

C++ Software Design @ O'Reilly

2. C++ Software Design

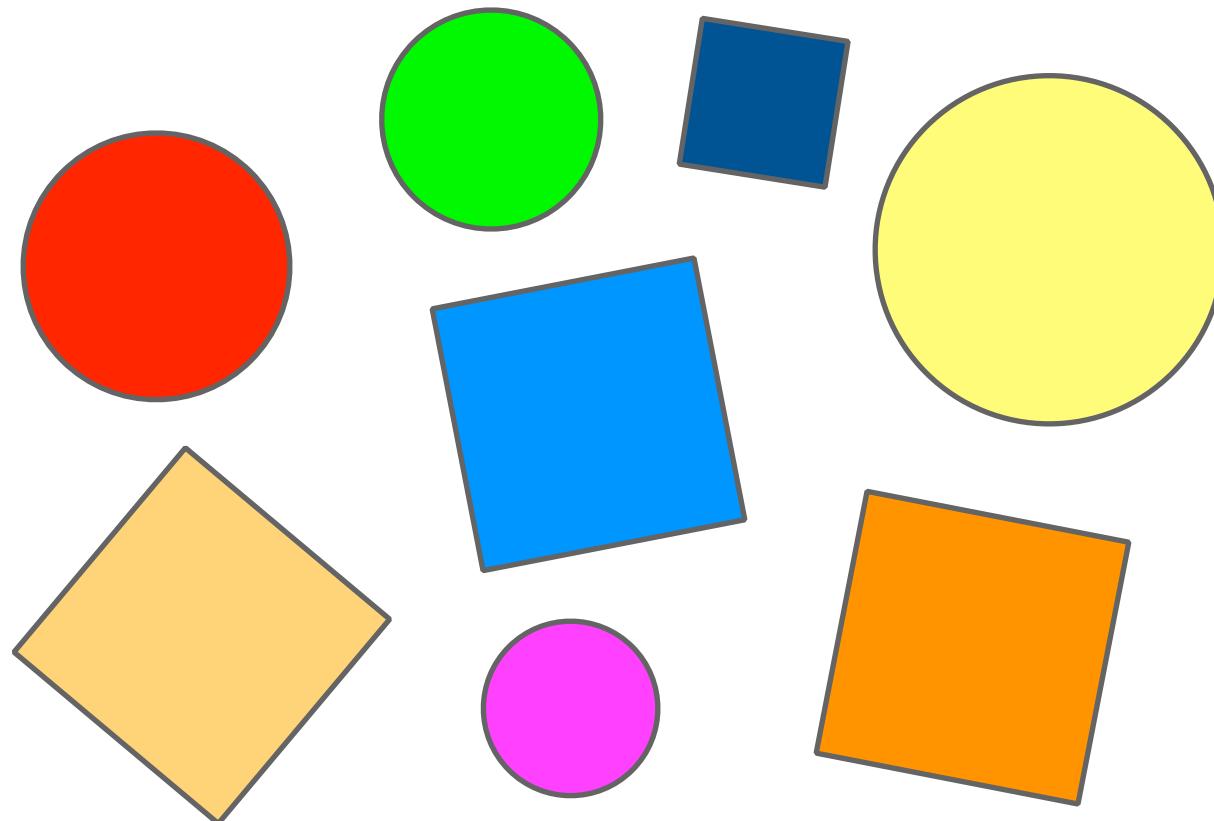
Klaus Iglberger
December, 5th-6th, 2024

Content

1. Motivation
2. Software Architecture vs. Design
3. Visitor
4. Strategy
5. Policy-Based Design
6. Interlude
7. External Polymorphism
8. Type Erasure
9. Prototype
10. Bridge

2.1. Motivation

Our Toy Problem: Drawing Shapes



An Example

Task (2_Cpp_Software_Design/Motivation/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() )  
        {  
            case Circle:  
                draw( *s );  
                break;  
            case Square:  
                draw( *s );  
                break;  
            case Triangle:  
                draw( *s );  
                break;  
            default:  
                break;  
        }  
    }  
}
```

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 Expert's Opinion



"Since change is the dominant cost of software development and understanding code is the dominant cost of change, communicating the structure and intent of working code is one of the most valuable skills you can exercise."

(Kent Beck, Tidy First?)

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)

The Expert's Opinion



"What “propagates” change? Coupling. So the cost of software is approximately equal to the coupling [...]:

$\text{cost}(\text{software}) \sim= \text{cost}(\text{changes}) \sim= \text{cost}(\text{big changes}) \sim= \text{coupling.}$

Or, to highlight the importance of software design:

$\text{cost}(\text{software}) \sim= \text{coupling.}$ "

(Kent Beck, Tidy First?)

Guidelines

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

Overview

Single-Responsibility Principle (SRP)

Open-Closed Principle (OCP)

Liskov Substitution Principle (LSP)

Interface Segregation Principle (ISP)

Dependency Inversion Principle (DIP)

Overview

Single-Responsibility Principle (SRP)

Open-Closed Principle (OCP)

Liskov Substitution Principle (LSP)

Development Inversion Principle (DIP)

Interface Segregation Principle (ISP)



Robert C. Martin

Overview

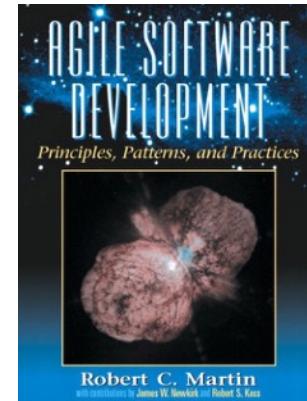
Single-Responsibility Principle (SRP)

Open-Closed Principle (OCP)

Liskov Substitution Principle (LSP)

Dependency Inversion Principle (DIP)

Interface Segregation Principle (ISP)



Robert C. Martin

2002

Overview

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_Cpp_Software_Design/Motivation/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;
```

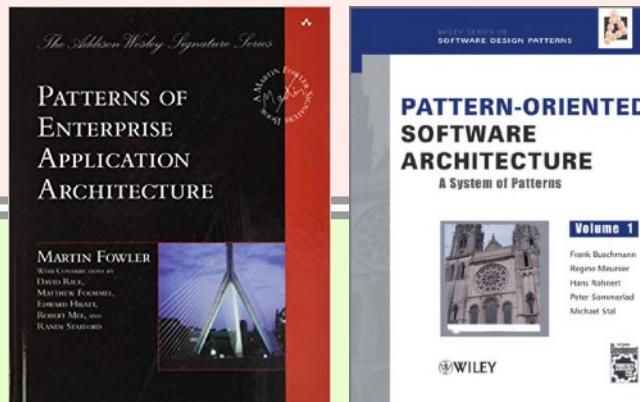
2.2. Software Architecture vs. Design

What is Software Design/Architecture?

2. C++ Software Design - Software Architecture vs. Design

Software Architecture

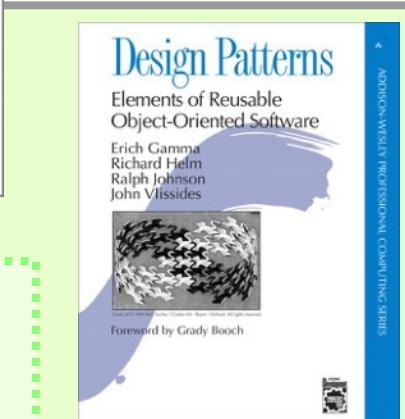
- ⦿ How are big entities depending on each other?
- ⦿ Design decisions that are harder to change
- ⦿ Architectural patterns
- ⦿ Examples:
 - ⦿ Client-Server Architecture
 - ⦿ Microservices
 - ⦿ MVC, ...



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
 - ⦿ ...

- Idioms**
- ⦿ NVI Idiom (Template Method Design Pattern)
 - ⦿ Pimpl Idiom (Bridge Design Pattern)



Implementation Details

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

- ⦿ Temporary-Swap Idiom
- ⦿ RAII Idiom
- ⦿ enable_if
- ⦿ Factory function

What is Software Design/Architecture?

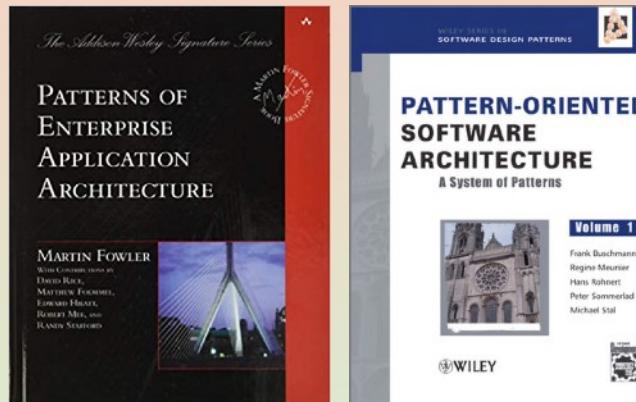


“..., I’ll assert that there is no difference between [architecture and design]. None at all.”
(Robert C. Martin, Clean Architecture)

2. C++ Software Design - Software Architecture vs. Design

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
 - ⦿ ...

