeSound() const override { std::cout << "baa\n"; }

    // ... more animal-specific functions
};

//---- <Main.cpp> -----
#include <Sheep.h>
#include <cstdlib>
#include <memory>

int main()
{
    // Creating the one and only Dolly
    std::unique_ptr<Animal> const dolly = std::make_unique<Sheep>( "Dolly" );

    // Triggers Dolly's beastly sound
    dolly->makeSound();

    // Copying Dolly
    // ????

    return EXIT_SUCCESS;
}
```

# Motivation

```
//---- <Animal.h> -----
class Animal
{
public:
    virtual ~Animal() = default;
    virtual void makeSound() const = 0;

    // ... more animal-specific functions
};

//---- <Sheep.h> -----
#include <Animal.h>

class Sheep : public Animal
{
public:
    void makeSound() const override { std::cout << "baa\n"; };

    // ... more animal-specific functions
};

//---- <Main.cpp> -----
#include <Sheep.h>
#include <cstdlib>
#include <memory>
```

# Motivation

```
//---- <Animal.h> -----
class Animal
{
public:
    virtual ~Animal() = default;
    virtual void makeSound() const = 0;
    virtual std::unique_ptr<Animal> clone() const = 0;
    // ... more animal-specific functions
};

//---- <Sheep.h> -----
#include <Animal.h>

class Sheep : public Animal
{
public:
    void makeSound() const override { std::cout << "baa\n"; }
    std::unique_ptr<Animal> clone() const override
    {
        return std::make_unique<Sheep>(*this);
    }
    // ... more animal-specific functions
};

//---- <Main.cpp> -----
```

# Motivation

```
public:  
void makeSound() const override { std::cout << "baa\n"; };  
std::unique_ptr<Animal> clone() const override  
{  
    return std::make_unique<Sheep>(*this);  
}  
// ... more animal-specific functions  
};  
  
//---- <Main.cpp> -----  
  
#include <Sheep.h>  
#include <cstdlib>  
#include <memory>  
  
int main()  
{  
    // Creating the one and only Dolly  
    std::unique_ptr<Animal> const dolly = std::make_unique<Sheep>( "Dolly" );  
  
    // Triggers Dolly's beastly sound  
    dolly->makeSound();  
  
    // Copying Dolly  
    auto dolly2 = animal->clone();  
  
    return EXIT_SUCCESS;  
}
```

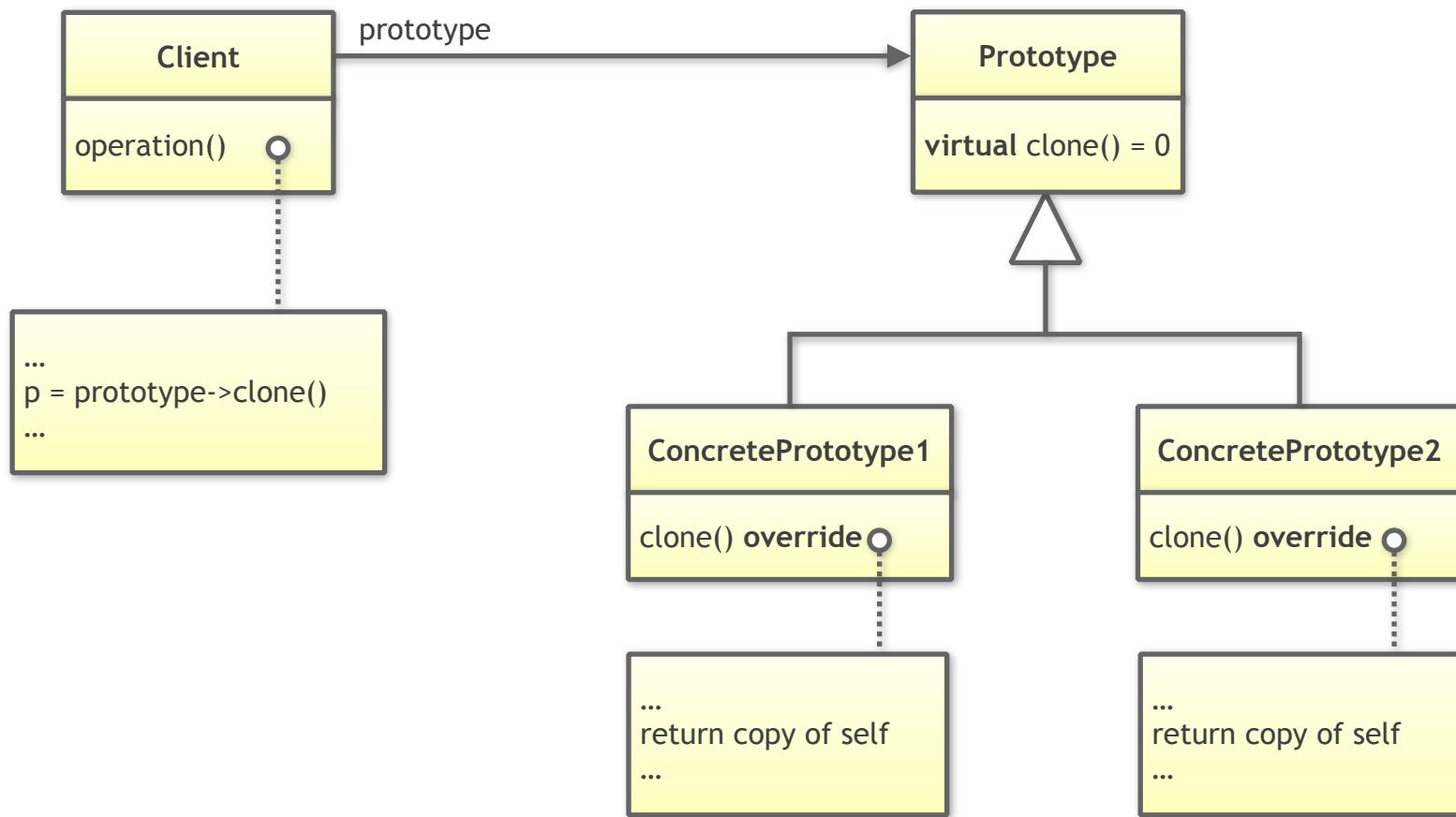
# The Intent



*"Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype."*

*(GoF, Design Patterns: Elements of Reusable Object-Oriented Software)*

# The Prototype Design Pattern

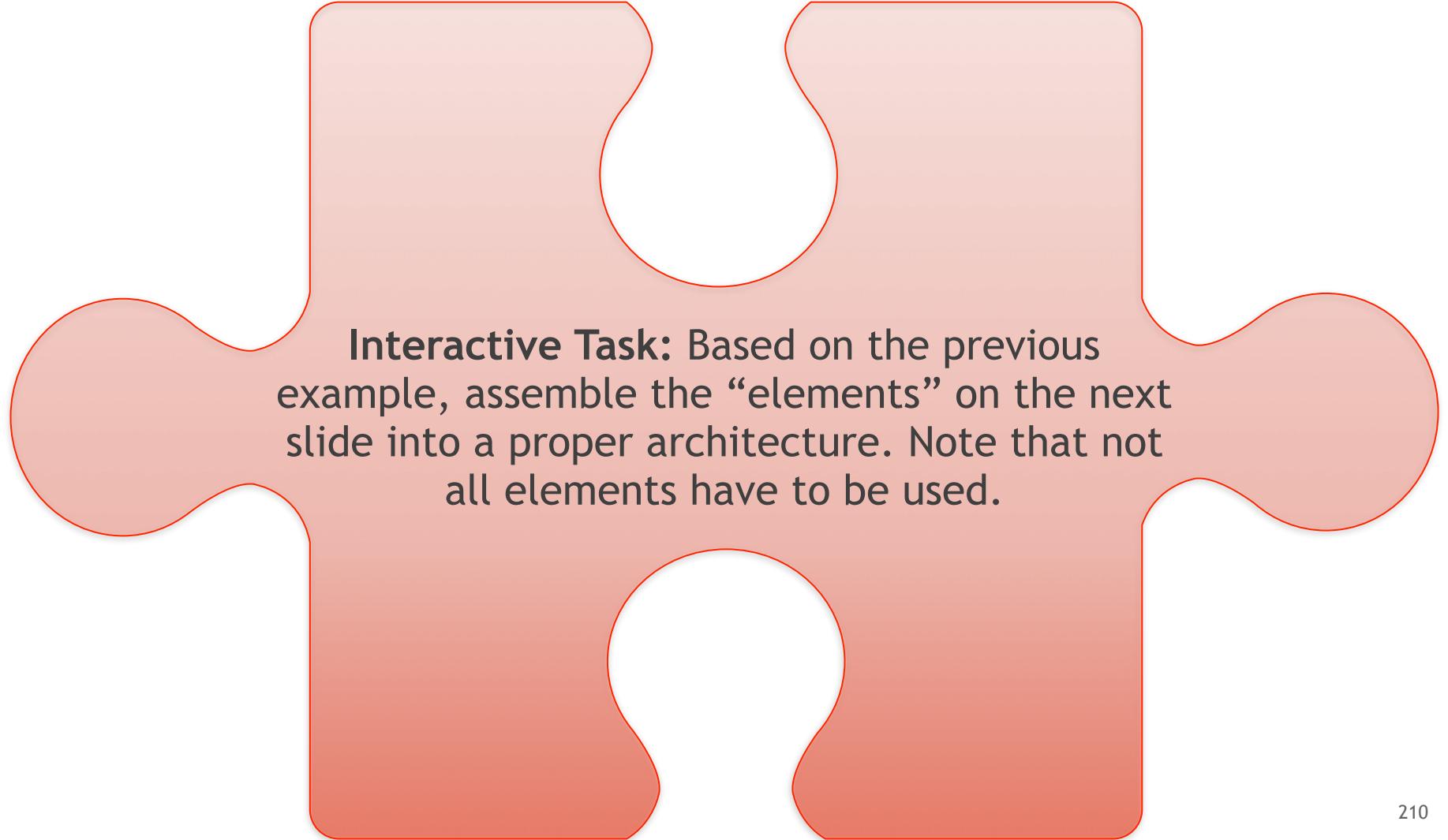


# The Classic Command Pattern

---

**Task (C\_Modern\_Cpp\_Design\_Patterns/Prototype):** Discuss the advantages and disadvantages of the given implementation of the classic Prototype design pattern.

# An Architectural Puzzle



**Interactive Task:** Based on the previous example, assemble the “elements” on the next slide into a proper architecture. Note that not all elements have to be used.

# An Architectural Puzzle

```
class Dog : public Animal
{
public:
    void makeSound() const override;
    std::unique_ptr<Animal> clone() const override;
};
```

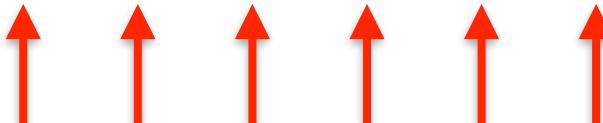
```
class Cat : public Animal
{
public:
    void makeSound() const override;
    std::unique_ptr<Animal> clone() const override;
};
```

```
class Sheep : public Animal
{
public:
    void makeSound() const override;
    std::unique_ptr<Animal> clone() const override;
};
```

```
class Animal
{
public:
    virtual ~Animal() = default;
    virtual void makeSound() const = 0;
    virtual std::unique_ptr<Animal> clone() const = 0;
};
```

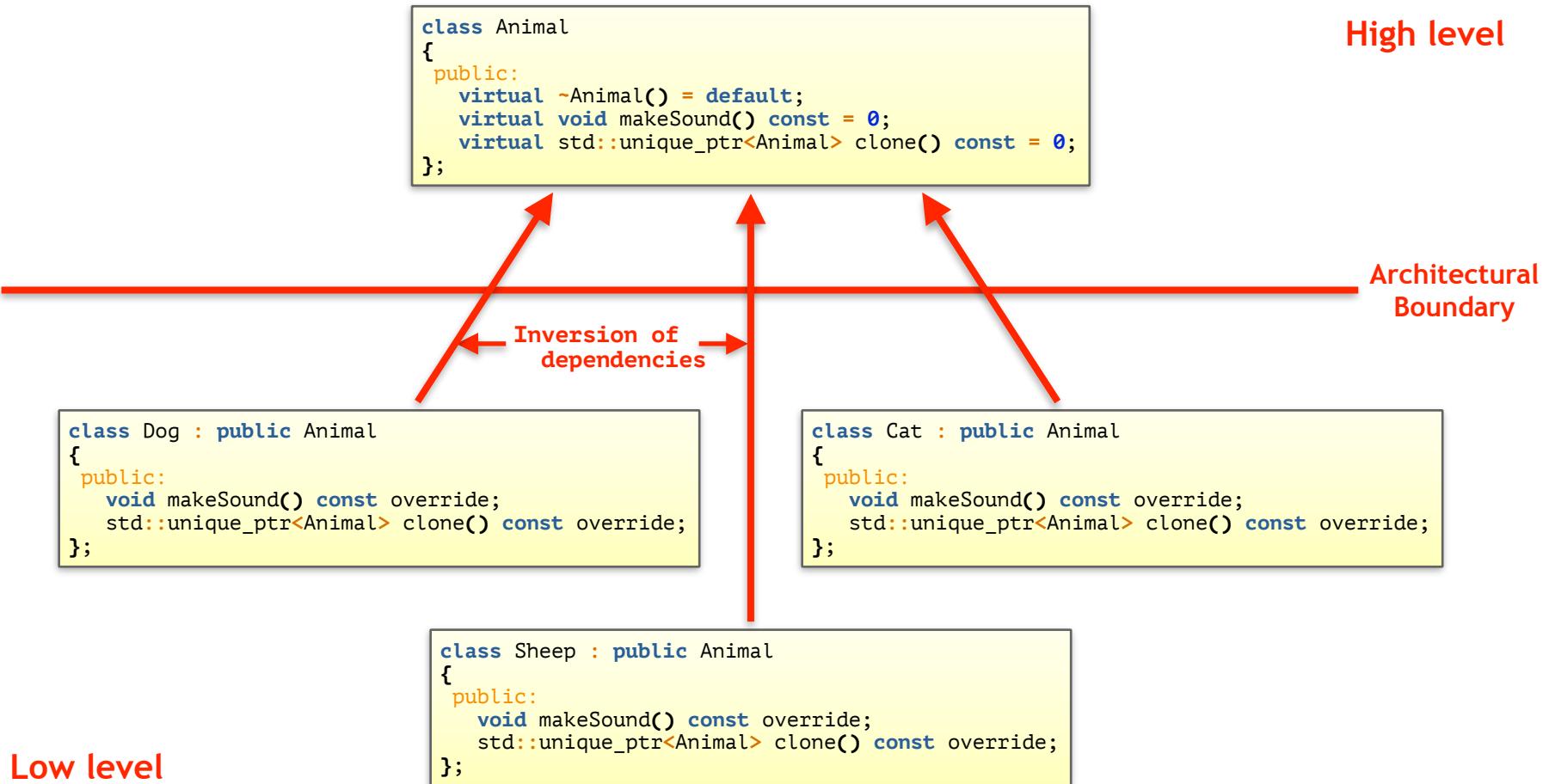
Architectural  
Boundary

Low level



High level

# An Architectural Puzzle (Solution)



# Guidelines

**Guideline:** Use the Prototype design pattern to create exact copies of an abstraction.

**Core Guideline C.130:** For making deep copies of polymorphic classes prefer a virtual `clone` function instead of copy construction/assignment.

**Core Guideline I.11:** Never transfer ownership by a raw pointer (`T*`) or reference (`T&`).

**Core Guideline F.26:** Use a `unique_ptr<T>` to transfer ownership where a pointer is needed.

# Guidelines

---

**Guideline:** Remember that the Prototype design patterns encapsulates and distributes object creation (i.e. potentially violates SRP).

## 2.9. Bridge

---

# Motivation

```
//---- <ElectricCar.h> -----

#include <Battery.h>
#include <ElectricEngine.h>
// ...

class ElectricCar
{
public:
    void drive();
    // ...
private:
    ElectricEngine engine_;
    Battery battery_;

    // ... more car-specific data members (wheels, drivetrain, ...)
};
```

This diagram illustrates a code snippet for an `ElectricCar` class. Two red arrows point from the explanatory text on the right to specific parts of the code: one arrow points to the `#include` statements at the top, and another points to the `private` section where the `ElectricEngine` and `Battery` objects are declared.

This creates a direct dependency on the `ElectricEngine` and `Battery` classes: Changes to either of them will be visible to all classes including the `<ElectricCar.h>` header.

# Motivation

```
//---- <ElectricCar.h> -----

#include <memory>
// ...
struct Battery;           // Forward declaration
struct ElectricEngine;    // Forward declaration
```

```
class ElectricCar
{
public:
    void drive();
    // ...
private:
    std::unique_ptr<ElectricEngine> engine_;
    std::unique_ptr<Battery> battery_;
    // ...
};
```

```
//---- <ElectricCar.cpp> -----
```

```
#include <ElectricEngine.h>
```

```
// ...
```



Relying on forward declarations instead removes dependencies to the implementation details of **ElectricEngine** and **Battery**, but still reveals that an **ElectricEngine** and **Battery** are used. Switching to another classes will affect all classes including the **<ElectricCar.h>** header.

# Motivation

```
//---- <Engine.h> -------

class Engine
{
public:
    virtual ~Engine() = default;
    virtual void start() = 0;
    virtual void stop() = 0;
    // ... more engine-specific functions

private:
    // ...
};

//---- <ElectricCar.h> -------

#include <Engine.h>
#include <memory>

class ElectricCar
{
public:
    void drive();
    // ...
private:
    std::unique_ptr<Engine> engine_;
    // ... same for battery

    // ... more car-specific data members (wheels, drivetrain, ...)
};
```

# Motivation

```
//---- <Engine.h> -----
class Engine
{
public:
    virtual ~Engine() = default;
    virtual void start() = 0;
    virtual void stop() = 0;
    // ... more engine-specific functions

private:
    // ...
};

//---- <ElectricCar.h> -----
#include <Engine.h>
#include <memory>

class ElectricCar
{
public:
    void drive();
    // ...
private:
    std::unique_ptr<Engine> engine_;
    // ... same for battery

    // ... more car-specific data members (wheels, drivetrain, ...)
};
```

# Motivation

```
class Engine
{
public:
    virtual ~Engine() = default;
    virtual void start() = 0;
    virtual void stop() = 0;
    // ... more engine-specific functions

private:
    // ...
};

//---- <ElectricCar.h> -------

#include <Engine.h>
#include <memory>

class ElectricCar
{
public:
    void drive();
    // ...

private:
    std::unique_ptr<Engine> engine_;
    // ... same for battery

    // ... more car-specific data members (wheels, drivetrain, ...)
};

//---- <ElectricEngine.h> -----
```

# Motivation

```
class Engine
{
public:
    virtual ~Engine() = default;
    virtual void start() = 0;
    virtual void stop() = 0;
    // ... more engine-specific functions

private:
    // ...
};

//---- <ElectricCar.h> -------

#include <Engine.h>
#include <memory>

class ElectricCar
{
public:
    void drive();
    // ...
private:
    std::unique_ptr<Engine> engine_;
    // ... same for battery

    // ... more car-specific data members (wheels, drivetrain, ...)
};
```

Introducing the **Engine** abstraction enables the **ElectricCar** class to switch to different implementations without anyone noticing. Thus we have build a **Bridge** to the implementation details.

# Motivation

```
// ...
private:
    std::unique_ptr<Engine> engine_;

    // ... more car-specific data members (wheels, drivetrain, ...)
};

//---- <ElectricEngine.h> -----
#include <Engine.h>

class ElectricEngine : public Engine
{
public:
    void start() override;
    void stop() override;

private:
    // ...
};

//---- <ElectricCar.cpp> -----
#include <ElectricCar.h>
#include <ElectricEngine.h>

ElectricCar::ElectricCar( /*maybe some engine arguments*/ )
    : engine_{ std::make_unique<ElectricEngine>( /*engine arguments*/ ) }
    // ... Initialization of the other data members
```

# Motivation

```
#include <Engine.h>

class ElectricEngine : public Engine
{
public:
    void start() override;
    void stop() override;

private:
    // ...
};

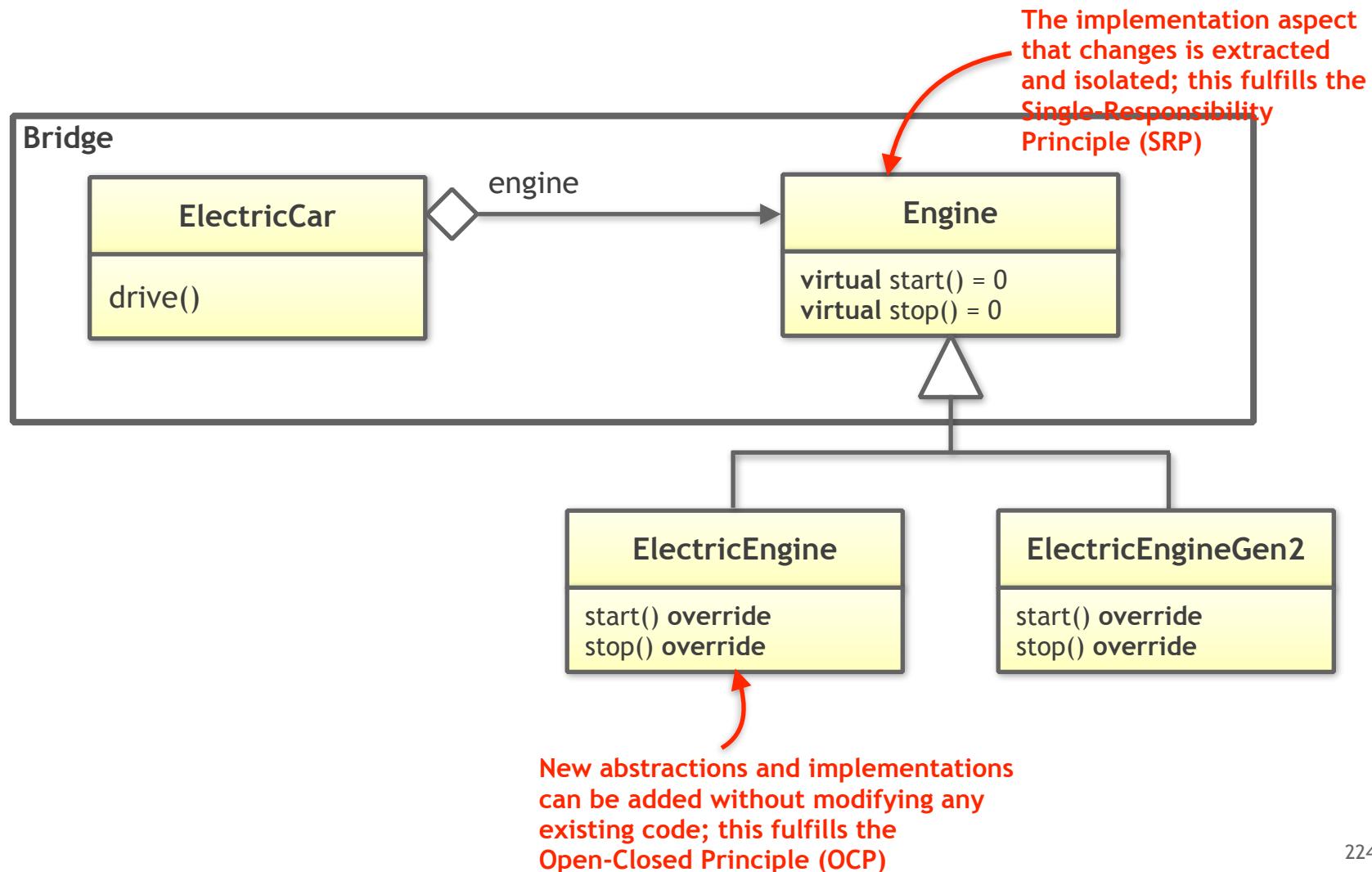
//---- <ElectricCar.cpp> -----

#include <ElectricCar.h>
#include <ElectricEngine.h>

ElectricCar::ElectricCar( /*maybe some engine arguments*/ )
    : engine_{ std::make_unique<ElectricEngine>( /*engine arguments*/ ) }
    // ... Initialization of the other data members
{}

// ... Other 'ElectricCar' member functions, primarily using the 'Engine'
// abstraction, but potentially also explicitly dealing with an
// 'ElectricEngine'.
```

# The Classic Bridge Design Pattern



# Motivation

```
//---- <Car.h> -----
#include <Engine.h>
#include <memory>
#include <utility>

class Car {
protected:
    Car( std::unique_ptr<Engine> engine )
        : engine_{ std::move(engine) }
    {}

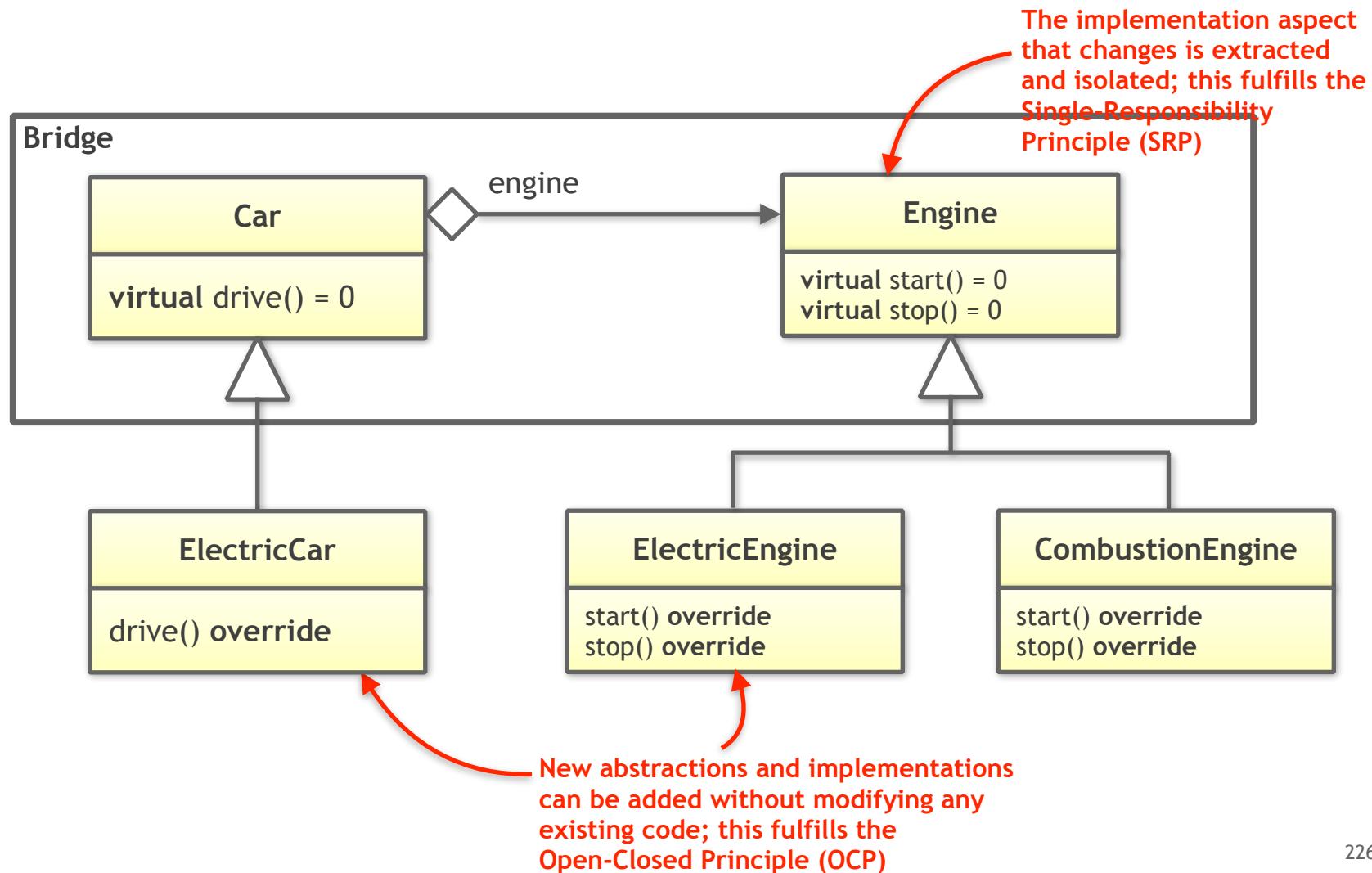
public:
    virtual ~Car() = default;
    virtual void drive() = 0;
    // ... more car-specific functions