Idioms

- ⦿ NVI Idiom (Template Method Design Pattern)
- ⦿ Pimpl Idiom (Bridge Design Pattern)

- ⦿ Temporary-Swap Idiom
- ⦿ RAII Idiom
- ⦿ `enable_if`
- ⦿ Factory function

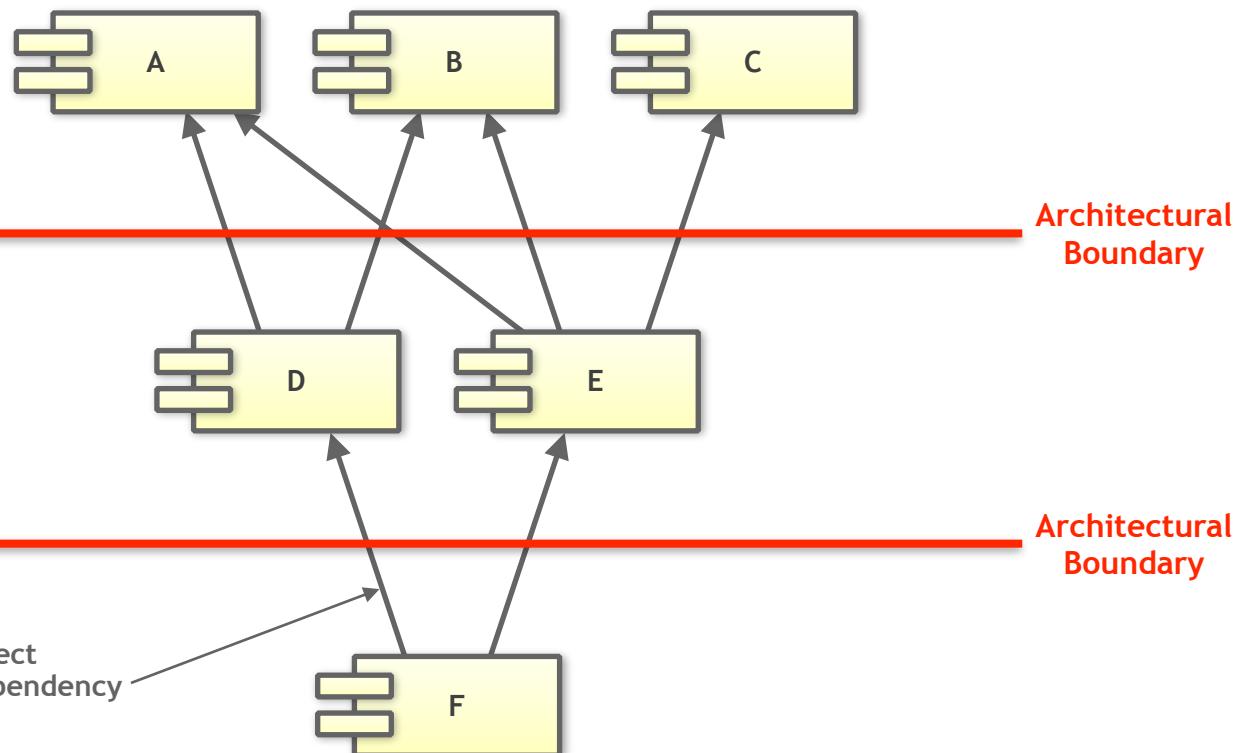
Implementation Details

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

What is Software Design/Architecture?

High level

(stable, low dependencies)



This is an architecture: all arrows point upwards to the next higher level of abstraction. Everything builds on more stable components.

Low level

(volatile, malleable, high dependencies)

What is Software Design/Architecture?

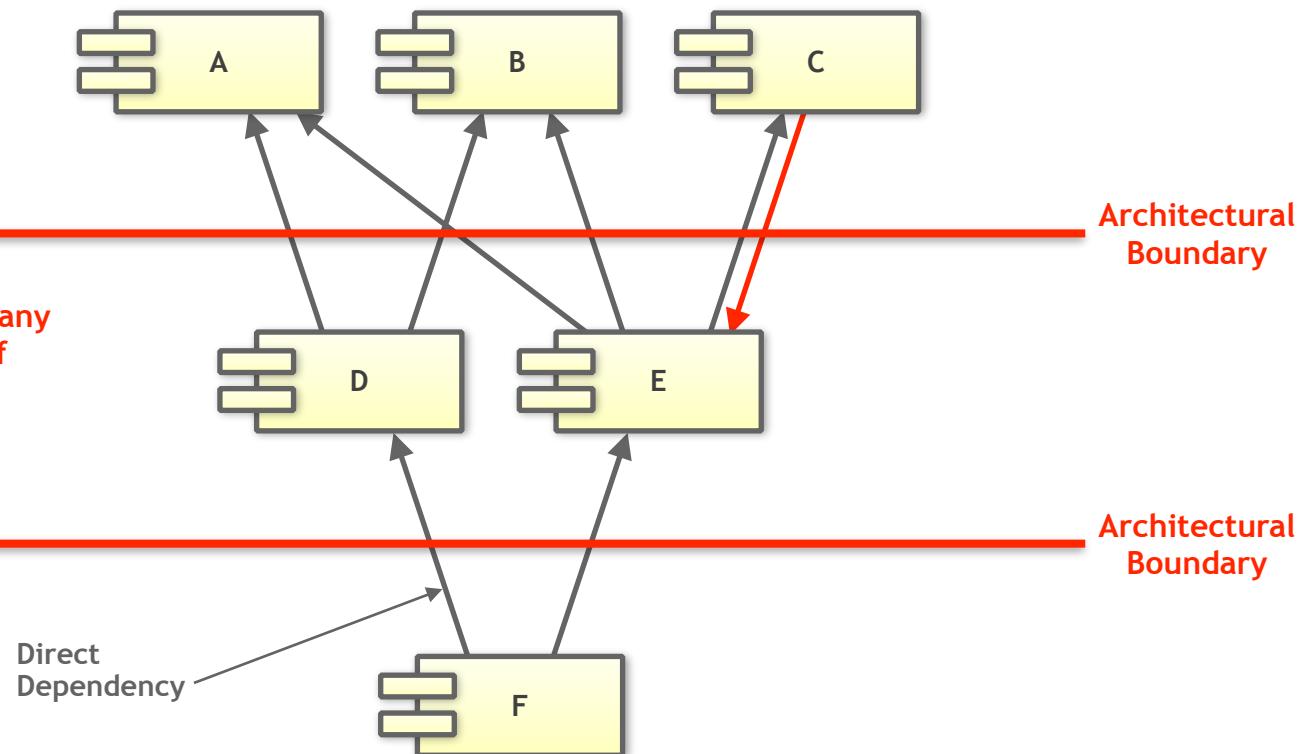
High level

(stable, low dependencies)

This is NOT an architecture: if any arrow points downwards (i.e. if anything “stable” depends on something less stable) the architecture is broken.

Low level

(volatile, malleable, high dependencies)



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.

The First Law of Software Architecture



"Everything in software architecture is a trade-off."

(Neal Ford, Mark Richards, Head First Software Architecture)



What is a Design Pattern?

What is a Design Pattern?



"A Design Pattern is a named canonical form for a combination of software structures and procedures that have proven to be useful over the years."

(Robert C. Martin, A Little About Patterns, The Clean Code Blog, June, 30th, 2014)

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**.

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.”

Terminology



"The use of [design patterns] provides us in our daily lives with decisive speed advantages for understanding complex structures. This is also why patterns found their way into software development years ago. ... Consistently applied patterns help us deal with the complexity of source code."

(Carola Lilienthal, Software Architecture Metrics)

The Attitude Toward Design Patterns



1 month ago

Really? Design Patterns in 2021?

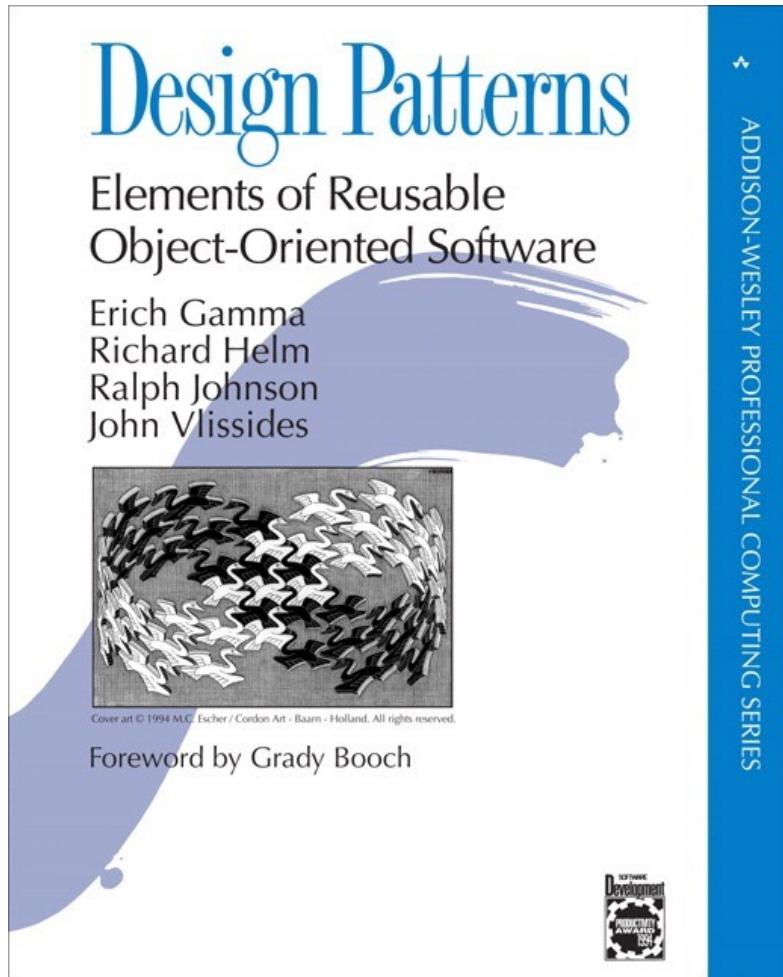


REPLY

Guidelines

Guideline: Design Patterns are Everywhere!

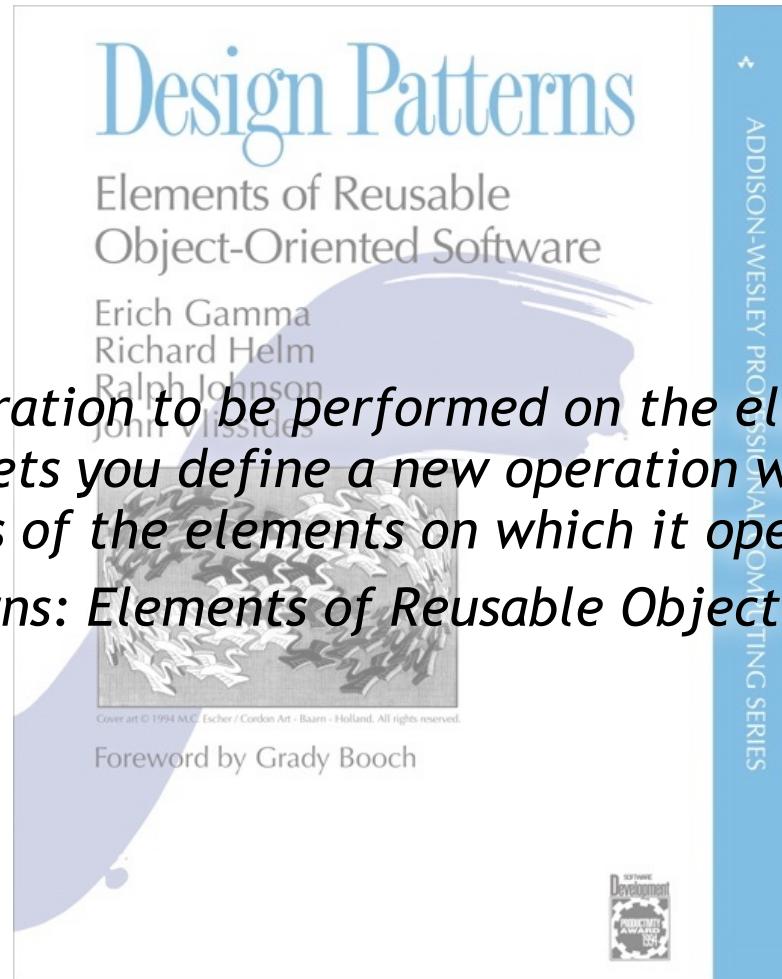
Terminology



- The Gang of Four (GoF) book
- Published in 1994
- Source of 23 of the most commonly used design patterns
- Almost all design patterns are based on inheritance