protected:
    Engine* getEngine() { return engine_.get(); }

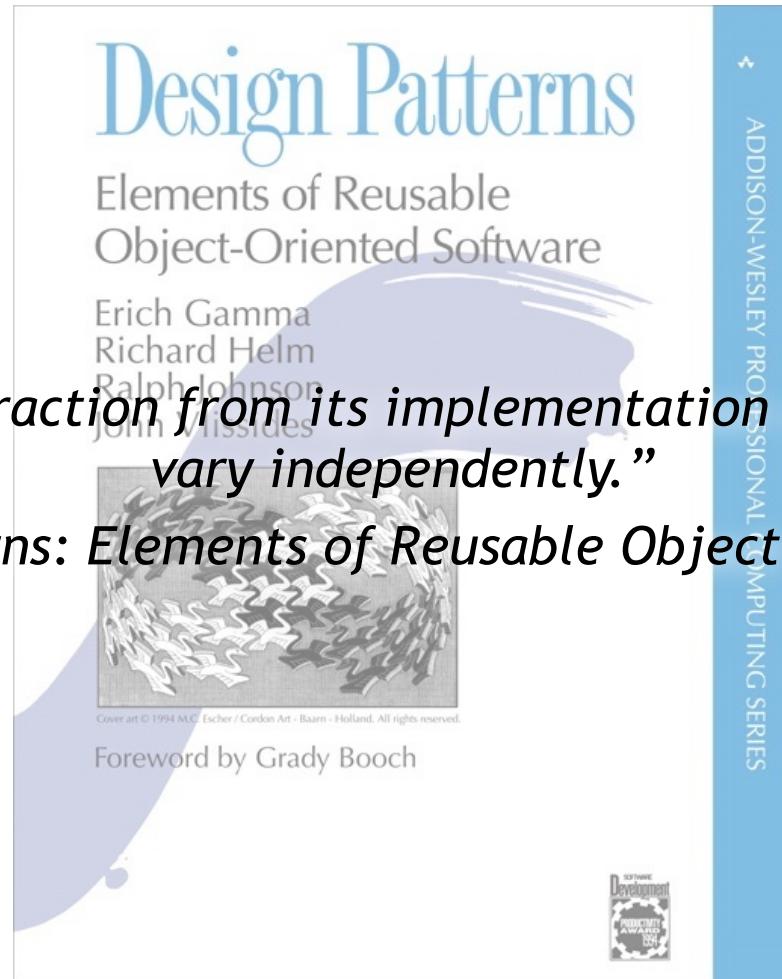
private:
    std::unique_ptr<Engine> pimpl_; // Pointer-to-implementation (pimpl)
};
```

Introducing the **Car** abstraction  
enables the reuse of the Bridge  
for many different kinds of car.

# The Classic Bridge Design Pattern



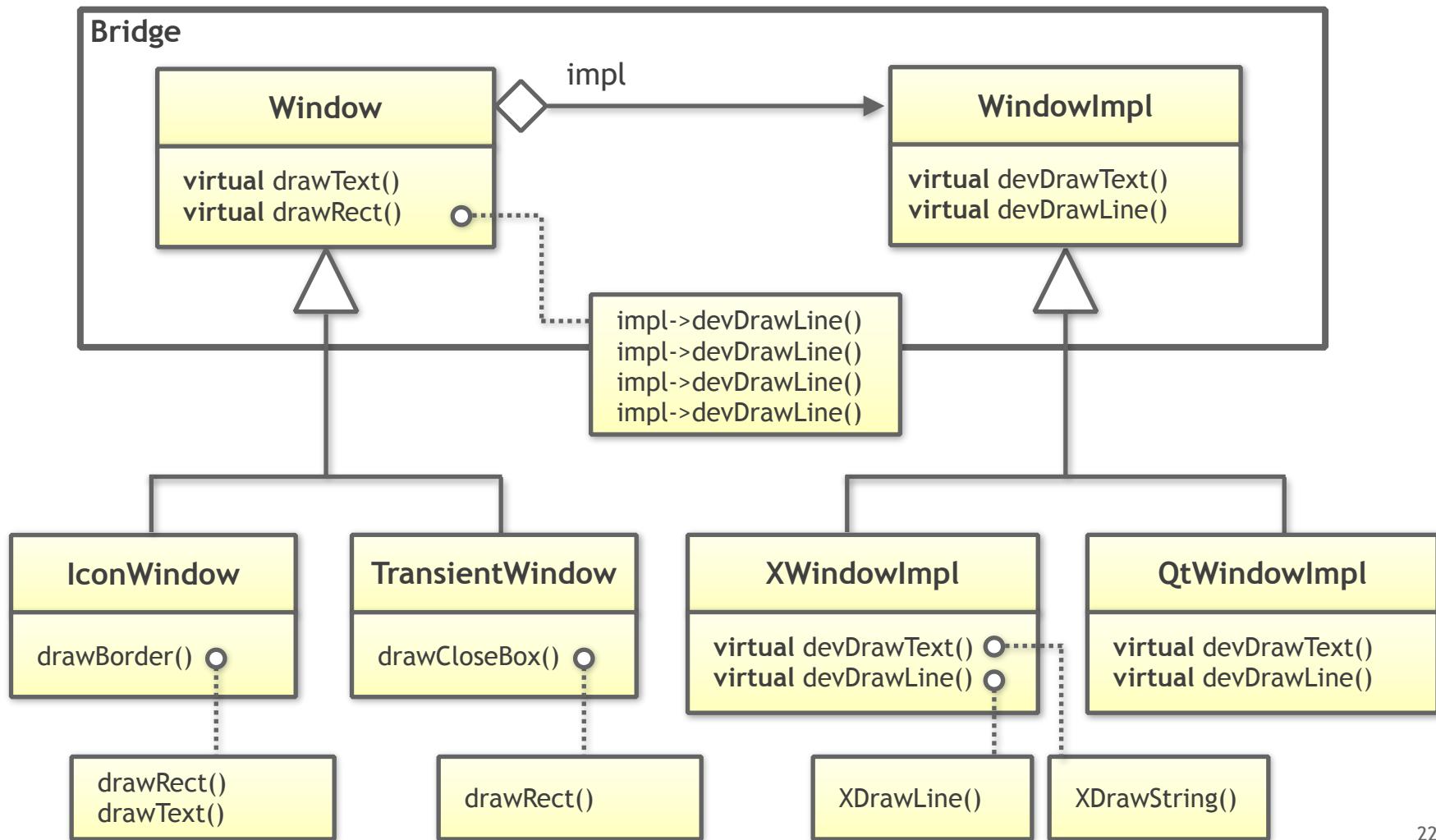
# The Intent



*"Decouple an abstraction from its implementation so that the two can vary independently."*

(GoF, *Design Patterns: Elements of Reusable Object-Oriented Software*)

# A Second Bridge Example

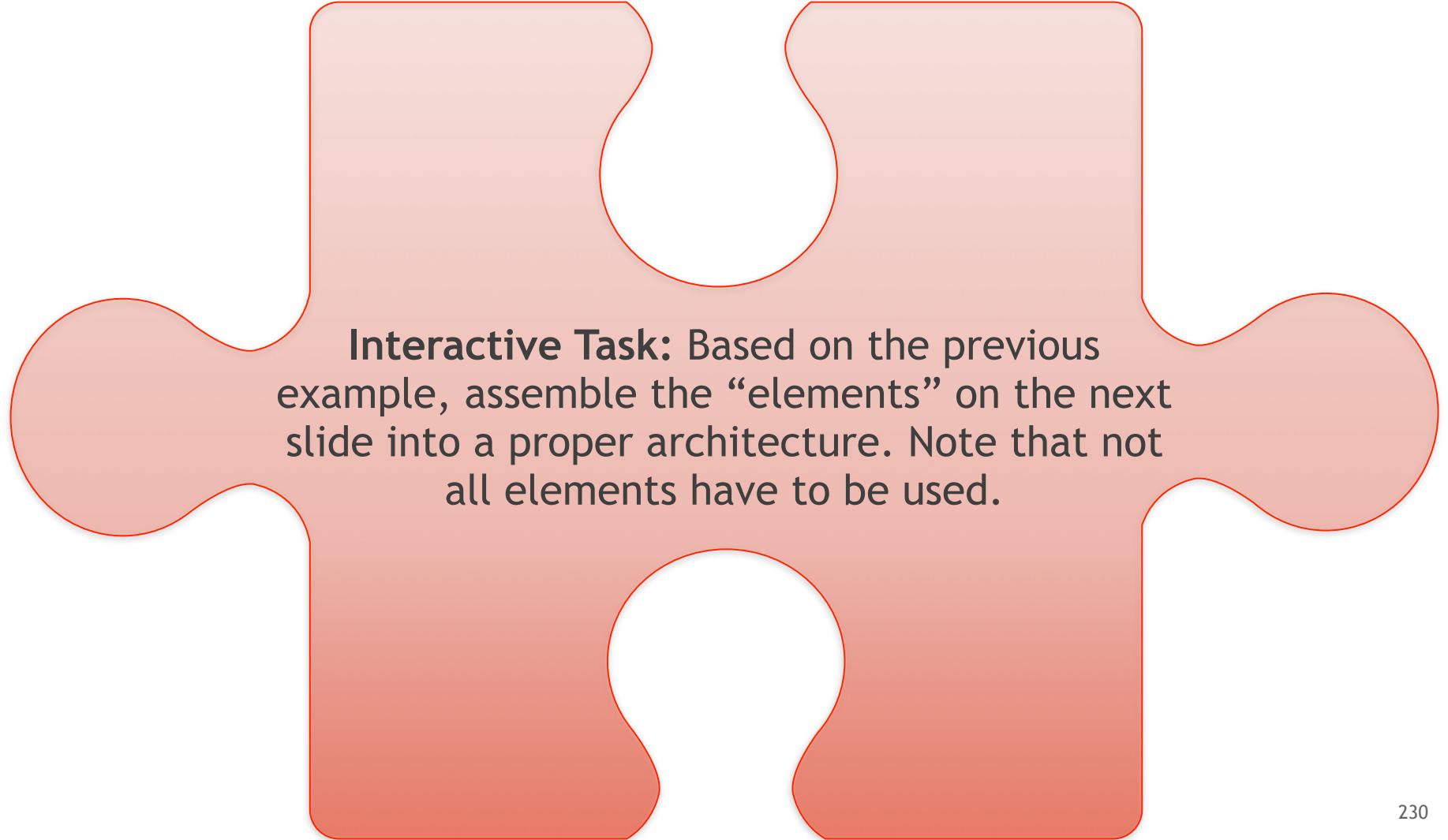


# Programming Task

---

**Task (C\_Modern\_Cpp\_Design\_Patterns/Car\_Bridge):** Consider the given Car example, which demonstrates the Bridge design pattern. What would have to be changed to implement it in terms of the Strategy design pattern?

# An Architectural Puzzle



**Interactive Task:** Based on the previous example, assemble the “elements” on the next slide into a proper architecture. Note that not all elements have to be used.

# An Architectural Puzzle

```
class Car
{
public:
    Car( std::unique_ptr<Engine> );
    virtual ~Car() = default;

    // ...

private:
    std::unique_ptr<Engine> pimpl_;
};
```

```
class ElectricCar : public Car
{
public:
    ElectricCar()
        : Car( std::make_unique<ElectricEngine>( /*...*/ ) )
    {}

    // ...
};
```

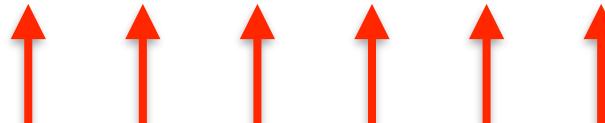
```
class Engine
{
public:
    virtual ~Engine() = default;

    virtual void start( /*...*/ ) const = 0;
    virtual void stop( /*...*/ ) const = 0;
};
```

```
class ElectricEngine : public Engine
{
public:
    void start( /*...*/ ) const override;
    void stop( /*...*/ ) const override;
};
```

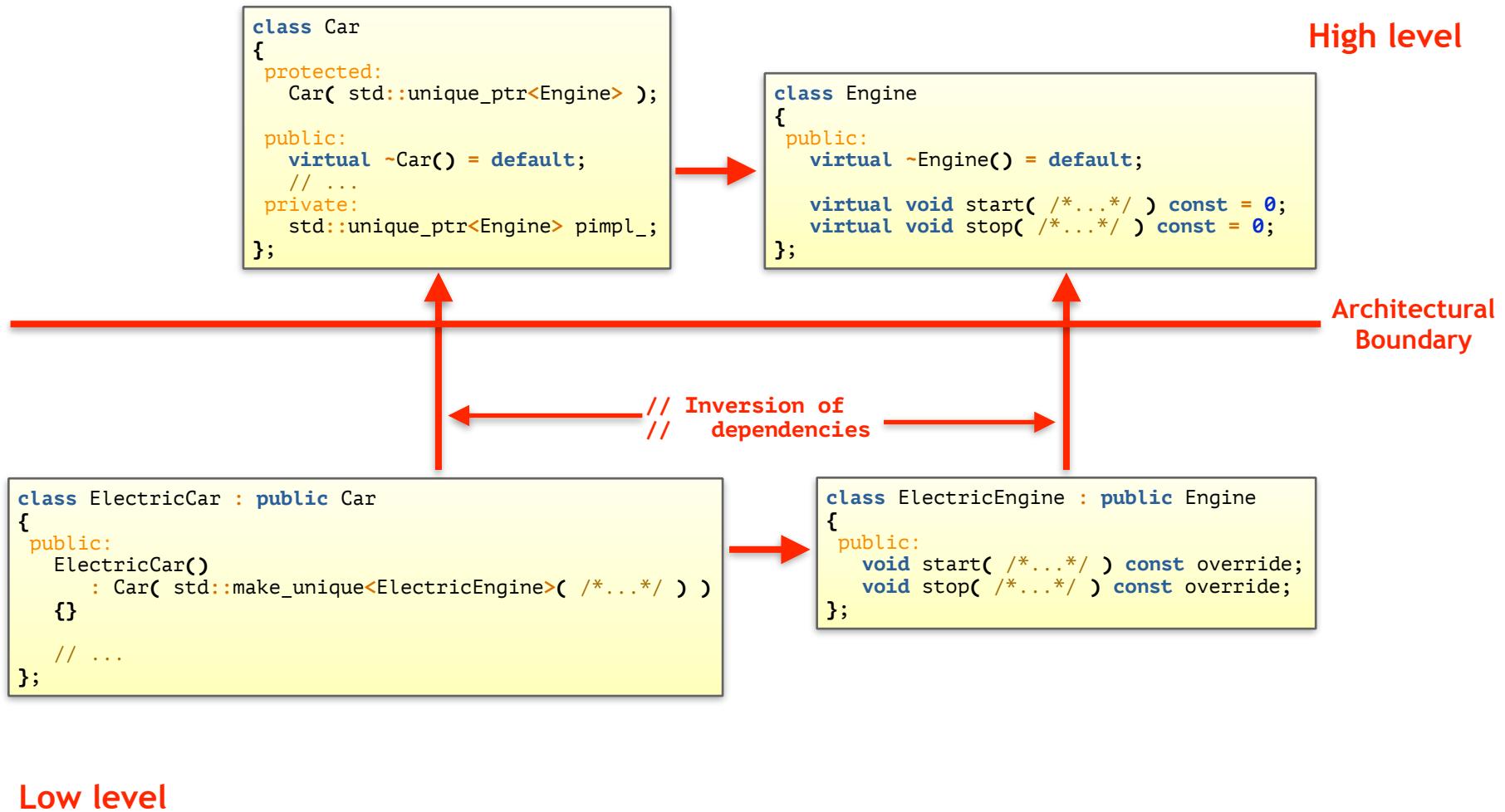
Architectural  
Boundary

Low level

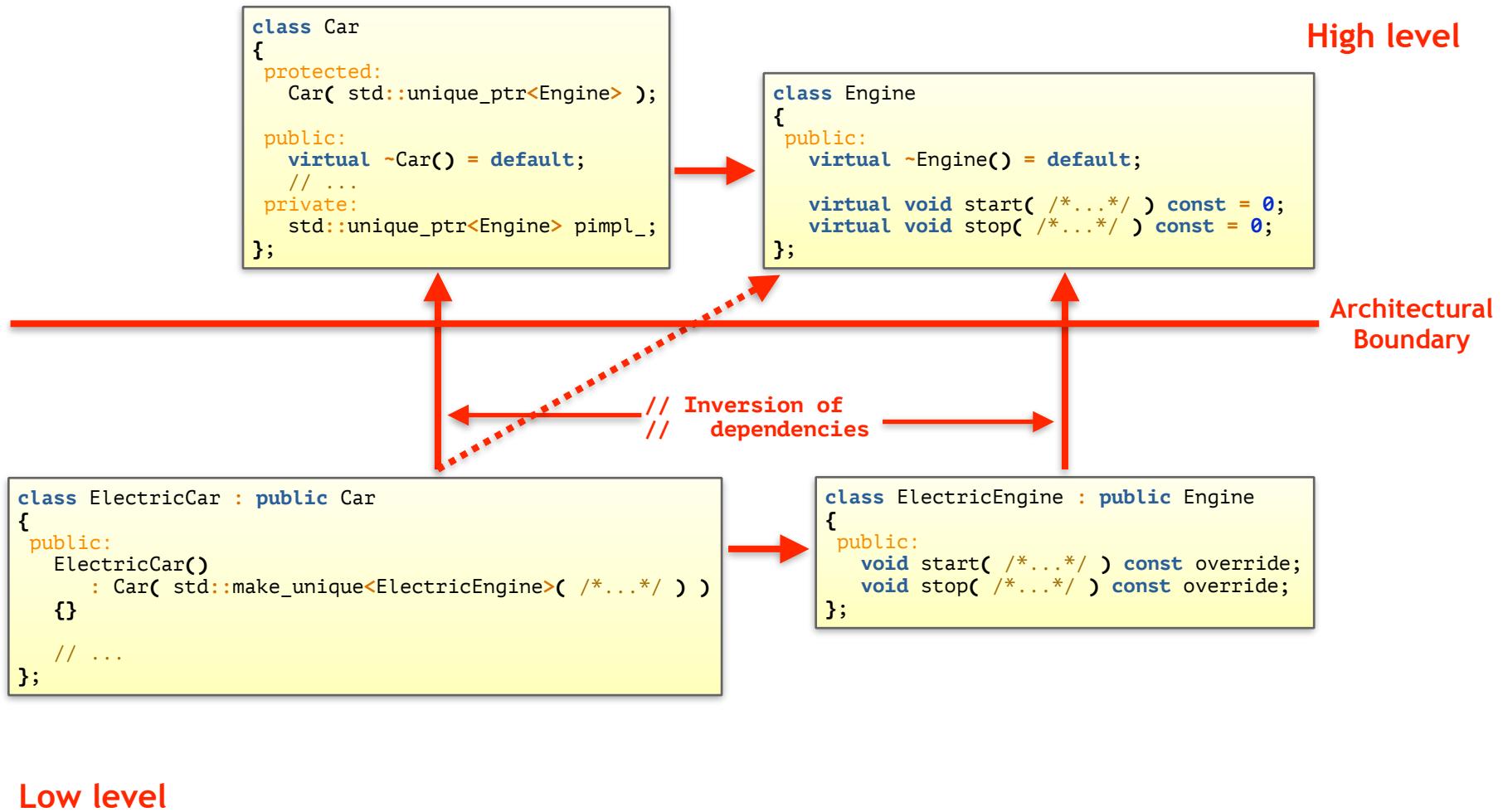


High level

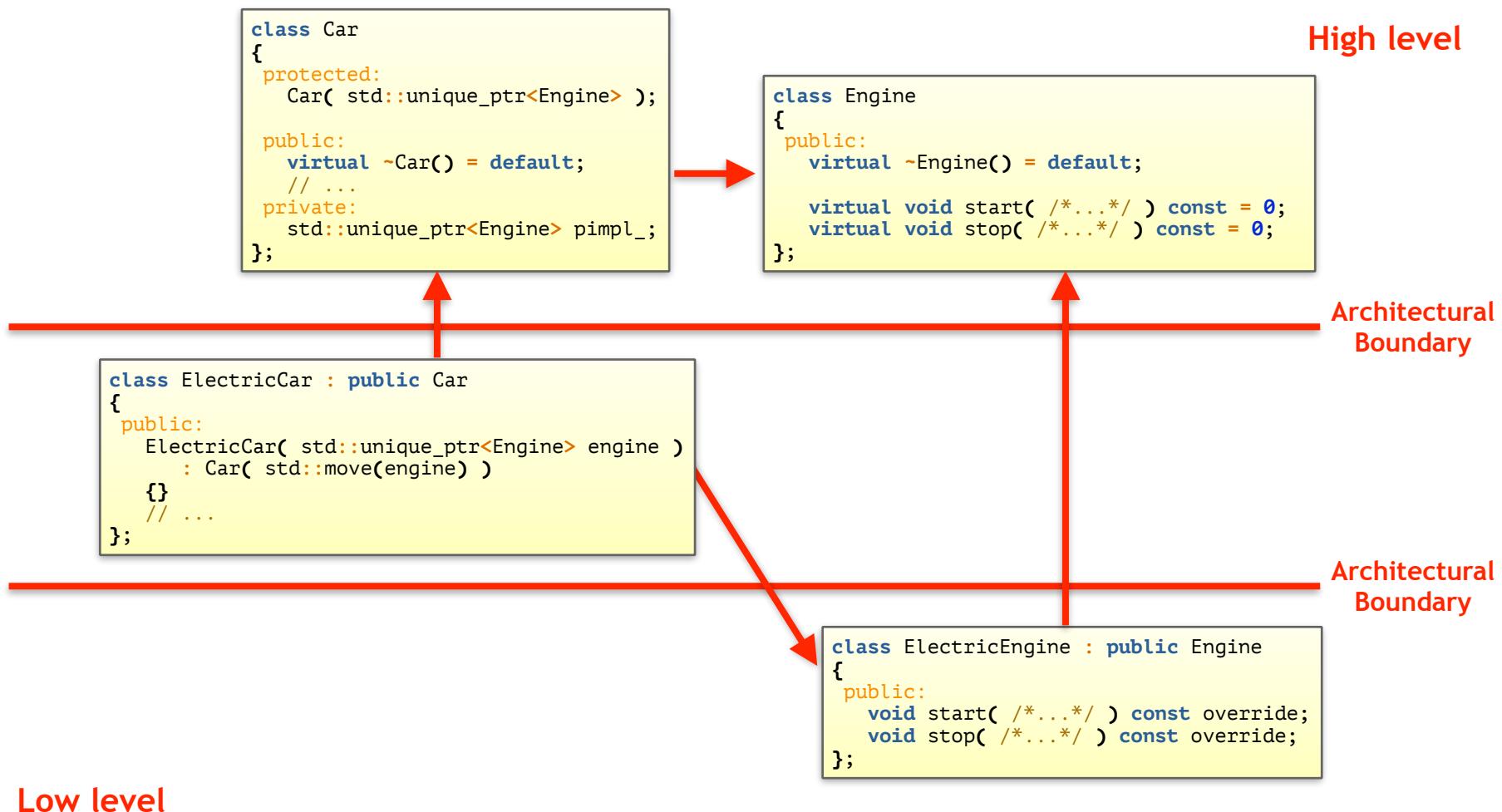
# An Architectural Puzzle (Solution)



# An Architectural Puzzle (Solution)



# Comparison to the Strategy Design Pattern



# Programming Task

## Task (C\_Modern\_Cpp\_Design\_Patterns/Bridge)

**Step 1:** In the conceptual header file <X.h>, which #include directives could be immediately removed without ill effect? You may not make any changes other than removing or rewriting #include directives.

**Step 2:** What further #includes could be removed if we made some suitable changes, and how? This time, you may make changes to X as long as X's base classes and its public interface remain unchanged; and current code that already uses X should not be affected beyond requiring a simple recompilation.

**Step 3:** Now you may make any changes to X as long as they don't change its public interface so that existing code that uses X is unaffected beyond requiring a simple recompilation. Again, note that the comment are important.

# Bridge and Performance

---

The Pimpl idiom (simplest form of the Bridge Design Pattern) can be “used” to store rarely used data members and to reduce the size of a class in order to speed up performance:

```
struct Person
{
    std::string forename{ "Homer" };
    std::string surname{ "Simpson" };
    int age{ 42 };

    struct Pimpl {
        std::string address{ "712 Red Bark Lane" };
        std::string zip{ "89011" };
        std::string city{ "Henderson" };
        std::string state{ "Nevada" };
    };

    std::unique_ptr<Pimpl> pimpl{ new Pimpl{} };
};
```

# Programming Task

---

**Task (C\_Modern\_Cpp\_Design\_Patterns/BridgedMembers):** Copy-and-paste the following code into [quick-bench.com](https://quick-bench.com). Benchmark the time to compute the total age of all persons contained in a std::vector.

# Alternative: Fast-Pimpl Idiom (Static Memory)

Instead of dynamic memory it is possible to use static memory:

```
class X : public A
{
public:
    // ...

private:
    struct XImpl;

    static constexpr size_t buffersize = 100;
    static constexpr size_t bufferalign = 16;

    using Buffer = std::aligned_storage<buffersize,bufferalign>::type;
    Buffer buffer_;

};

X::X( const C& c )
    : A{}
{
    static_assert( sizeof(XImpl) <= sizeof(buffer_) );
    static_assert( alignof(XImpl) <= alignof(Buffer) );

    new (&buffer_) XImpl{ c };
}
```

# Guidelines

---

**Guideline:** Use the Bridge design pattern to create a compilation firewall, i.e. to move dependencies from the header to the source file.

**Guideline:** Use the pimpl idiom to store rarely used members and to reduce the size of a class in order to speed up performance.

**Core Guideline I.27:** For stable library ABI, consider the Pimpl idiom

## 2.10. External Polymorphism

---

# External Polymorphism

---

[External Polymorphism \(3rd Pattern Languages of Programming Conference, September 4-6, 1996\)](#)

## External Polymorphism

An Object Structural Pattern for Transparently Extending C++ Concrete Data Types

Chris Cleeland

chris@envision.com

Envision Solutions, St. Louis, MO 63141

Douglas C. Schmidt and Timothy H. Harrison

schmidt@cs.wustl.edu and harrison@cs.wustl.edu

Department of Computer Science

Washington University, St. Louis, Missouri, 63130

This paper appeared in the Proceedings of the 3<sup>rd</sup> Pattern Languages of Programming Conference, Allerton Park, Illinois, September 4–6, 1996.

1. *Space efficiency* – The solution must not constrain the storage layout of existing objects. In particular, classes that have no virtual methods (*i.e.*, concrete data types) must not be forced to add a virtual table pointer.
2. *Polymorphism* – All library objects must be accessed in a uniform, transparent manner. In particular, if new classes are included into the system, we won’t want to change existing code.

Consider the following example using classes from the ACE network programming framework [3]:

## 1 Intent

Allow C++ classes unrelated by inheritance and/or having no virtual methods to be treated polymorphically. These unrelated classes can be treated in a common manner by software that uses them.

## 2 Motivation

Working with C++ classes from different sources can be dif-

1. `SOCK_Acceptor acceptor; // Global storage`
- 2.

# The Intent

---

### External Polymorphism

An Object Structural Pattern for Transparently Extending C++ Concrete Data Types

Chris Cleeland  
chris@envision.com

Envision Solutions, St. Louis, MO 63141

Douglas C. Schmidt and Timothy H. Harrison  
schmidt@cs.wustl.edu and harrison@cs.wustl.edu  
Department of Computer Science  
Washington University, St. Louis, Missouri, 63130

This paper appeared in the Proceedings of the 3<sup>rd</sup> Pattern Languages of Programming Conference, Alerton Park, Illinois, September 4–6, 1996.

**"Allow C++ classes unrelated by inheritance and/or having no virtual methods to be treated polymorphically. These unrelated classes can be treated in a common manner by software that uses them."**  
*(Cleeland, Schmidt and Harrison, External Polymorphism)*

**1. Intent**  
Allow C++ classes unrelated by inheritance and/or having no virtual methods to be treated polymorphically. These unrelated classes can be treated in a common manner by software that uses them.

**2. Motivation**  
Working with C++ classes from different sources can be difficult. Often an application may wish to "project" common behavior onto C++ classes, but is restricted by the classes' inheritance tree. It may be necessary to inherit from multiple classes or resolve conflicts by applying a well-known design pattern such as Adapter or Decorator [1]. Occasionally there are more complex requirements, such as the need to change both underlying interface and implementation. In such cases, classes may need to behave as if they had a common ancestor.

For instance, consider the case where we are debugging an application constructed using classes from various C++ libraries. It would be convenient to be able to ask any instance to "dump" its state in a human-readable format to a file or console display. It would be even more convenient to gather all live class instances into a collection and iterate over that collection asking each instance to dump itself.

Since collections are homogeneous, a common base class must exist to maintain a single collection. Since classes are already designed, implemented and in use, however, modifying the inheritance tree to introduce a common base class is not an option – we may not have access to the source, either! In addition, classes in OO languages like C++ may be *concrete data types* [2], which require strict storage layouts that could be compromised by hidden pointers (such as C++'s virtual table pointer). Re-implementing these classes with a common, polymorphic, base class is not feasible.

1. *Space efficiency* – The solution must not constrain the storage layout of existing objects. In particular, classes that have no virtual methods (*i.e.*, concrete data types) must not be forced to add a virtual table pointer.

2. *Polymorphism* – All library objects must be accessible uniformly through a common interface. If new objects are added to the system, we will want to change existing code.

Consider the following example using classes from the ACE network programming framework [3]:

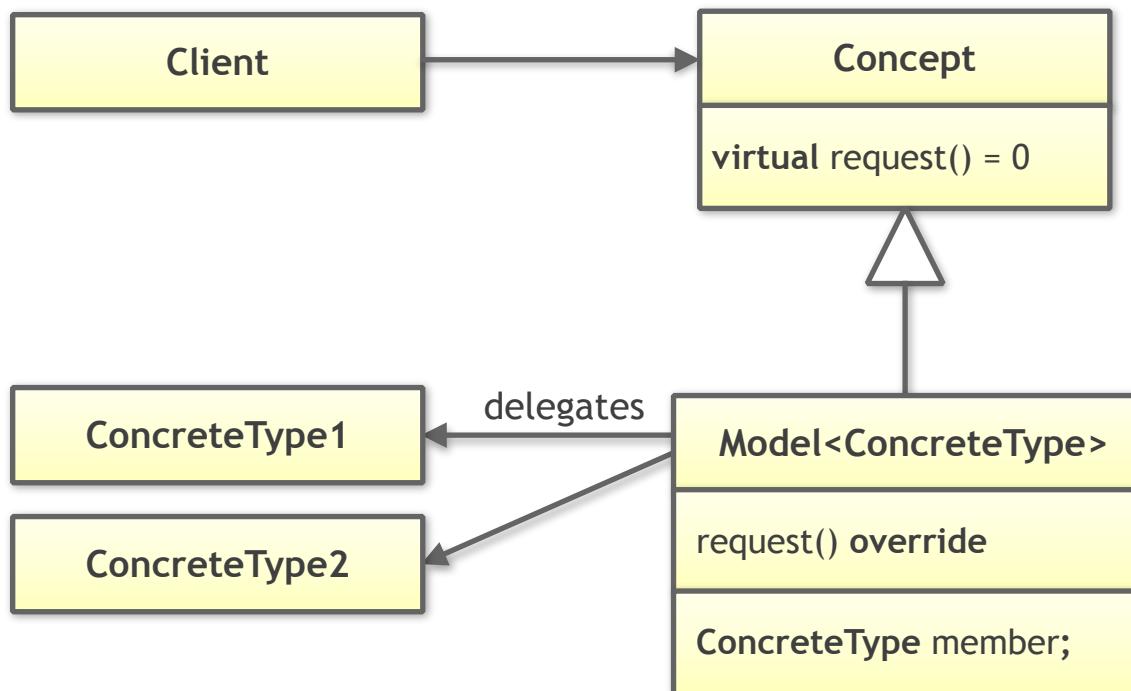
```
1. Sock_Stream::this = 0x7e393ab, handle_ = {-1}
2. SOCK_Acceptor::this = 0x2d49a45b, handle_ = {-1}
3. int main (void) {
4.     SOCK_Stream stream; // Automatic storage
5.     INET_Addr::this = 0x3c48a432,
6.     new INET_Addr::this;
7. }
```

The Sock\_Stream, SOCK\_Acceptor, and INET\_Addr classes are all concrete data types since they don't all inherit from a common ancestor and/or they don't contain virtual functions. If during a debugging session an application wanted to examine the state of all live ACE objects at line 7, we might get the following output:

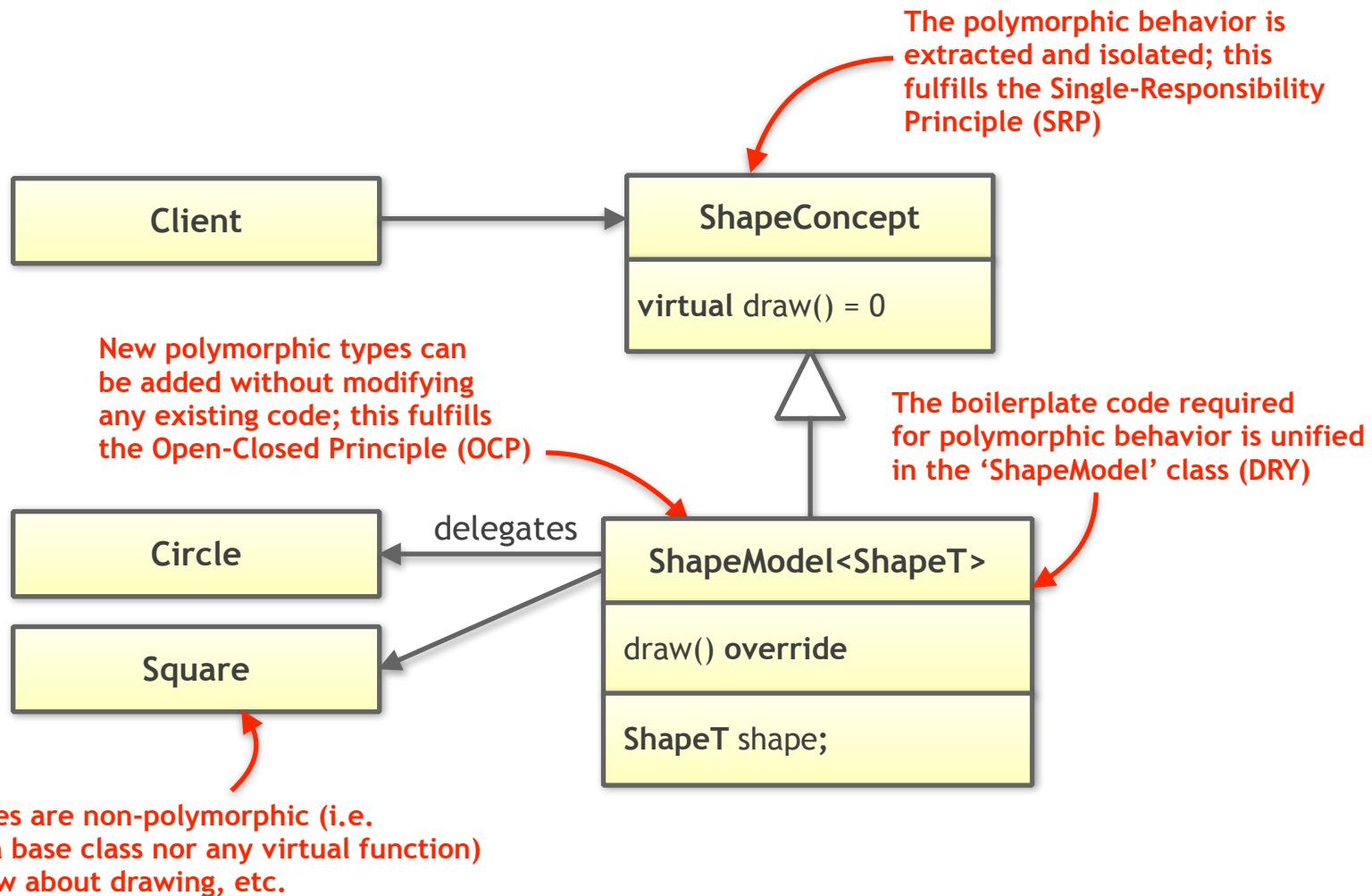
```
Sock_Stream::this = 0x7e393ab, handle_ = {-1}
SOCK_Acceptor::this = 0x2d49a45b, handle_ = {-1}
INET_Addr::this = 0x3c48a432,
port_ = {0}, addr_ = {0.0.0.0}
```

An effective way to project the capability to dump state onto these class without modifying their binary layout is to use the *External Polymorphism pattern*. This pattern constructs a parallel, external inheritance hierarchy that projects polymorphic behavior onto a set of concrete class that need not be related by inheritance. The following OMT diagram illustrates how the External Polymorphism pattern can be used to create the external, parallel hierarchy of classes:

# The External Polymorphism Design Pattern



# The External Polymorphism Design Pattern



# External Polymorphism

---

**Task (C\_Modern\_Cpp\_Design\_Patterns/ExternalPolymorphism):**  
Refactor the given Strategy-based solution and extract the polymorphic behavior of all shapes by means of the External Polymorphism design pattern. Note that the general behavior should remain unchanged.

# A Solution based on External Polymorphism