2.3. Visitor

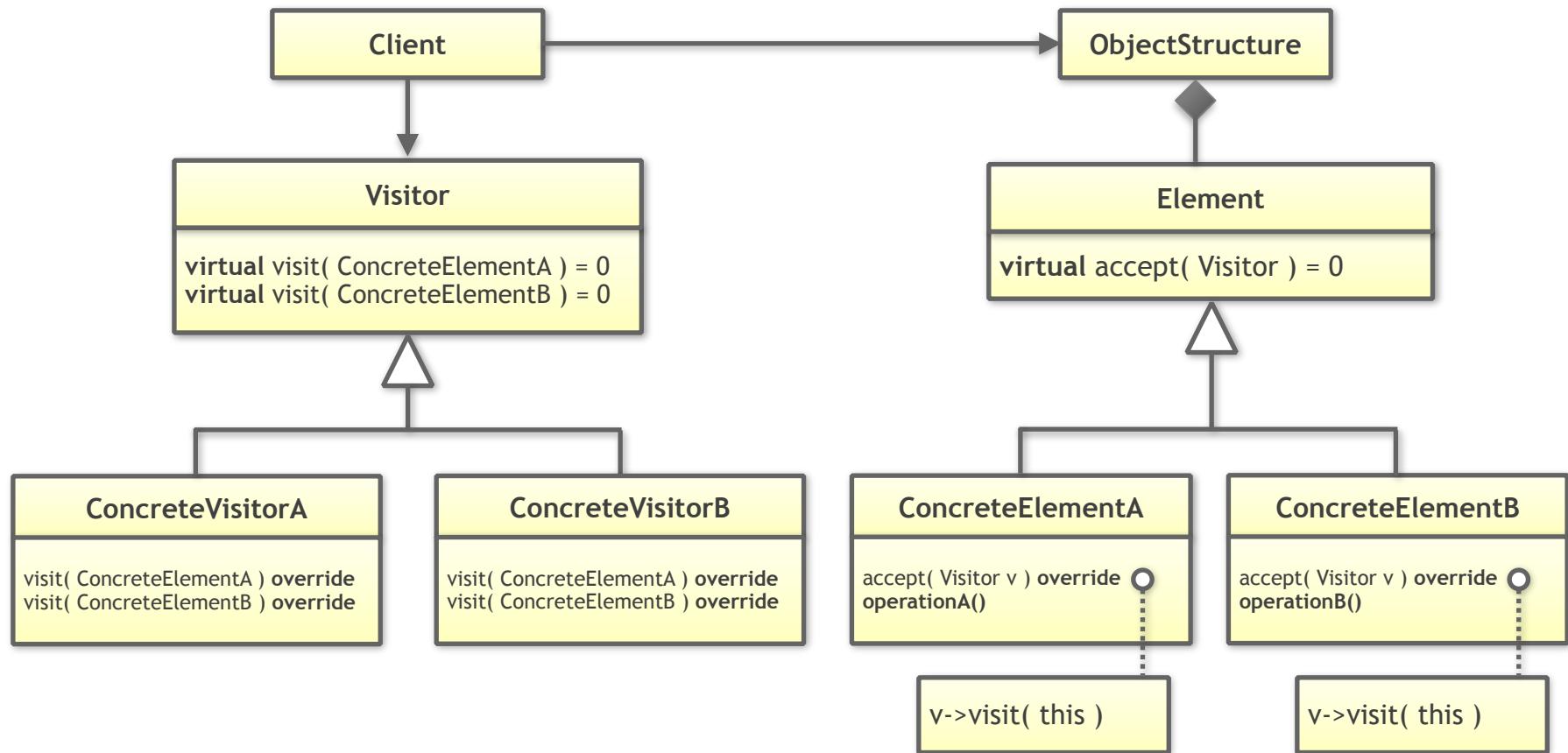
The Intent



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

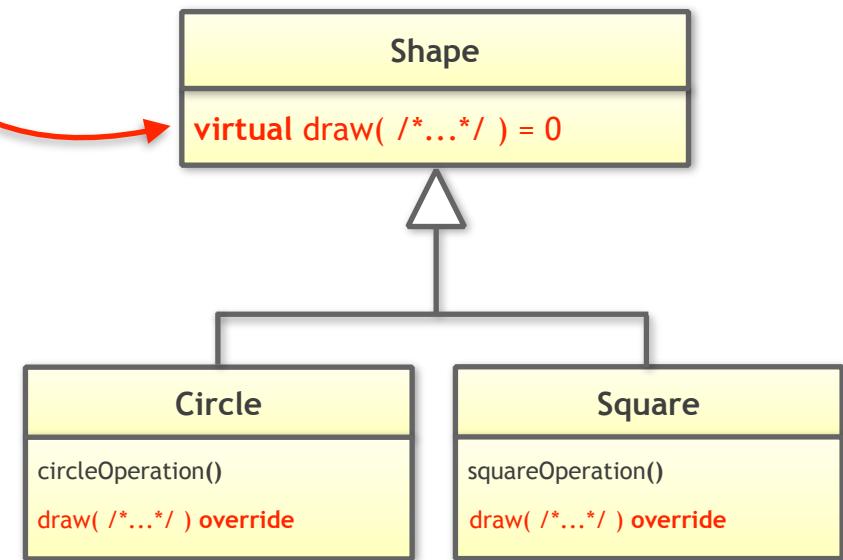
(GoF, Design Patterns: Elements of Reusable Object-Oriented Software)

The Classic Visitor Design Pattern

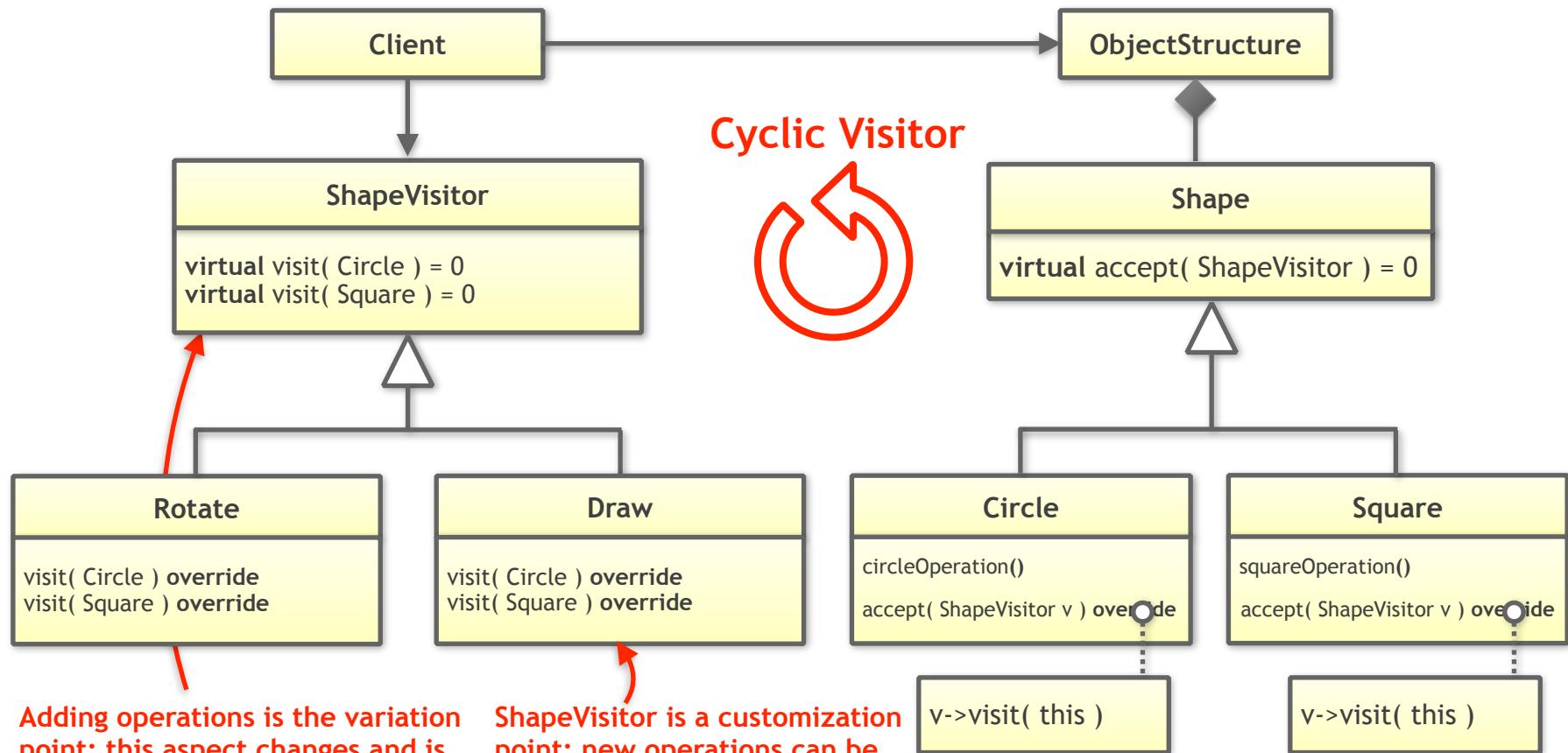


The Classic Visitor Design Pattern

The weakness of Object-Oriented Programming: adding an operation requires an update of all derived classes. This violates the Open-Closed Principle (OCP).



The Classic Visitor Design Pattern



The Classic Visitor Design Pattern

Task (2_Cpp_Software_Design/Visitor/Visitor_Classic):

Step 1: Refactor the given Shape hierarchy by means of the visitor design pattern to enable the easy addition of operations (e.g. drawing, rotating, serialization, ...).

Step 2: Implement the area() operations as a classic visitor. Hint: the area of a circle is $\text{radius} * \text{radius} * \text{M_PI}$, the area of a square is $\text{side} * \text{side}$.

Step 3: Switch from one to another graphics library. Discuss the feasibility of the change: how easy is the change? How many pieces of code on which level of the architecture have to be touched?

The Classic Visitor Design Pattern

Task (2_Cpp_Software_Design/Visitor/Visitor_Refactoring):

Step 1: Implement the area() operations as a classic visitor. Hint: the area of a circle is `radius*radius*M_PI`, the area of a square is `side*side`.

Step 2: Refactor the classic Visitor solution by a value semantics based solution. Note that the general behavior should remain unchanged.

Step 3: Switch from one to another graphics library. Discuss the feasibility of the change: how easy is the change? How many pieces of code on which level of the architecture have to be touched?

Step 4: Discuss the advantages of the value semantics based solution in comparison to the classic solution.

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: std::variant

```
using Shape = std::variant<Circle,Square>;  
  
Shape s = ...;  
  
if( std::holds_alternative<Circle>(s) ) {  
    double radius = std::get<Circle>(s).radius();  
    double area = radius * radius * std::pi_v;  
}  
else if( std::holds_alternative<Square>(s) ) {  
    double side = std::get<Square>(s).side();  
    double area = side * side;  
}  
// ...
```

A “Modern C++” Solution: std::variant



“Nothing prevents you to handle a certain case. If you forget to handle one of the cases, it is a bug. And this becomes particularly problematic if, God forbids, somebody ever comes around and adds a new kind of shape to the variant.”

(Andreas Weis, Building C++ Interfaces That Are Hard to Use Incorrectly, ACCU 2023)

A “Modern C++” Solution: std::variant

```
double getArea( Circle const& circle );
double getArea( Square const& square );

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

Shape s = ...;

double area =
    std::visit(
        []( auto const& shape ){ return getArea(shape); }, s );
```

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);
}
```

Returning from a Visitor – The Classic Way

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

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

class Area : public ShapeVisitor
{
public:
    void visit( Circle const& circle ) const override
    {
        result_ = circle.radius()*circle.radius()*M_PI;
    }

    void visit( Square const& square ) const override
    {
        result_ = square.side()*square.side();
    }

    double getResult() const { return result_; }

private:
    double mutable result_;
};
```

Returning from a Visitor – The Classic Way

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

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

class Area : public ShapeVisitor
{
public:
    void visit( Circle const& circle ) const override
    {
        result_ = circle.radius()*circle.radius()*M_PI;
    }

    void visit( Square const& square ) const override
    {
        result_ = square.side()*square.side();
    }

    double getResult() const { return result_; }

private:
    double mutable result_;
};
```

Returning from a Visitor – The Classic Way

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

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

class Area : public ShapeVisitor
{
public:
    void visit( Circle const& circle ) const override
    {
        result_ = circle.radius()*circle.radius()*M_PI;
    }

    void visit( Square const& square ) const override
    {
        result_ = square.side()*square.side();
    }

    double getResult() const { return result_; }

private:
    double mutable result_;
};
```

Returning from a Visitor – The Classic Way

```
class Area : public ShapeVisitor
{
public:
    void visit( Circle const& circle ) const override
    {
        result_ = circle.radius()*circle.radius()*M_PI;
    }

    void visit( Square const& square ) const override
    {
        result_ = square.side()*square.side();
    }

    double getResult() const { return result_; }

private:
    double mutable result_;
};

int main()
{
    std::unique_ptr<Shape> shape = ...;
    Area area{};

    shape.accept( area );
    double const result = area.getResult();
}
```

Returning from a Visitor – std::variant

```
class Area
{
public:
    double operator()( Circle const& circle ) const
    {
        return circle.radius() * circle.radius() * M_PI;
    }
    double operator()( Square const& square ) const
    {
        return square.side() * square.side();
    }
};

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

int main()
{
    Shape shape = ...;

    double const area = std::visit( Area{}, shape );
}
```

Note that every visitor can return some different. This offers much more flexibility than an inheritance hierarchy.

Inheritance Creates a Tight Coupling



"If a class relationship can be expressed in more than one way, use the weakest relationship that's practical. Given that inheritance is nearly the strongest relationship you can express in C++ (second only to friendship), it's only really appropriate when there is no equivalent weaker alternative."

(Herb Sutter, Exceptional C++)

A “Modern C++” Solution: std::variant

```
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: std::variant

```
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: std::variant

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

A “Modern C++” Solution: std::variant

```
private:  
    double side;  
    // ... Remaining data members  
};  
  
using Shape = std::variant<Circle,Square>;  
  
class Draw  
{  
public:  
    void operator()( Circle const& ) const;  
    void operator()( Square const& ) const;  
    // ...  
};  
  
void drawAllShapes( std::vector<Shape> const& shapes )  
{  
    for( auto const& s : shapes )  
    {  
        std::visit( Draw{}, s );  
    }  
}  
  
int main()  
{  
    using Shapes = std::vector<Shape>;
```

A “Modern C++” Solution: std::variant

```
private:  
    double side;  
    // ... Remaining data members  
};  
  
using Shape = std::variant<Circle,Square>;  
  
class Draw  
{  
public:  
    void operator()( Circle const& ) const;  
    void operator()( Square const& ) const;  
    // ...  
};  
  
void drawAllShapes( std::vector<Shape> const& shapes )  
{  
    for( auto const& s : shapes )  
    {  
        std::visit( Draw{}, s );  
    }  
}  
  
int main()  
{  
    using Shapes = std::vector<Shape>;
```

A “Modern C++” Solution: std::variant

```
private:  
    double side;  
    // ... Remaining data members  
};  
  
using Shape = std::variant<Circle,Square>;  
  
class Draw  
{  
public:  
    void operator()( Circle const& ) const;  
    void operator()( Square const& ) const;  
    // ...  
};  
  
void drawAllShapes( std::vector<Shape> const& shapes )  
{  
    for( auto const& s : shapes )  
    {  
        std::visit( Draw{}, s );  
    }  
}  
  
int main()  
{  
    using Shapes = std::vector<Shape>;
```

A “Modern C++” Solution: std::variant

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

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

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

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

Performance Comparison

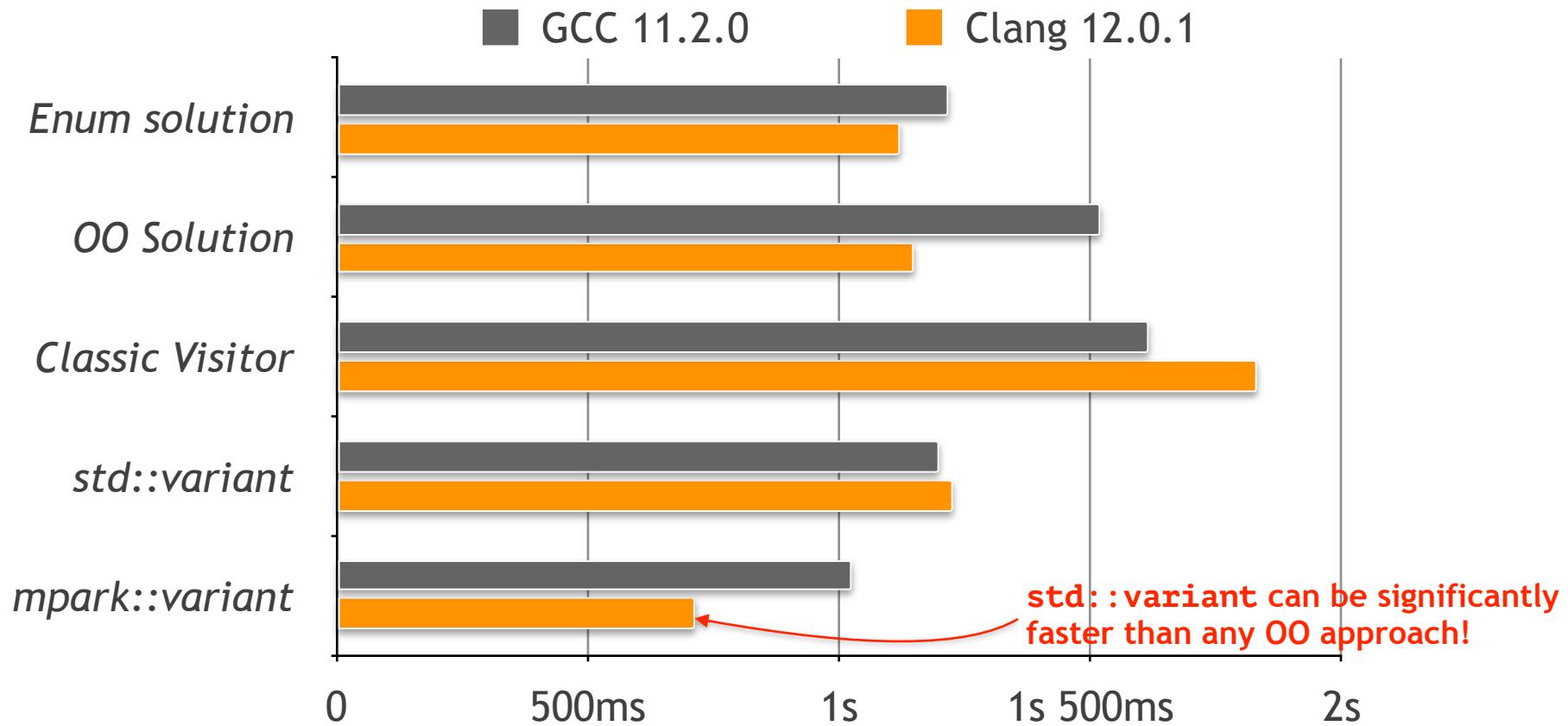
Performance ... *sigh*

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

Performance Comparison

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

Performance Comparison



Unknown Reviewer on Design Patterns



"I believe that object-oriented programming and especially its theory is overestimated. ... C++ always had templates, and now also has std::variant, which makes most of the use of inheritance unnecessary."