```
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(), ...
```

# A Solution based on External Polymorphism

```
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(), ...
```

# A Solution based on External Polymorphism

```
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 ShapeConcept  
{  
public:  
    virtual ~ShapeConcept() = default;  
    virtual void draw() const = 0;  
};
```

# A Solution based on External Polymorphism

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

    virtual void draw() const = 0;
};

template< typename ShapeT >
class ShapeModel : public ShapeConcept
{
public:
    using DrawStrategy = std::function<void(ShapeT const&)>

    ShapeModel( ShapeT const& shape, DrawStrategy strategy )
        : shape_{ shape }
        , strategy_{ std::move(strategy) }
    {
        if( !strategy_ ) {
            throw std::invalid_argument( "Invalid draw strategy" );
        }
    }

    void draw() const override { strategy_(shape_); }

private:
    ShapeT shape_;
    DrawStrategy strategy_;
};
```

# A Solution based on External Polymorphism

```
{  
public:  
    virtual ~ShapeConcept() = default;  
  
    virtual void draw() const = 0;  
};  
  
  
template< typename ShapeT >  
class ShapeModel : public ShapeConcept  
{  
public:  
    using DrawStrategy = std::function<void(ShapeT const&)>;  
  
    ShapeModel( ShapeT const& shape, DrawStrategy strategy )  
        : shape_{ shape }  
        , strategy_{ std::move(strategy) }  
    {  
        if( !strategy_ ) {  
            throw std::invalid_argument( "Invalid draw strategy" );  
        }  
    }  
  
    void draw() const override { strategy_(shape_); }  
  
private:  
    ShapeT shape_;  
    DrawStrategy strategy_;  
};
```

# A Solution based on External Polymorphism

```
void draw() const override { strategy_(shape_); }

private:
    ShapeT shape_;
    DrawStrategy strategy_;
};

class OpenGLStrategy
{
public:
    explicit OpenGLStrategy( /*...*/ ) {}

    void operator()( Circle const& circle ) const;

    void operator()( Square const& square ) const;

private:
    // ... OpenGL specific data members
};

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

# A Solution based on External Polymorphism

```
public:  
    explicit OpenGLStrategy( /*...*/ ) {}  
  
    void operator()( Circle const& circle ) const;  
  
    void operator()( Square const& square ) const;  
  
private:  
    // ... OpenGL specific data members  
};  
  
  
void drawAllShapes( std::vector<ShapeConcept> const& shapes )  
{  
    for( auto const& shape : shapes )  
    {  
        shape->draw();  
    }  
}  
  
  
int main()  
{  
    using Shapes = std::vector<ShapeConcept>;  
  
    // Creating some shapes  
    shapes.emplace_back(  
        std::make_unique<ShapeModel<Circle>>(  
            Circle{2.3},  
            TestDrawStrategy(Color::red) ) );
```

# A Solution based on External Polymorphism

```
        shape->draw();
    }
}

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

    // Creating some shapes
    shapes.emplace_back(
        std::make_unique<ShapeModel<Circle>>(
            Circle{2.3},
            TestDrawStrategy(Color::red) ) );
    shapes.emplace_back(
        std::make_unique<ShapeModel<Square>>(
            Square{1.2},
            TestDrawStrategy(Color::green) ) );
    shapes.emplace_back(
        std::make_unique<ShapeModel<Circle>>(
            Circle{4.1},
            TestDrawStrategy(Color::blue) ) );

    drawAllShapes( shapes );

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

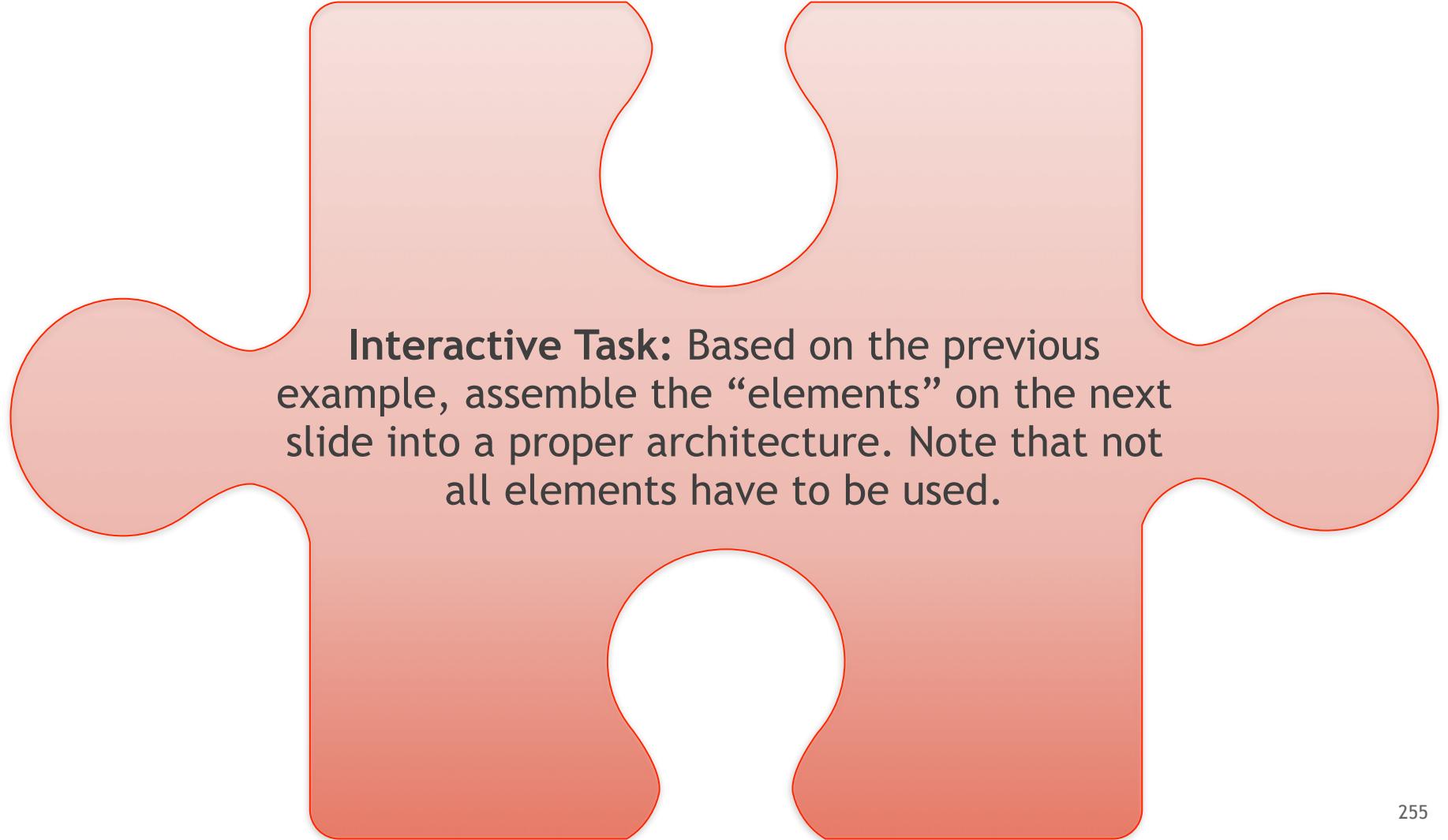
# External Polymorphism

---

The External Polymorphism design pattern ...

- ... helps to extract polymorphic behavior from a type (SRP);
- ... reduces the boilerplate code for polymorphic behavior (DRY);
- ... allows the addition of non-polymorphic types (OCP);
- ... is non-intrusive;
- ... isolates concrete types (e.g. Circle, Square, ...) from their operations (affordances).

# An Architectural Puzzle



**Interactive Task:** Based on the previous example, assemble the “elements” on the next slide into a proper architecture. Note that not all elements have to be used.

# An Architectural Puzzle (Solution)

```
class ShapeConcept
{
public:
    virtual ~ShapeConcept() = default;
    virtual void draw() const = 0;
};
```

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

```
template< typename ShapeT >
class ShapeModel : public ShapeConcept
{
public:
    using DrawStrategy = std::function<void(ShapeT const&)>;
    void draw() const override;
    // ...
private:
    ShapeT shape_;
    DrawStrategy strategy_;
};
```

```
class Circle
{
public:
    Circle( double radius );
    // ...
};
```

```
class Square
{
public:
    Square( double side );
    // ...
};
```