(Unknown Reviewer)

A “Modern C++” Solution: A Second Look

```
private:  
    double side;  
    // ... Remaining data members  
};  
  
using Shape = std::variant<Circle,Square>;  
  
class Draw  
{  
public:  
    void operator()( Circle const& ) const;  
    void operator()( Square const& ) const;  
    // ...  
};  
  
void drawAllShapes( std::vector<Shape> const& shapes )  
{  
    for( auto const& s : shapes )  
    {  
        std::visit( Draw{}, s );  
    }  
}  
  
int main()  
{  
    using Shapes = std::vector<Shape>;
```

A “Modern C++” Solution: A Second Look

```
private:  
    double side;  
    // ... Remaining data members  
};  
  
using Shape = std::variant<Circle,Square>;
```

High level (stable, low dependencies)

Low level (volatile, malleable, high dependencies)

Architectural Boundary

```
class Draw  
{  
public:  
    void operator()( Circle const& ) const;  
    void operator()( Square const& ) const;  
    // ...  
};  
  
void drawAllShapes( std::vector<Shape> const& shapes )  
{  
    for( auto const& s : shapes )  
    {  
        std::visit( Draw{}, s );  
    }  
}
```

A “Modern C++” Solution: A Second Look

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

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

Note that the abstraction is on my side of the code. You cannot add any more shape type!

My Code

Architectural Boundary

Your Code

```
class Draw  
{  
public:  
    void operator()( Circle const& ) const;  
    void operator()( Square const& ) const;  
    // ...  
};  
  
void drawAllShapes( std::vector<Shape> const& shapes )  
{  
    for( auto const& s : shapes )  
    {  
        std::visit( Draw{}, s );  
    }  
}
```

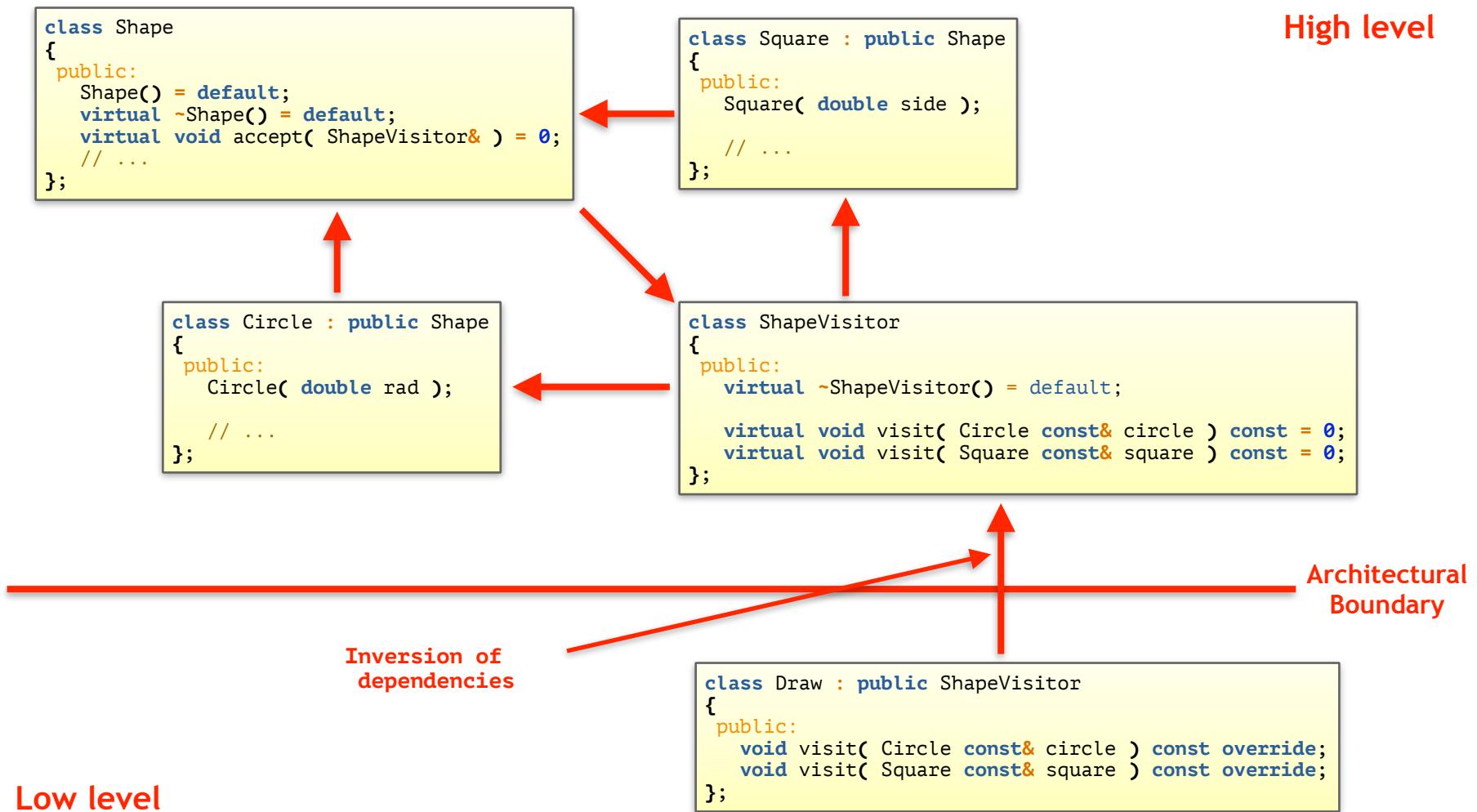
There Is No Silver Bullet



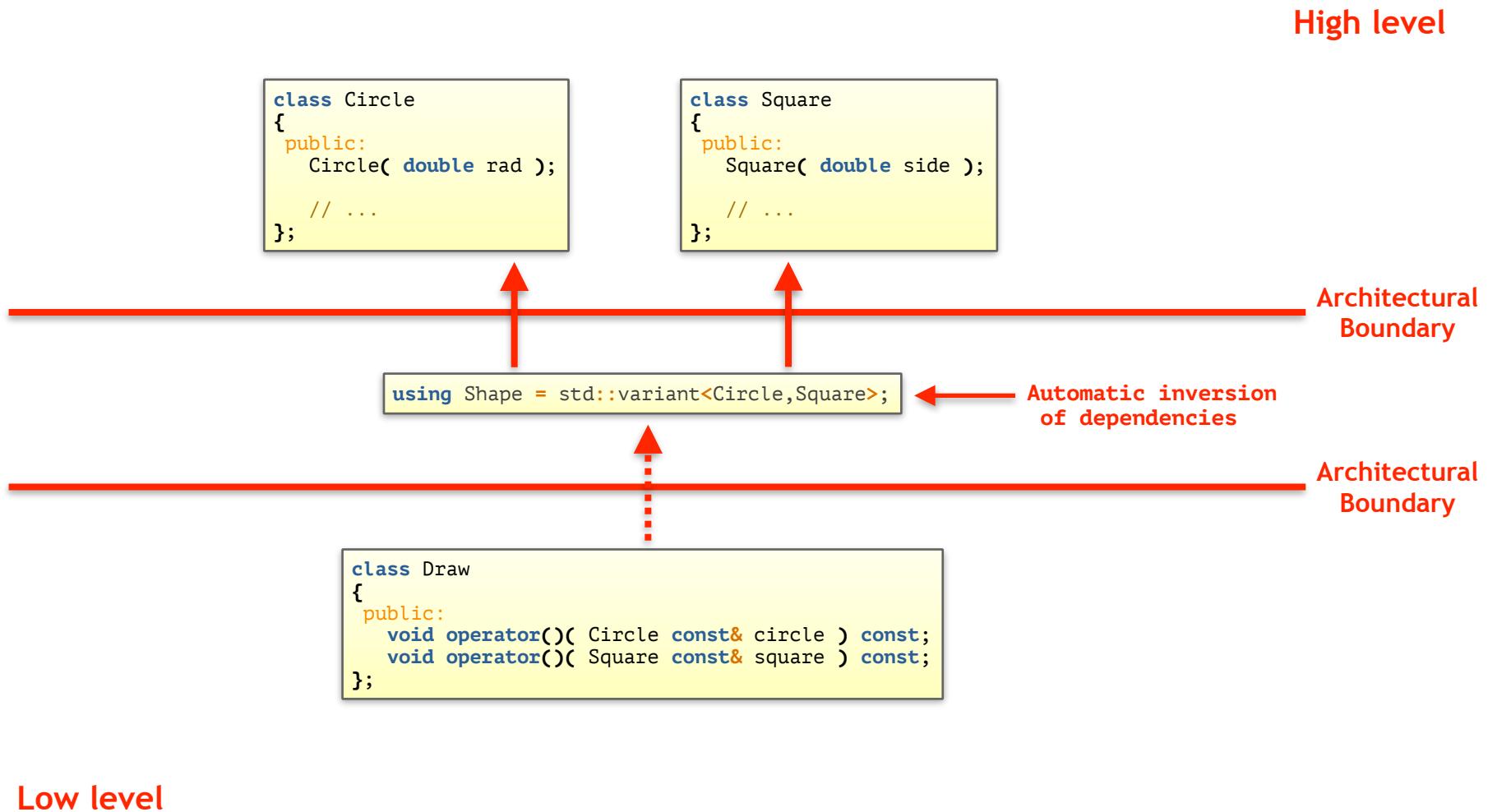
"For every upside, there is a downside."
(Neal Ford, Mark Richards, Head First Software Architecture)



Dependency Structure (Cyclic Visitor)



Dependency Structure (Cyclic Visitor)



The Myth: std::variant is Static Polymorphism

The Myth: std::variant is Static Polymorphism



”... A great many tutorials exist for replacing virtual functions with compile-time polymorphism mechanisms, such as std::variant and templates. ...”

(Abstract of CppCon talk)

The Myth: std::variant is Static Polymorphism



“... This talk will explore techniques for replacing runtime polymorphism with compile-time polymorphism such that virtual functions are never necessary. ...”

(Abstract of CppCon talk)

std::variant is Runtime Polymorphism!

std::variant is a runtime polymorphism mechanism!

It's content is only known at runtime, i.e. it uses runtime dispatch.

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**.

YouTube DE Search

C++'s evolution priorities

Historical

The slide features two large triangles side-by-side. The left triangle is blue and labeled 'Historical', containing the words 'add things', 'fix things', and 'simplify'. The right triangle is yellow and labeled 'A future worth considering?', 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.

add things
fix things
simplify

add things
fix things
simplify

A future worth considering?

5

Herb Sutter

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

Video Sponsorship Provided By:

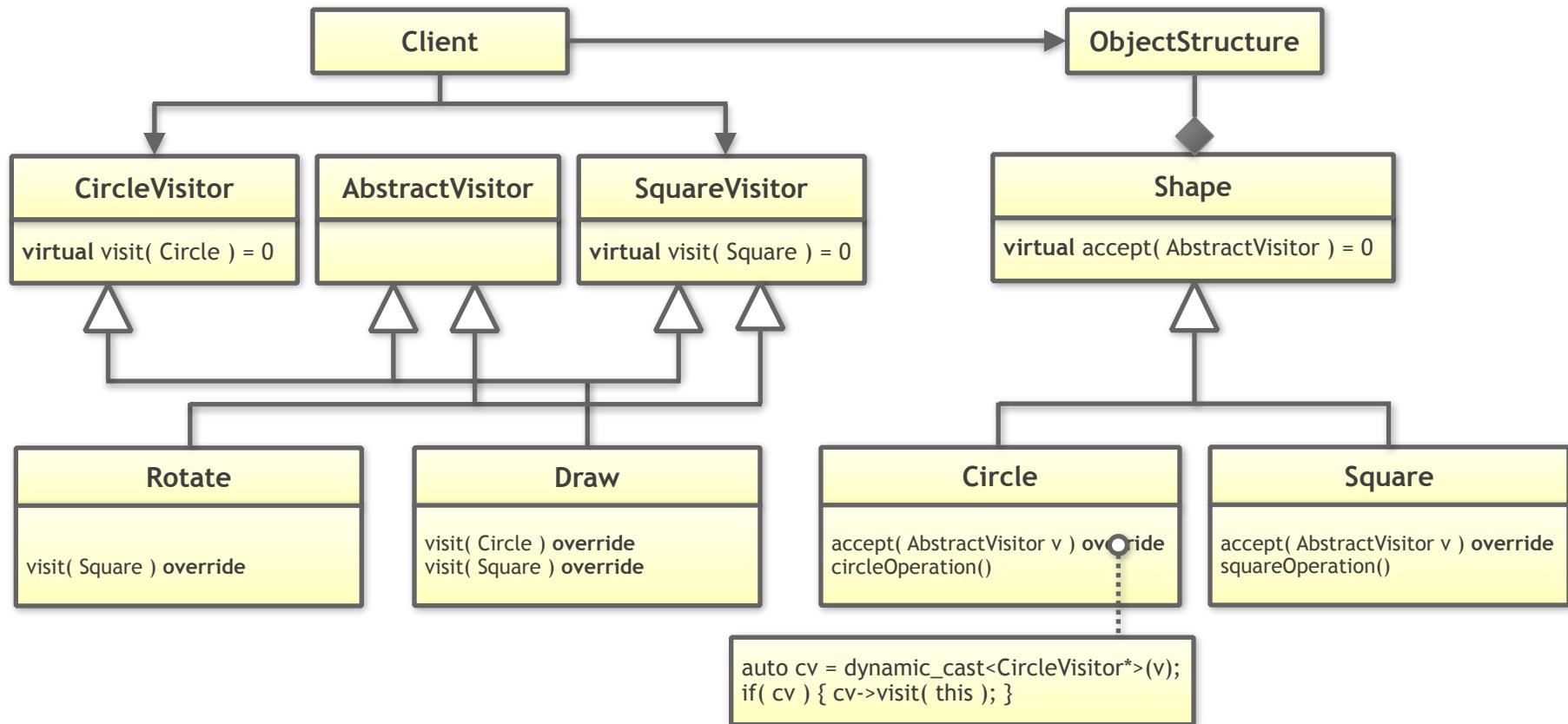
ansatz

CC BY SA

The Acyclic Visitor Design Pattern

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

The Acyclic Visitor Design Pattern



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

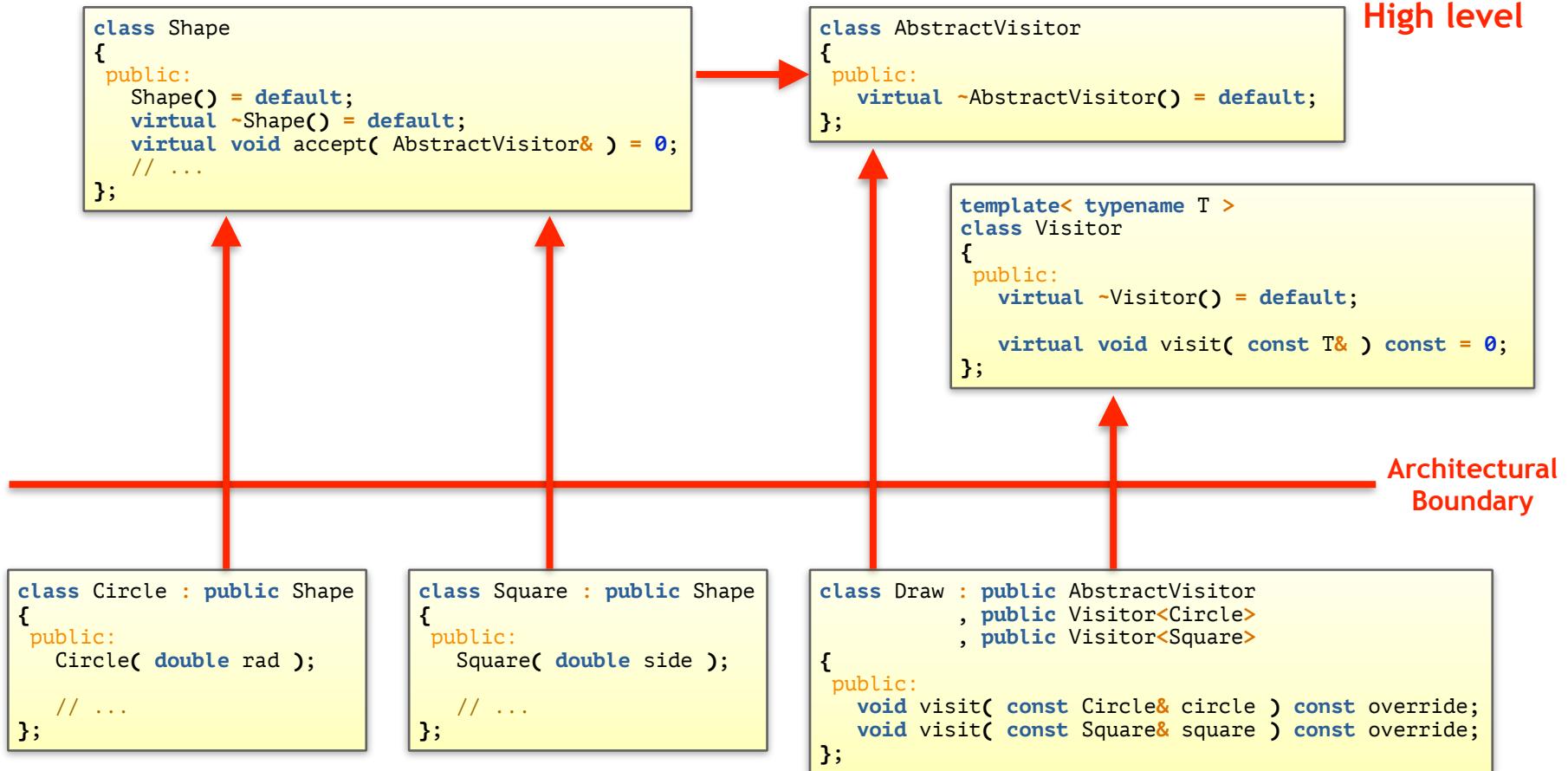
```
void drawAllShapes( std::vector<std::unique_ptr<Shape>> const& shapes )
{
    for( auto const& s : shapes )
    {
        s->accept( Draw{} )
    }
}

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 );
}
```

Dependency Structure (Acyclic Visitor)



Low level

std::variant – Return Values and Parameters

Task (2_Cpp_Software_Design/Visitor/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.

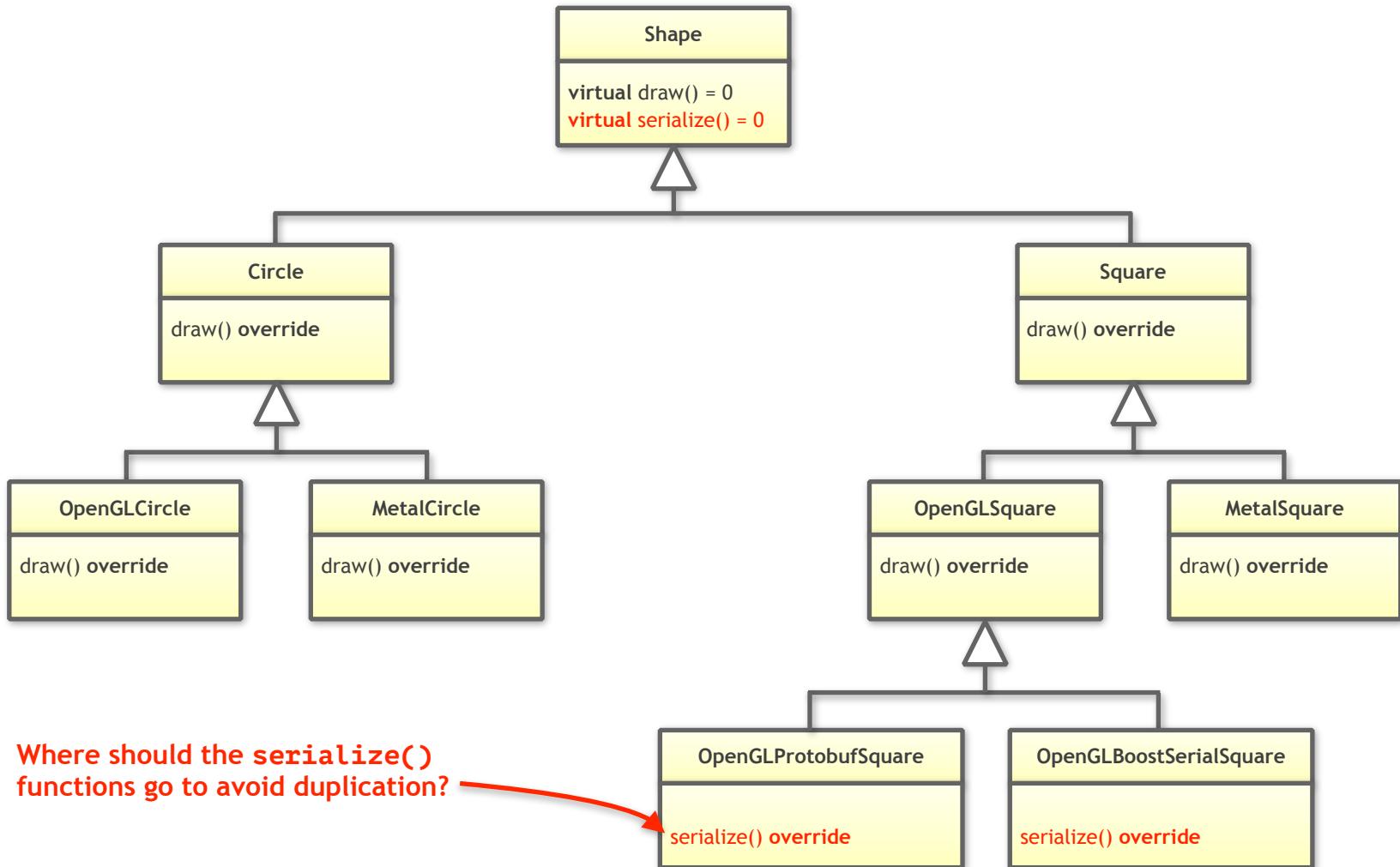
Guideline: Don't use the Visitor design pattern when you need to frequently extend the set of types.