Architectural  
Boundary

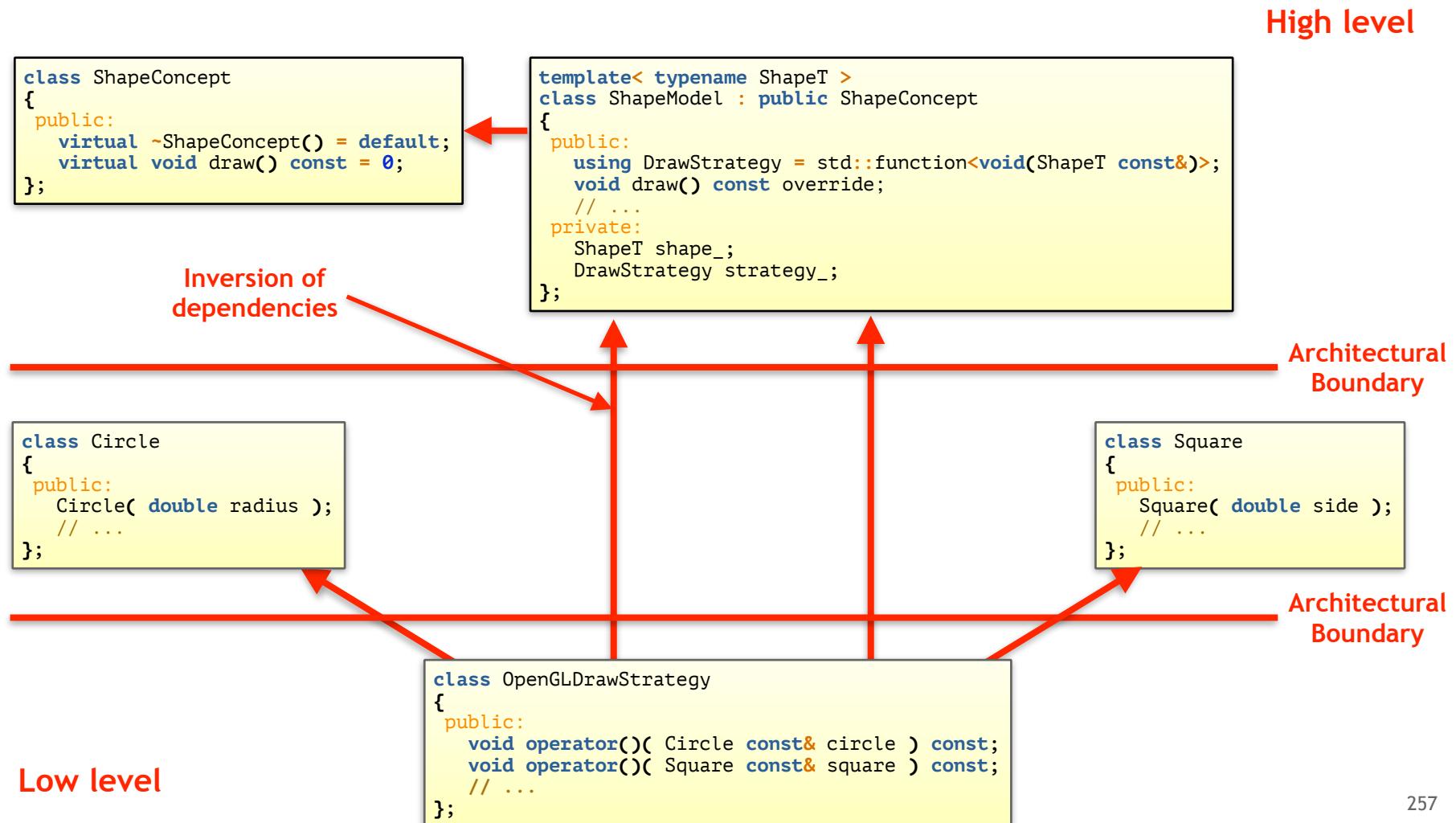
Architectural  
Boundary

Low level

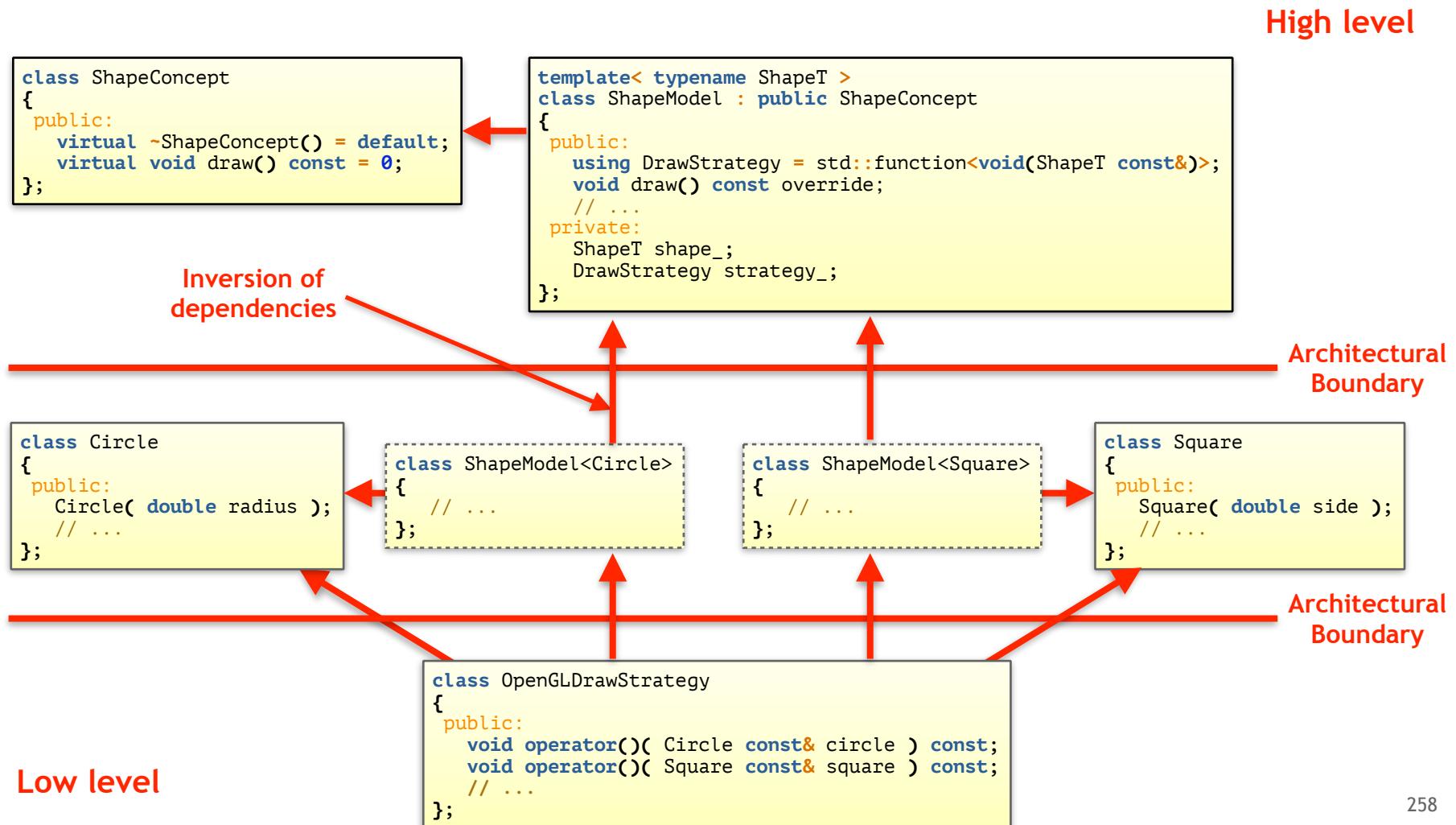


High level

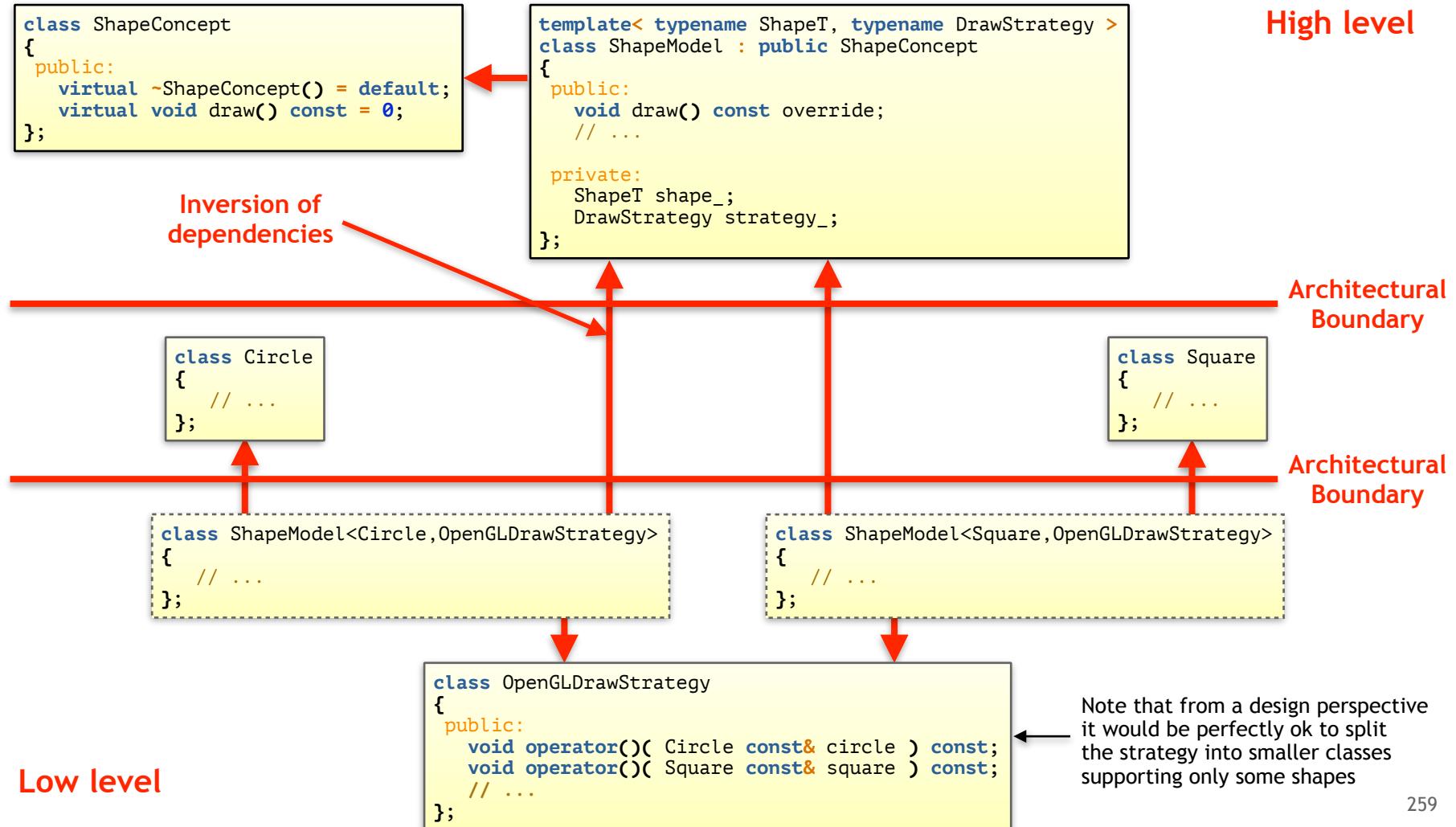
# An Architectural Puzzle (Solution)



# An Architectural Puzzle (Solution)



# An Architectural Puzzle (Solution)



Low level

High level

Architectural Boundary

Architectural Boundary

# Guidelines

---

**Guideline:** Use the External Polymorphism design pattern to extract polymorphic behavior from types.

**Guideline:** Use the External Polymorphism design pattern to treat non-polymorphic types polymorphically.

## 2.11. Type Erasure

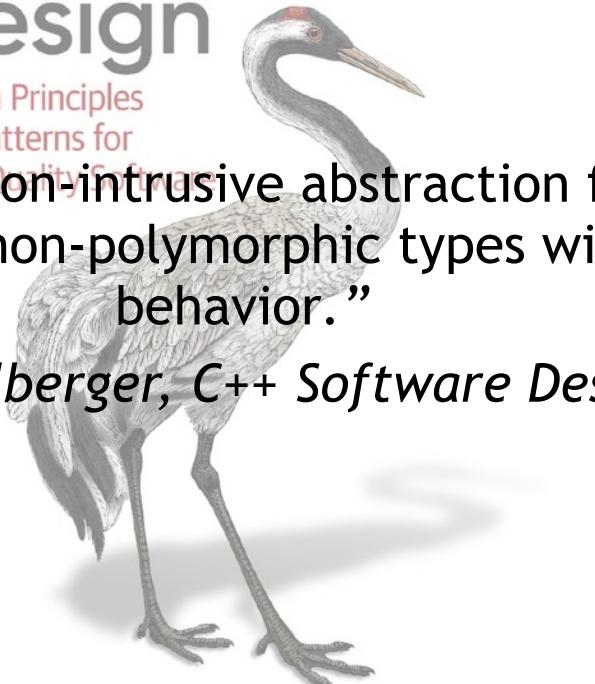
---

# The Intent

O'REILLY®

## C++ Software Design

Design Principles  
and Patterns for  
High-Quality C++



Klaus Iglberger

”Provide a value-based, non-intrusive abstraction for an extendable set of unrelated, potentially non-polymorphic types with the same semantic behavior.”

*(Klaus Iglberger, C++ Software Design)*

# Type Erasure

GoingNative 2013 Inheritance Is The Base Class of Evil

Inheritance Is The Base Class of Evil  
Sean Parent | Principal Scientist

© 2013 Adobe Systems Incorporated. All Rights Reserved.

0:37 / 24:19

63

A screenshot of a video player displaying a presentation slide. The slide has a dark blue header with the text "GoingNative 2013 Inheritance Is The Base Class of Evil" and the Adobe logo. The main content area is white and features the title "Inheritance Is The Base Class of Evil" and the speaker's name "Sean Parent | Principal Scientist". Below the title is a large, abstract graphic composed of overlapping red, orange, and grey organic shapes. At the bottom of the slide is a copyright notice: "© 2013 Adobe Systems Incorporated. All Rights Reserved.". The video player interface at the bottom shows a progress bar at 0:37 / 24:19, and various control icons for play, volume, and settings. The number "63" is visible in the bottom right corner of the video frame.

# Type Erasure

---

[Valued Conversions \(C++ Report 12\(7\), July-August 2000\)](#)

---

**Kevlin Henney** is an independent consultant and trainer based in the UK. He may be contacted at [kevlin@curbralan.com](mailto:kevlin@curbralan.com).



### FROM MECHANISM TO METHOD

---

## Valued Conversions

**H**OW WOULD YOU like to pay for that?" Good question. Digging deep into pockets, wallets, and bags uncovered a wealth of possibilities, a handful of different currencies and mechanisms to choose from: credit cards, debit cards, coins, bills, and a couple of IOUs, each form in some way substitutable for another when realizing monetary value.

Cash is the simplest, least troublesome form for small amounts and quick transactions. However, sifting through the metal and paper, it seemed that my currencies were no good. Well, that's

to any concrete form of inheritance. Substitutability here is based on values and conversions between values. Sometimes the use is implicit, at other times it must be made explicit. Conversions can be fully value preserving, widening, or narrowing. Widening conversions are always safe and typically acceptable (e.g., tipping), whereas narrowing conversions may not be (e.g., shortchanging tends to lead to exceptional or even undefined behavior).

Rescuing me from further metaphor stretching, the point-of-sale system and the assistant's smile kicked into life.

# Type Erasure

<https://twitter.com/ericniebler/status/1274031123522220033>



**Eric Niebler #BLM**  
@ericniebler

...

If I could go back in time and had the power to change C++, rather than adding virtual functions, I would add language support for type erasure and concepts. Define a single-type concept, automatically generate a type-erasing wrapper for it.

7:27 PM · Jun 19, 2020 · Twitter for Android

6 Retweets 2 Quote Tweets 125 Likes



**Eric Niebler #BLM** @ericniebler · Jun 20

...

Replying to @ericniebler

This got more likes than I expected, and not one person saying "but but but my elaborate class hierarchies...." Huh.



4



15



# Type Erasure – Examples from the Standard

---

```
#include <iostream>
#include <memory>

struct Deleter
{
    template< typename T >
    void operator()( T* ptr ) {
        std::cout << "Deleting ptr " << ptr << "\n";
        delete ptr;
    }
};

int main()
{
    std::shared_ptr<int> sptr1{ new int{42}, Deleter{} };
    std::shared_ptr<int> sptr2{ sptr1 };
}
```

# Type Erasure – Examples from the Standard

```
#include <any>
#include <string>

using namespace std;

int main()
{
    any a{};

    a = 7;    // Stores the integer 7

    any b{ "Any string"s };    // Creates an any with "Any string"s

    a = b;

    const string s
        = any_cast<string>( a );    // Extracts "Any string"s
}
```

# Type Erasure – Examples from the Standard

```
#include <functional>

void foo(int i) {
    std::cout << "foo: " << i << '\n';
}

int main()
{
    std::function<void(int)> f{};

    f = [](int i){ std::cout << "lambda: " << i << '\n'; };

    auto g = f; // Value semantics: Creates a deep copy

    f = foo;

    f( 1 );
    g( 2 );
}
```

# std::function – A Simplified Implementation

---

**Task (C\_Modern\_Cpp\_Design\_Patterns/Function1):** Implement a simplified std::function to demonstrate the type erasure design pattern. Use the inheritance-based approach.

# std::function – A Simplified Implementation

---

```
template< typename Fn >
class function;
```

# std::function – A Simplified Implementation

```
template< typename Fn >
class function;

template< typename R, typename... Args >
class function<R(Args...)>
{
    // ...
};
```

# std::function – A Simplified Implementation

```
template< typename Fn >
class function;

template< typename R, typename... Args >
class function<R(Args...)>
{
    // ...
private:
    class Concept
    {
        public:
            virtual ~Concept() = default;
            virtual R operator()( Args... ) const = 0;
            virtual Concept* clone() const = 0;
    };
    // ...
};
```

# std::function – A Simplified Implementation

```
template< typename Fn >
class function;

template< typename R, typename... Args >
class function<R(Args...)>
{
    // ...
    class Concept { ... };

    // ...
    Concept* pimpl;
};
```

# std::function – A Simplified Implementation

```
template< typename Fn >
class function;

template< typename R, typename... Args >
class function<R(Args...)>
{
    // ...
    class Concept { ... };

    template< typename Fn >
    class Model : public Concept {
        explicit Model( Fn fn ) : fn_( fn ) {}
        R operator()( Args... args ) const override
            { return fn_( std::forward<Args>( args )... ); }
        Concept* clone() const override
            { return new Model( fn_ ); }
        Fn fn_;
    };
    // ...
};
```

# std::function – A Simplified Implementation

```
template< typename Fn >
class function;

template< typename R, typename... Args >
class function<R(Args...)>
{
public:
    template< typename Fn >
    function( Fn fn ) : pimpl_( new Model<Fn>( fn ) ) {}

private:
    // ...
};
```

# std::function – A Simplified Implementation

```
template< typename Fn >
class function;

template< typename R, typename... Args >
class function<R(Args...)>
{
public:
    template< typename Fn > function( Fn fn );
    function( function const& f )
        : pimpl_( f.pimpl_->clone() ) {}
    function& operator=( function f )
        { std::swap( pimpl_, f.pimpl_ ); return *this; }

private:
    // ...
};
```

# std::function – A Simplified Implementation

```
template< typename Fn >
class function;

template< typename R, typename... Args >
class function<R(Args...)>
{
public:
    template< typename Fn > function( Fn fn );
    function( function const& f );
    function& operator=( function f );
    function( function&& f ) : pimpl_( f.pimpl_ )
    {
        f.pimpl_ = nullptr;
    }
    function& operator=( function&& f )
    {
        delete pimpl_;
        pimpl_ = f.pimpl_;
        f.pimpl_ = nullptr;
        return *this;
    }

private:
    // ...
};
```

# std::function – A Simplified Implementation

```
template< typename Fn >
class function;

template< typename R, typename... Args >
class function<R(Args...)>
{
public:
    template< typename Fn > function( Fn fn );
    function( function const& f );
    function& operator=( function f );
    function( function&& f );
    Function& operator=( Function&& f );
    ~function() { delete pimpl_; }

private:
    // ...
};
```

# std::function – A Simplified Implementation

```
template< typename Fn >
class function;

template< typename R, typename... Args >
class function<R(Args...)>
{
public:
    // ...

    R operator()( Args&&... args )
        { return (*pimpl_)( std::forward<Args>( args )... ); }

private:
    // ...
};
```

# std::function – A Simplified Implementation

---

**Task (C\_Modern\_Cpp\_Design\_Patterns/Function2):** Implement a simplified std::function to demonstrate the type erasure design pattern. Use the void\*-based approach.

# Applied Type Erasure

---

**Task (C\_Modern\_Cpp\_Design\_Patterns/TypeErasure):** Implement the Shape class by means of Type Erasure. Shape may require all types to provide a free draw() function that draws them to the screen.