Guideline: Avoid the over-/abuse of inheritance.

Guideline: Prefer multi-paradigm solutions.

2.4. Strategy

A Design Challenge



A Design Challenge

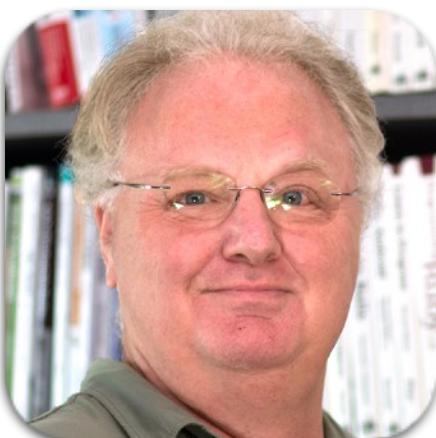
```
class OpenGLProtobufSquare : public Square
{
public:
    // ...
    virtual void draw( Screen& s, /*...*/ ) const;
    virtual void serialize( ByteStream& bs, /*...*/ ) const;
    // ...
};
```

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

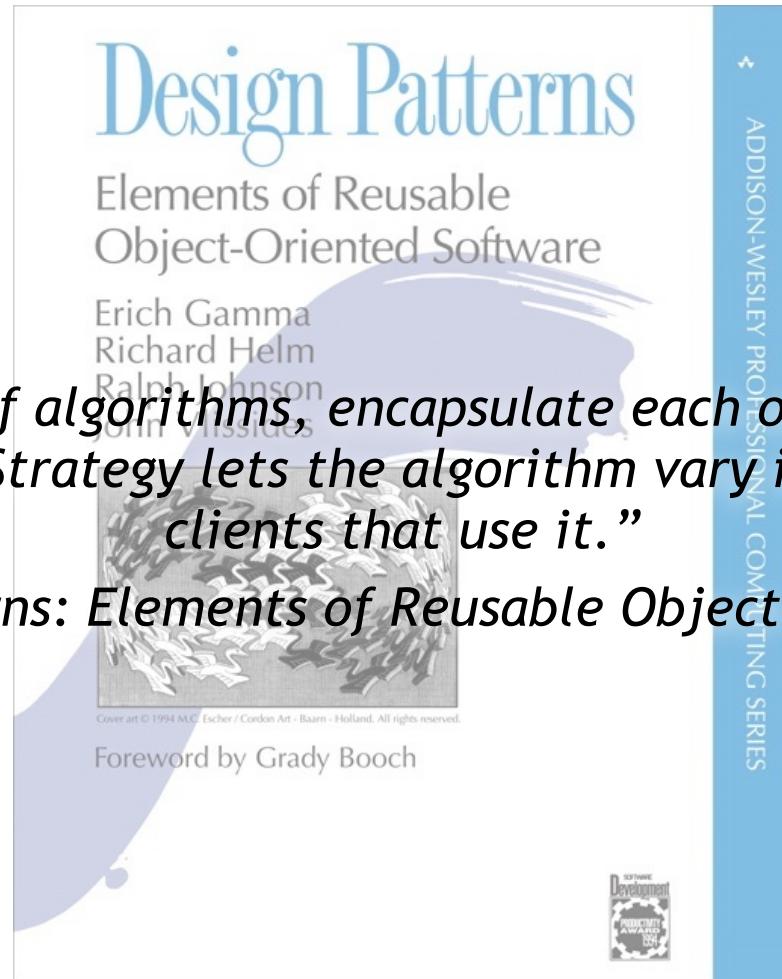


"Inheritance is Rarely the Answer."
(Andrew Hunt, David Thomas, The Pragmatic Programmer)

Guidelines

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

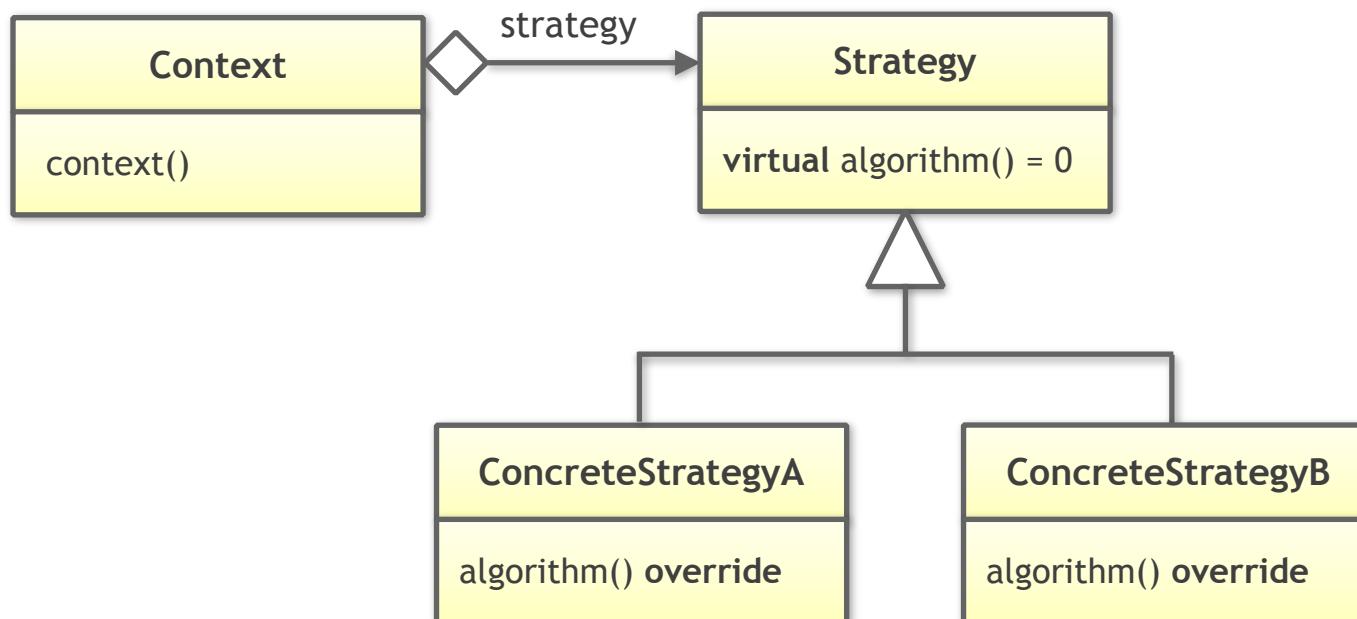
The Intent



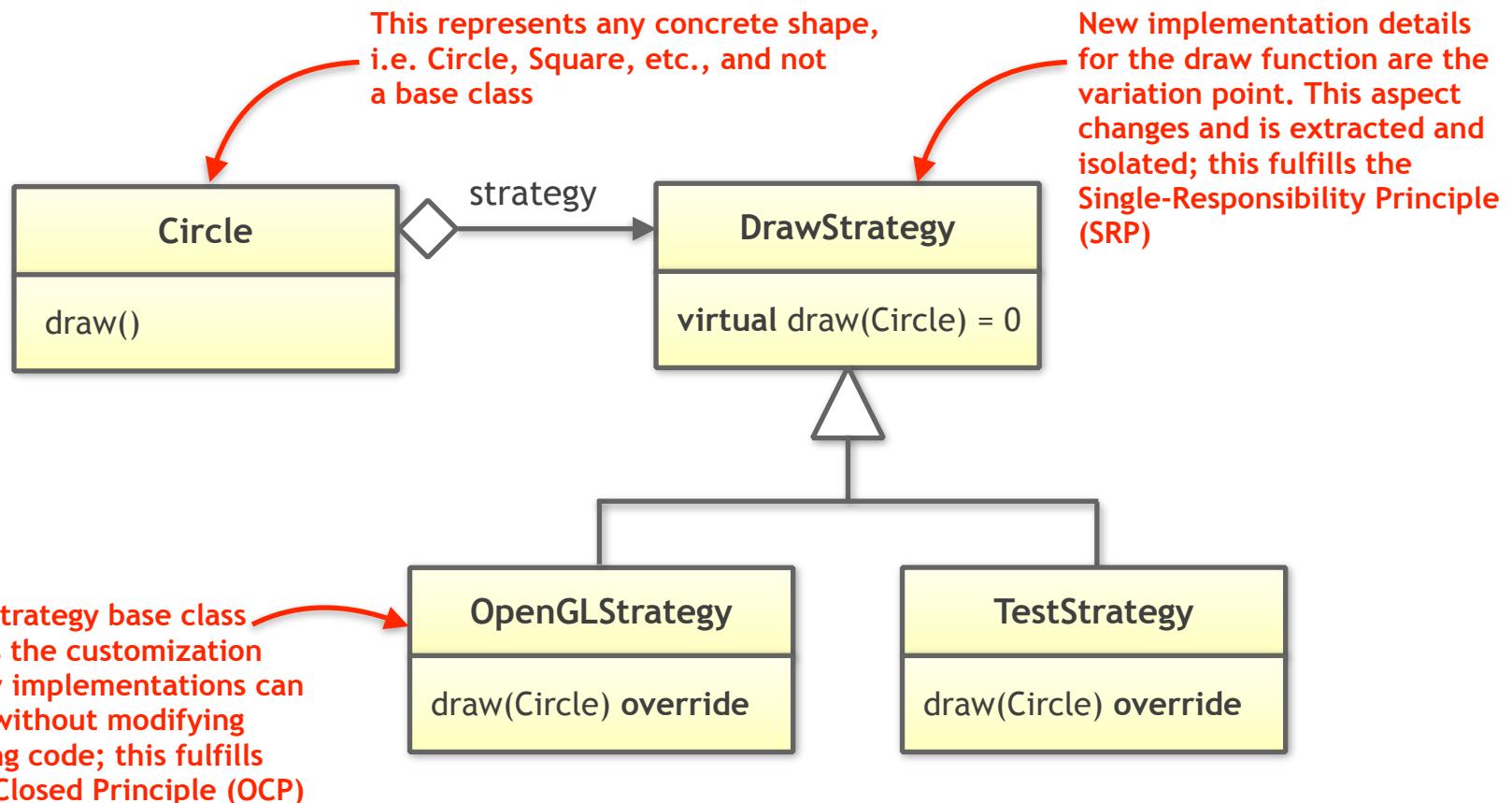
"Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it."