# A Type-Erased 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
};

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

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

    double getSide() const noexcept;
```

# A Type-Erased 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
};

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

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

    double getSide() const noexcept;
```

# A Type-Erased Solution

```
void draw( Circle const& );  
  
class Square  
{  
public:  
    explicit Square( double s )  
        : side{ s }  
        , // ... Remaining data members  
    {}  
  
    double getSide() const noexcept;  
    // ... getCenter(), getRotation(), ...  
  
private:  
    double side;  
    // ... Remaining data members  
};  
  
void translate( Square const&, Vector2D const& );  
void rotate( Square const&, double const& );  
void draw( Square const& );  
  
class Shape  
{  
private:  
    struct Concept  
    {  
        virtual ~Concept() = default;  
        virtual void doTranslate( Vector2D const& v, const int n ) = 0;
```

# A Type-Erased Solution

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

class Shape
{
private:
    struct Concept
    {
        virtual ~Concept() = default;
        virtual void do_translate( Vector2D const& v ) const = 0;
        virtual void do_rotate( double const& q ) const = 0;
        virtual void do_draw() const = 0;
        // ...
    };
};

template< typename ShapeT >
struct Model : public Concept
{
    Model( ShapeT const& value )
        : object{ value }
    {}

    void do_translate( Vector2D const& v ) const override
    {
        translate( object, v );
    }

    void do_rotate( double const& q ) const override
    {
        rotate( object, q );
    }
};
```

# A Type-Erased Solution

```
template< typename ShapeT >
struct Model : public Concept
{
    Model( ShapeT const& value )
        : object{ value }
    {}

    void do_translate( Vector2D const& v ) const override
    {
        translate( object, v );
    }

    void do_rotate( double const& q ) const override
    {
        rotate( object, q );
    }

    void do_draw() const override
    {
        draw( object );
    }

    // ...

    ShapeT object;
};

std::unique_ptr<Concept> pimpl;

friend void translate( Shape& shape, Vector2D const& v )
```

# A Type-Erased Solution

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

class Shape
{
private:
    struct Concept
    {
        virtual ~Concept() = default;
        virtual void do_translate( Vector2D const& v ) const = 0;
        virtual void do_rotate( double const& q ) const = 0;
        virtual void do_draw() const = 0;
        // ...
    };
};

template< typename ShapeT >
struct Model : public Concept
{
    Model( ShapeT const& value )
        : object{ value }
    {}

    void do_translate( Vector2D const& v ) const override
    {
        translate( object, v );
    }

    void do_rotate( double const& q ) const override
    {
        rotate( object, q );
    }
};
```

# A Type-Erased Solution

```
    ShapeT object;
};

std::unique_ptr<Concept> pimpl;

friend void translate( Shape& shape, Vector2D const& v )
{
    shape.pimpl->do_translate( v );
}

friend void rotate( Shape& shape, double const& q )
{
    shape.pimpl->do_rotate( q );
}

friend void draw( Shape const& shape )
{
    shape.pimpl->do_draw();
}

public:
    template< typename ShapeT >
    Shape( ShapeT const& shape )
        : pimpl{ new Model<ShapeT>( shape ) }
    {}

    // Special member functions
    Shape( Shape const& s );
    Shape( Shape&& s );
    Shape& operator=( Shape const& s );
    Shape& operator=( Shape&& s );
```

# A Type-Erased Solution

```
    ShapeT object;
};

std::unique_ptr<Concept> pimpl;

friend void translate( Shape& shape, Vector2D const& v )
{
    shape.pimpl->do_translate( v );
}

friend void rotate( Shape& shape, double const& q )
{
    shape.pimpl->do_rotate( q );
}

friend void draw( Shape const& shape )
{
    shape.pimpl->do_draw();
}

public:
template< typename ShapeT >
Shape( ShapeT const& shape )
    : pimpl{ new Model<ShapeT>( shape ) }
{ }

// Special member functions
Shape( Shape const& s );
Shape( Shape&& s );
Shape& operator=( Shape const& s );
Shape& operator=( Shape&& s );
```

# A Type-Erased Solution

```
    shape.pimpl->do_draw();
}

public:
    template< typename ShapeT >
    Shape( ShapeT const& shape )
        : pimpl{ new Model<ShapeT>( shape ) }
    {}

    // Special member functions
    Shape( Shape const& s );
    Shape( Shape&& s );
    Shape& operator=( Shape const& s );
    Shape& operator=( Shape&& s );

    // ...
};

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

int main()
{
    // ...
}
```

# A Type-Erased Solution

```
Shape( Shape const& s );
Shape( Shape&& s );
Shape& operator=( Shape const& s );
Shape& operator=( Shape&& s );

// ...
};

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

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 );
}
```

# A Type-Erased Solution

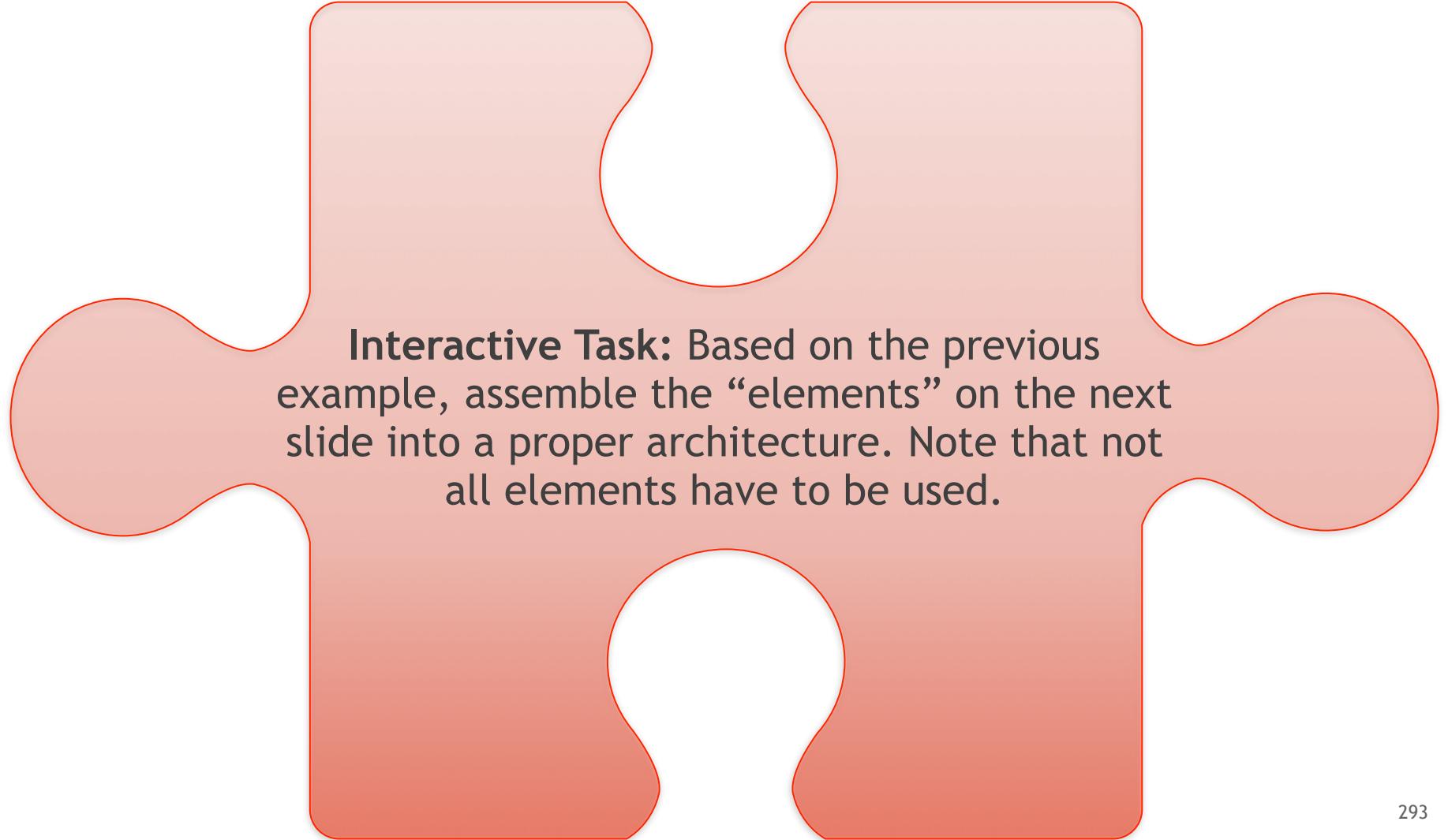
```
for( auto const& shape : shapes )
{
    draw( shape );
}

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 );
}
```

# An Architectural Puzzle



**Interactive Task:** Based on the previous example, assemble the “elements” on the next slide into a proper architecture. Note that not all elements have to be used.

# An Architectural Puzzle (Solution)

```
class Circle
{
public:
    Circle( double radius );
    // ...
};
```

```
class Shape
{
public:
    template< typename S >
    Shape( const S& );
    // ...
private:
    struct Concept;
    template< typename S > struct Model;
    // ...
    std::unique_ptr<Concept> pimpl_;
};

void draw( Shape& shape );
```

```
class Square
{
public:
    Square( double side );
    // ...
};
```

```
void draw( const Circle& circle ) const;
```

```
void draw( const Square& square ) const;
```

Architectural  
Boundary

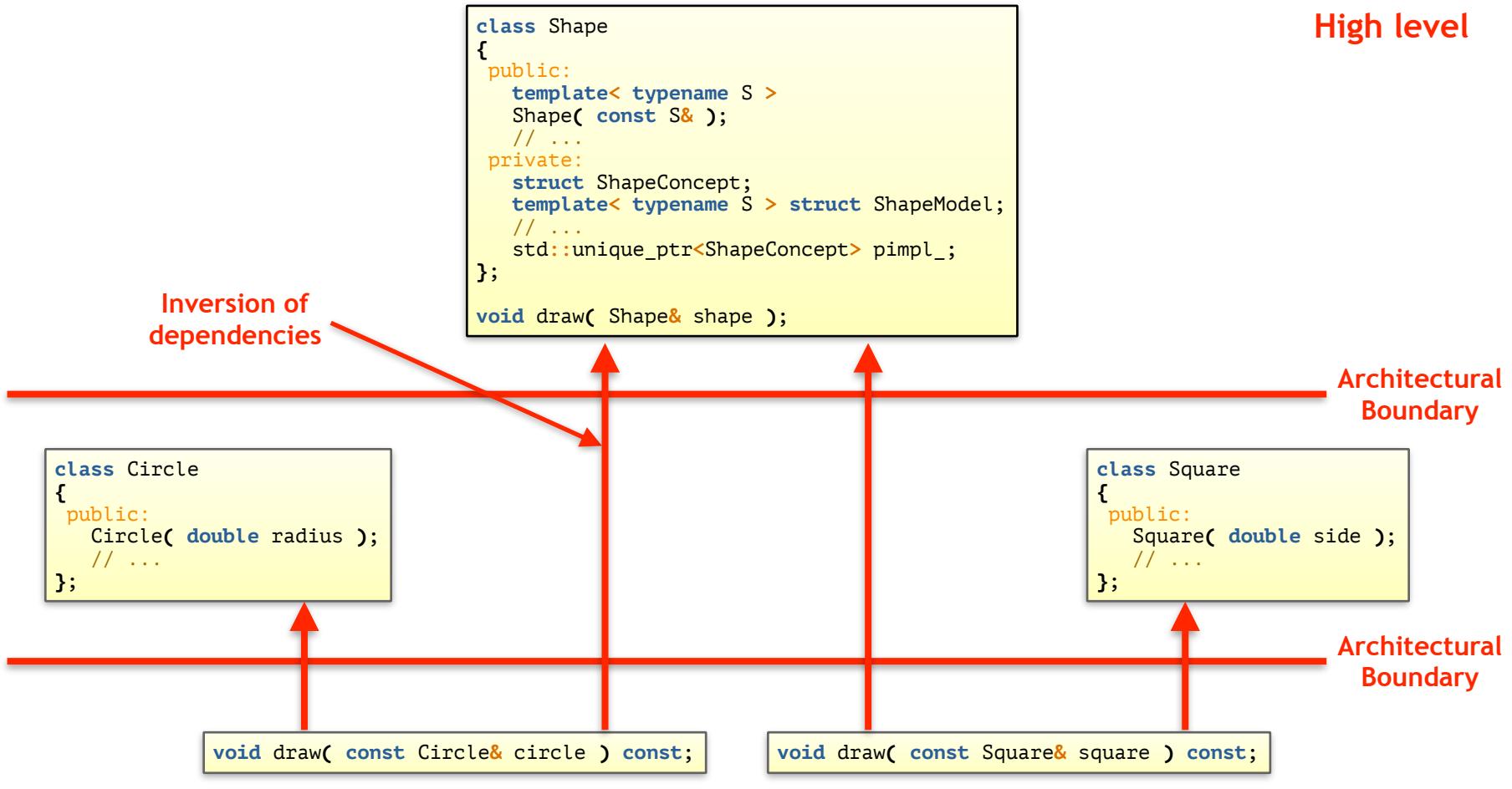
Architectural  
Boundary

Low level

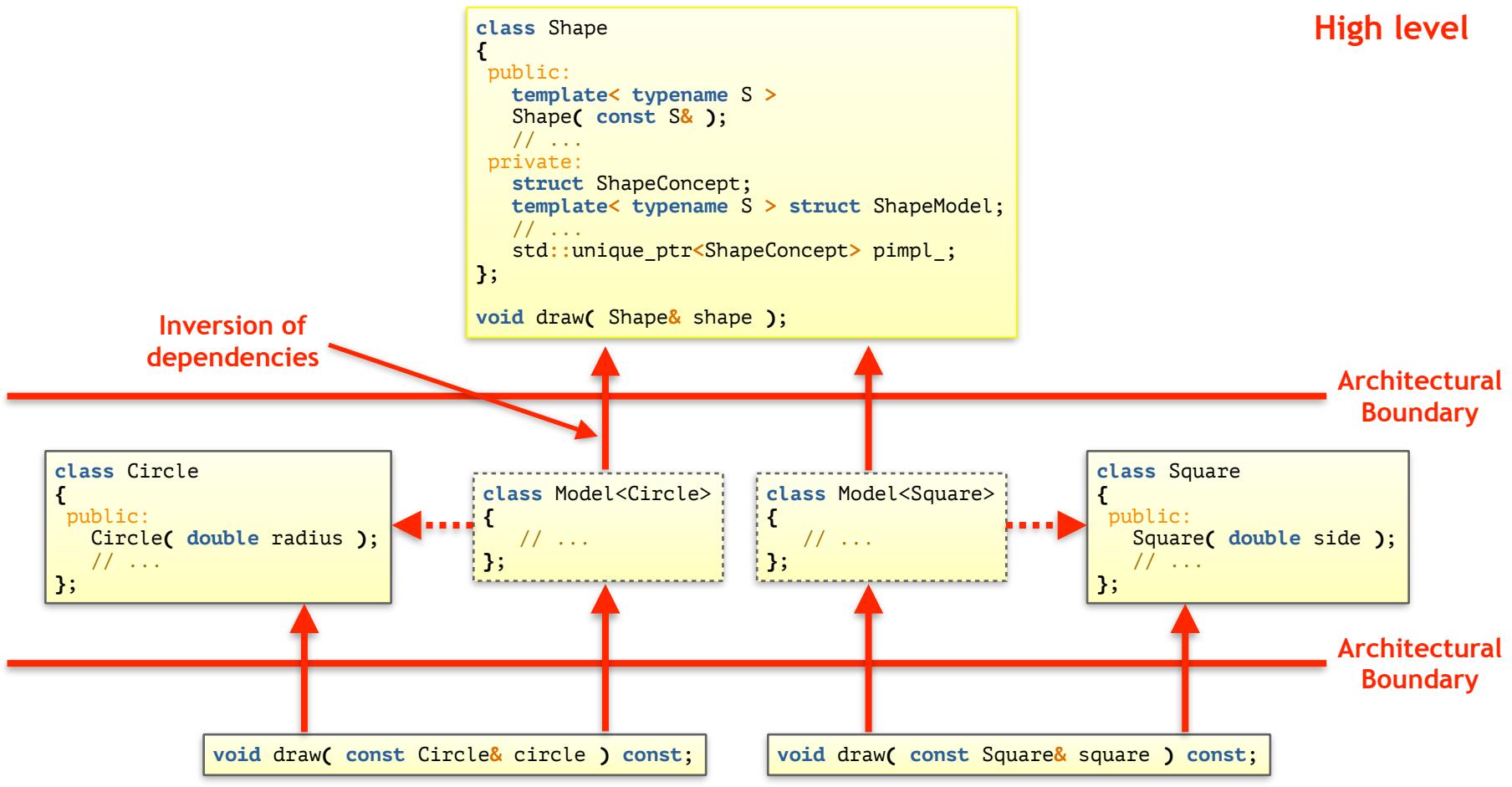


High level

# An Architectural Puzzle (Solution)



# An Architectural Puzzle (Solution)



Low level

# Type Erasure with Small Buffer Optimization

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

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

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

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

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

    double getSide() const noexcept;
```

# Type Erasure with Small Buffer Optimization

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

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

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

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

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

    double getSide() const noexcept;
```

# Type Erasure with Small Buffer Optimization

```
void draw( Circle const& );  
  
class Square  
{  
public:  
    explicit Square( double s )  
        : side{ s }  
        , // ... Remaining data members  
    {}  
  
    double getSide() const noexcept;  
    // ... getCenter(), getRotation(), ...  
  
private:  
    double side;  
    // ... Remaining data members  
};  
  
void translate( Square const&, Vector2D const& );  
void rotate( Square const&, double const& );  
void draw( Square const& );  
  
class Shape  
{  
private:  
    struct Concept  
    {  
        virtual ~Concept() = default;  
        virtual void doTranslate( Vector2D const& v ) const = 0;
```

# Type Erasure with Small Buffer Optimization

```
class Shape
{
private:
    struct Concept
    {
        virtual ~Concept() = default;
        virtual void do_translate( Vector2D const& v ) const = 0;
        virtual void do_rotate( double const& q ) const = 0;
        virtual void do_draw() const = 0;
        // ...
    };
};

template< typename ShapeT >
struct Model final : public Concept
{
    Model( ShapeT const& value )
        : object{ value }
    {}

    void do_translate( Vector2D const& v ) const override
    {
        translate( object, v );
    }

    void do_rotate( double const& q ) const override
    {
        rotate( object, q );
    }

    void do_draw() const override
    {
        draw( object );
    }
};
```

# Type Erasure with Small Buffer Optimization

```
class Shape
{
private:
    struct Concept
    {
        virtual ~Concept() = default;
        virtual void do_translate( Vector2D const& v ) const = 0;
        virtual void do_rotate( double const& q ) const = 0;
        virtual void do_draw() const = 0;
        // ...
    };
};

template< typename ShapeT >
struct Model final : public Concept
{
    Model( ShapeT const& value )
        : object{ value }
    {}

    void do_translate( Vector2D const& v ) const override
    {
        translate( object, v );
    }

    void do_rotate( double const& q ) const override
    {
        rotate( object, q );
    }

    void do_draw() const override
    {
        draw( object );
    }
};
```

# Type Erasure with Small Buffer Optimization

```
};

template< typename ShapeT >
struct Model final : public Concept
{
    Model( ShapeT const& value )
        : object{ value }
    {}

    void do_translate( Vector2D const& v ) const override
    {
        translate( object, v );
    }

    void do_rotate( double const& q ) const override
    {
        rotate( object, q );
    }

    void do_draw() const override
    {
        draw( object );
    }

    // ...

    ShapeT object;
};

static constexpr size_t buffersize = 128UL;
static constexpr size_t alignment = 16UL;
```

# Type Erasure with Small Buffer Optimization

```
void do_draw() const override
{
    draw( object );
}

// ...

ShapeT object;
};

static constexpr size_t buffersize = 128UL;
static constexpr size_t alignment = 16UL;

alignas(alignment) std::array<std::byte, buffersize> buffer;

Concept* pimpl() noexcept
{
    return reinterpret_cast<Concept*>( buffer.data() );
}
const Concept* pimpl() const noexcept
{
    return reinterpret_cast<const Concept*>( buffer.data() );
}

friend void translate( Shape& shape, Vector2D const& v )
{
    shape.pimpl()->do_translate( v );
}

friend void rotate( Shape& shape, double const& q )
{
```

# Type Erasure with Small Buffer Optimization

```
{  
    return reinterpret_cast<Concept*>( buffer.data() );  
}  
const Concept* pimpl() const noexcept  
{  
    return reinterpret_cast<const Concept*>( buffer.data() );  
}  
  
friend void translate( Shape& shape, Vector2D const& v )  
{  
    shape.pimpl()->do_translate( v );  
}  
  
friend void rotate( Shape& shape, double const& q )  
{  
    shape.pimpl()->do_rotate( q );  
}  
  
friend void draw( Shape const& shape )  
{  
    shape.pimpl()->do_draw();  
}  
  
public:  
    template< typename ShapeT >  
    Shape( ShapeT const& shape )  
    {  
        using M = Model<ShapeT>;  
        static_assert( sizeof(M) <= buffersize, "Given type is too large" );  
        static_assert( alignof(M) <= alignment, "Given type is overaligned" );  
        ::new (pimpl()) M( shape );  
    }
```

# Type Erasure with Small Buffer Optimization

```
friend void draw( Shape const& shape )
{
    shape.pimpl()->do_draw();
}

public:
    template< typename ShapeT >
    Shape( ShapeT const& shape )
    {
        using M = Model<ShapeT>;
        static_assert( sizeof(M) <= buffersize, "Given type is too large" );
        static_assert( alignof(M) <= alignment, "Given type is overaligned" );
        ::new (pimpl()) M( shape );
    }

    // Special member functions
    ~Shape();
    Shape( Shape const& other );
    Shape& operator=( Shape const& other );

    // Move operations intentionally ignored!

    // ...
};

void drawAllShapes( std::vector<Shape> const& shapes )
{
    for( auto const& shape : shapes )
    {
```

# Type Erasure with Small Buffer Optimization

```
// Operator members declarations
~Shape();
Shape( Shape const& other );
Shape& operator=( Shape const& other );

// Move operations intentionally ignored!

// ...
};

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

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
}
```

# Type Erasure with Small Buffer Optimization

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

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 );
}
```

# Parameter Conventions: Concrete Types

Passing by reference-to-const to a concrete type we can express that we can only work with a specific kind of shape.

```
Square square{ 1.5 };
Shape  shape{ Square{ 1.5 } };
```

```
void f( const Square& );      // Reference to a concrete type:
                                // Works only for Square, not for
                                // other shapes
```

```
f( square );    // Compiles
f( shape );     // Does NOT compile
```

# Parameter Conventions: Abstract Types (I)

Passing by reference-to-const to the type erased type we can express that we can only work with any kind of shape.

```
Square square{ 1.5 };
Shape  shape{ Square{ 1.5 } };
```

```
void f( const Shape& );      // Reference to abstract type:
                                // Works for all kinds of
                                // shapes, but might allocate
```

```
f( square );    // Compiles
f( shape );     // Compiles, but creates a temporary 'Shape'
```

## Parameter Conventions: Abstract Types (II)

Passing by value to a non-owning kind of type erasure we can express that we can only work with any kind of shape, but don't need to own.

```
Square square{ 1.5 };
Shape  shape{ Square{ 1.5 } };
```

```
void f( ShapeConstRef );      // Pass via non-owning type erasure:
                                // Works for all kinds of
                                // shapes, no allocation

f( square );    // Compiles
f( shape );     // Compiles, no allocation
```

# Non-Intrusive API Design: Type Erasure

---

**Task (C\_Modern\_Cpp\_Design\_Patterns/TypeErasure\_Ref):** Implement the `ShapeConstRef` class, representing a reference to a constant shape, by means of Type Erasure. `ShapeConstRef` may require all types to provide a free `draw()` function that draws them to the screen.

# Non-Intrusive API Design: Type Erasure

---

```
class ShapeConstRef
{
public:
    template< typename ShapeT >
    ShapeConstRef( ShapeT const& shape )
        : shape_{ std::addressof(shape) }
        , draw_{ []( void const* shape ) {
                draw( *static_cast<ShapeT const*>(shape) );
            } }
    {}

private:
    friend void draw( ShapeConstRef const& shape )
    {
        shape.draw_( shape.shape_ );
    }

    using DrawOperation = void(void const*);

    void const* shape_{ nullptr };
    DrawOperation* draw_{ nullptr };
};

};
```

# Non-Intrusive API Design: Type Erasure

---

**Task (C\_Modern\_Cpp\_Design\_Patterns/Function\_Ref):** Implement a simplified std::function\_ref to represent a non-owning abstraction for any type of callable.

# Non-Intrusive API Design: Type Erasure

```
template< typename Fn >
class Function_Ref;

template< typename R, typename... Args >
class Function_Ref<R(Args...)>
{
public:
    template< typename Fn >
    Function_Ref( Fn const& fn )
        : invoke_( []( void const* c, Args... args ) -> R {
            auto const* const fn( static_cast<Fn const*>(c) );
            return (*fn)( std::forward<Args>(args)... );
        } )
        , callable_( std::addressof(fn) )
    {}

    R operator()( Args... args ) const
    {
        return invoke_( callable_, std::forward<Args>(args)... );
    }

private:
    using InvokeOperation = R(void const*, Args...);
```

# Non-Intrusive API Design: Type Erasure

```
template< typename R, typename... Args >
class Function_Ref<R(Args...)>
{
public:
    template< typename Fn >
    Function_Ref( Fn const& fn )
        : invoke_( [](<void const*> c, Args... args) -> R {
            auto const* const fn( static_cast<Fn const*>(c) );
            return (*fn)( std::forward<Args>(args)... );
        } )
        , callable_( std::addressof(fn) )
    {}

    R operator()( Args... args ) const
    {
        return invoke_( callable_, std::forward<Args>(args)... );
    }

private:
    using InvokeOperation = R(void const*, Args...);

    InvokeOperation* invoke_{ nullptr };
    void const* callable_;
};

};
```

# Type Erasure

---

In its basic form, type erasure is a clever combination of several classic design patterns:

- Prototype: Enables virtual copying;
- Bridge: Provides a compilation firewall;
- External Polymorphism: Adds a new inheritance hierarchy to enable the storage of any requirement fulfilling type;
- Proxy: Enables a seamless integration into existing code.

Also, it nicely combines the strengths of static and dynamic polymorphism by means of ...

- ... an encapsulated inheritance hierarchy;
- ... a class template.

# Type Erasure

---

Classic design patterns ...

- ... require a base class (dependency);
- ... promote heap allocation;
- ... require memory management.

Using type erasure instead ...

- ... simplifies code by encapsulating all responsibilities;
- ... facilitates comprehension;
- ... reduces dependencies;
- ... allows performance optimizations (e.g. SBO).

# Type Erasure – Advantages/Disadvantages

---

Use type erasure if ...

- ... you have a **closed set** of functions;
- ... you want to **extend types**, not functions;
- ... you want to promote **polymorphic usage of types**.

Don't use type erasure if ...

- ... you have an **open set** of functions;
- ... you want to **extend functions**, not types.

# Applied Type Erasure

---

**Task (C\_Modern\_Cpp\_Design\_Patterns/PolymorphicAllocator):**  
Implement the PolymorphicAllocator class by means of Type Erasure. PolymorphicAllocator may require all types to provide a free allocate() and a deallocate() member function.

# Guidelines

---

**Guideline:** Use Type Erasure to enable the non-intrusive and value-based addition of polymorphic types.

**Core Guideline T.49:** Where possible, avoid type-erasure

# Things to Remember

---

- Minimize couplings wherever possible
- Consider modern programming techniques to break inheritance relationships
- Prefer containment (composition/aggregation) to inheritance
- Prefer value semantics based solutions
- Keep it simple (KISS)

# Literature



# References

---

- Sean Parent, “Inheritance is the Base Class of Evil”. GoingNative 2013 (<http://channel9.msdn.com/Events/GoingNative/2013/Inheritance-Is-The-Base-Class-of-Evil>)
- John Lakos, “Value Semantics: It ain’t about the syntax! (Part I)”. CppCon 2015 (<https://www.youtube.com/watch?v=W3xI1HJUy7Q>)
- Klaus Iglberger, “Back to Basics: Value Semantics”. CppCon 2022 (<https://www.youtube.com/watch?v=G9MxNwUoSt0>)
- Dave Abrahams, “Values: Regularity, Independence, Projection, and the Future of Programming”. CppCon 2022 (TBA)
- Klaus Iglberger, “Design Patterns - Facts and Misconceptions”. CppCon 2021 (<https://www.youtube.com/watch?v=OvO2NR7pXjg>)
- Mateusz Pusz, “Effective replacement of dynamic polymorphism with std::variant”. CppCon 2018 (<https://www.youtube.com/watch?v=gKbORJtnVu8>)
- Michael Caisse, “Modern C++ in Embedded Systems”. C++Now 2018 (<https://www.youtube.com/watch?v=c9Xt6Me3mJ4>)
- Jon Kalb, “Back to Basics: Object-Oriented Programming”. CppCon 2019 (<https://www.youtube.com/watch?v=32tDTD9UJCE>)

# References

---

- Arthur O'Dwyer, “Back to Basics: Type Erasure”. CppCon 2019 (<https://www.youtube.com/watch?v=tbUCHifyT24>)
- Klaus Iglberger, “Breaking Dependencies: Type Erasure - A Design Analysis”. Meeting C++ 2021 (<https://www.youtube.com/watch?v=jKt6A3wnDyl>)
- Zach Laine, “Pragmatic Type Erasure: Solving OOP Problems with an Elegant Design Pattern”. CppCon 2014 (<https://www.youtube.com/watch?v=0I0FD3N5cgM>)
- Sy Brand, “Dynamic Polymorphism with Metaclasses and Code Injection”, CppCon 2020 ([https://www.youtube.com/watch?v=8c6BAQcYF\\_E](https://www.youtube.com/watch?v=8c6BAQcYF_E))
- Eduardo Madrid, “Not Leaving Performance On The Jump Table”, CppCon 2020 ([https://www.youtube.com/watch?v=e8SyxB3\\_mnw](https://www.youtube.com/watch?v=e8SyxB3_mnw))

# Online Resources

---

- Working Draft, Standard for Programming Language C++: <http://eel.is/c++draft/>
- C++ Reference: [www.cppreference.com](http://www.cppreference.com)
- C++ Core Guidelines: [isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines](https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines)
- Stackoverflow: [www.stackoverflow.com](http://www.stackoverflow.com)
- Compiler Explorer: [www.godbolt.org](http://www.godbolt.org)
- Quick-Bench: [www.quick-bench.com](http://www.quick-bench.com)
- C++ Insights: [www.cppinsights.io](http://www.cppinsights.io)
- Build-Bench: [www.build-bench.com](http://www.build-bench.com)
- C++ Shell: [cpp.sh](http://cpp.sh)
- Wandbox: [wandbox.org](http://wandbox.org)
- repl.it: [repl.it](http://repl.it)
- Intel Intrinsics Guide: [software.intel.com/sites/landingpage/IntrinsicsGuide](http://software.intel.com/sites/landingpage/IntrinsicsGuide)
- x86/x64 SIMD Instruction List: <https://www.officedaytime.com/simd512e/>

# Additional Online Resources

---

- C++ Bestiary: <http://videocortex.io/2017/Bestiary/>
- More C++ Idioms: [https://en.wikibooks.org/wiki/More\\_C%2B%2B\\_Idioms](https://en.wikibooks.org/wiki/More_C%2B%2B_Idioms)
- Codewars: <https://www.codewars.com>
- CodeKata: <http://codekata.com>

email: [klaus.iglberger@gmx.de](mailto:klaus.iglberger@gmx.de)

LinkedIn: [linkedin.com/in/klaus-iglberger-2133694/](https://www.linkedin.com/in/klaus-iglberger-2133694/)

Xing: [xing.com/profile/Klaus\\_Iglberger/cv](https://www.xing.com/profile/Klaus_Iglberger/cv)