(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 (2_Cpp_Software_Design/Strategy/Strategy_Classic):

Step 1: Refactor the given Shape hierarchy by means of the strategy design pattern to enable the runtime configuration of the draw() operation.

Step 2: Switch from one to another graphics library. Discuss the feasibility of the change: how easy is the change? How many pieces of code on which level of the architecture have to be touched?

Step 3: Add an area() operations for all shapes. How easy is the change? How many pieces of code on which level of the architecture have to be touched? Hint: the area of a circle is $\text{radius} * \text{radius} * \text{M_PI}$, the area of a square is $\text{side} * \text{side}$.

The Classic Strategy Design Pattern

Task (2_Cpp_Software_Design/Strategy/Strategy_Refactoring):

Step 1: Refactor the classic Strategy solution by a value semantics based solution. Note that the general behavior should remain unchanged.

Step 2: Switch from one to another graphics library. Discuss the feasibility of the change: how easy is the change? How many pieces of code on which level of the architecture have to be touched?

Step 3: Add an `area()` operations for all shapes. How easy is the change? How many pieces of code on which level of the architecture have to be touched? Hint: the area of a circle is `radius*radius*M_PI`, the area of a square is `side*side`.

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  
        , drawer{ 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
        , drawer{ 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> drawer;
};

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
        , drawer{ 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> drawer;
};

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> drawer;
};

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

```
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>() ) );  
  
    // 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
        , drawer{ 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> drawer;
};

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
        , drawer{ 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> drawer;
};

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

A Strategy-Based Solution

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
        , drawer{ 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
        , drawer{ 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
        , drawer{ 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
        , drawer{ 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 drawer;
};
```

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

A “Modern C++” Solution

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

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<Circle>( 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 shapes  
    for( auto const& s : shapes )  
    {  
        s->draw();  
    }  
}
```

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 {
        center = Vector2D{ sin(angle), cos(angle) } * radius + center;
    }

private:
    double radius;
    DrawCircleStrategy drawStrategy;
    Vector2D center;
};
```

A “Modern C++” Solution

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

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
        , drawer{ 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 drawer;
};

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

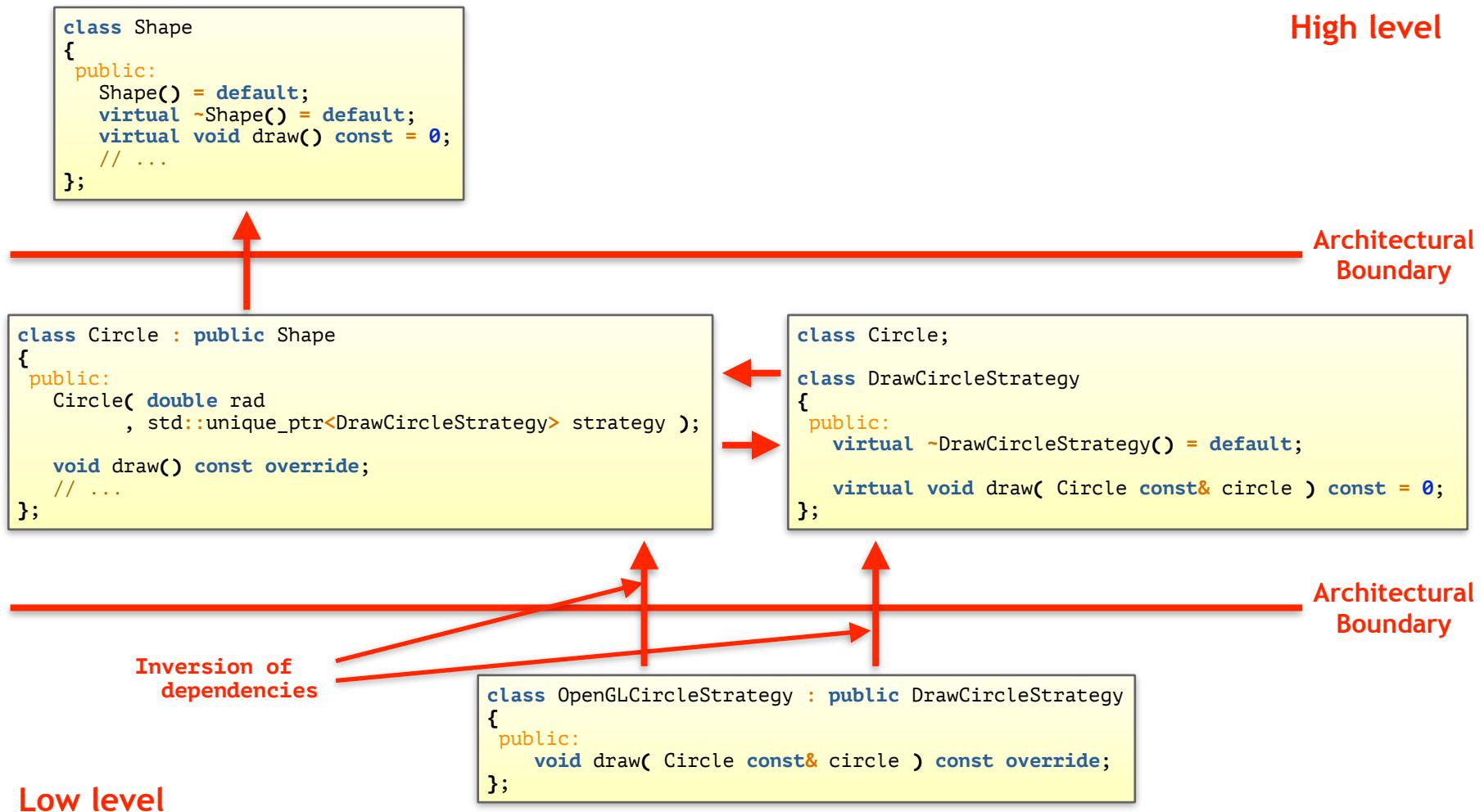
A “Modern C++” Solution

```
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 );
}
```

Dependency Structure



Dependency Structure

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

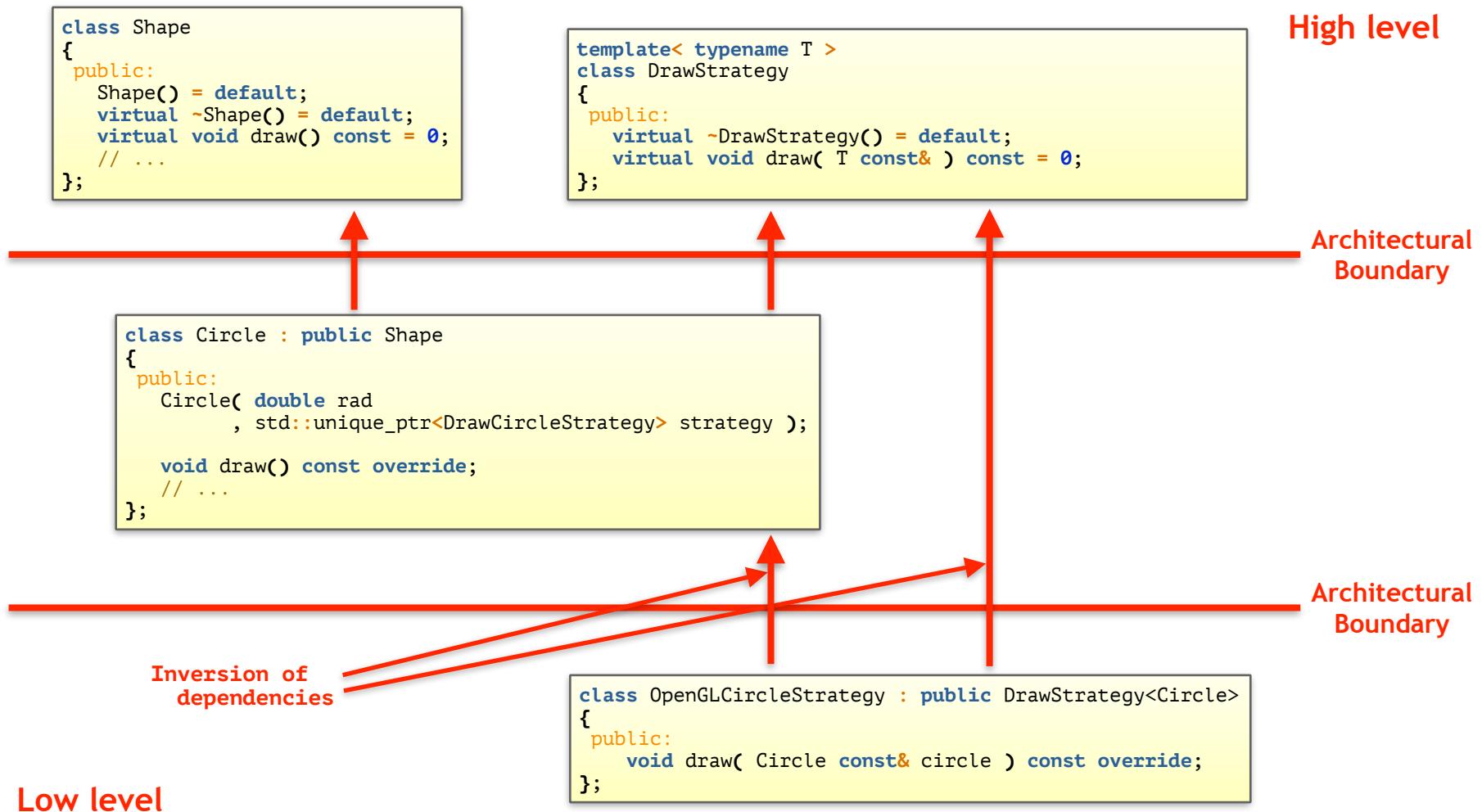
template< typename T >
class DrawStrategy
{
    public:
        virtual ~DrawStrategy() = default;
        virtual void draw( T const& circle ) const = 0;
};

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

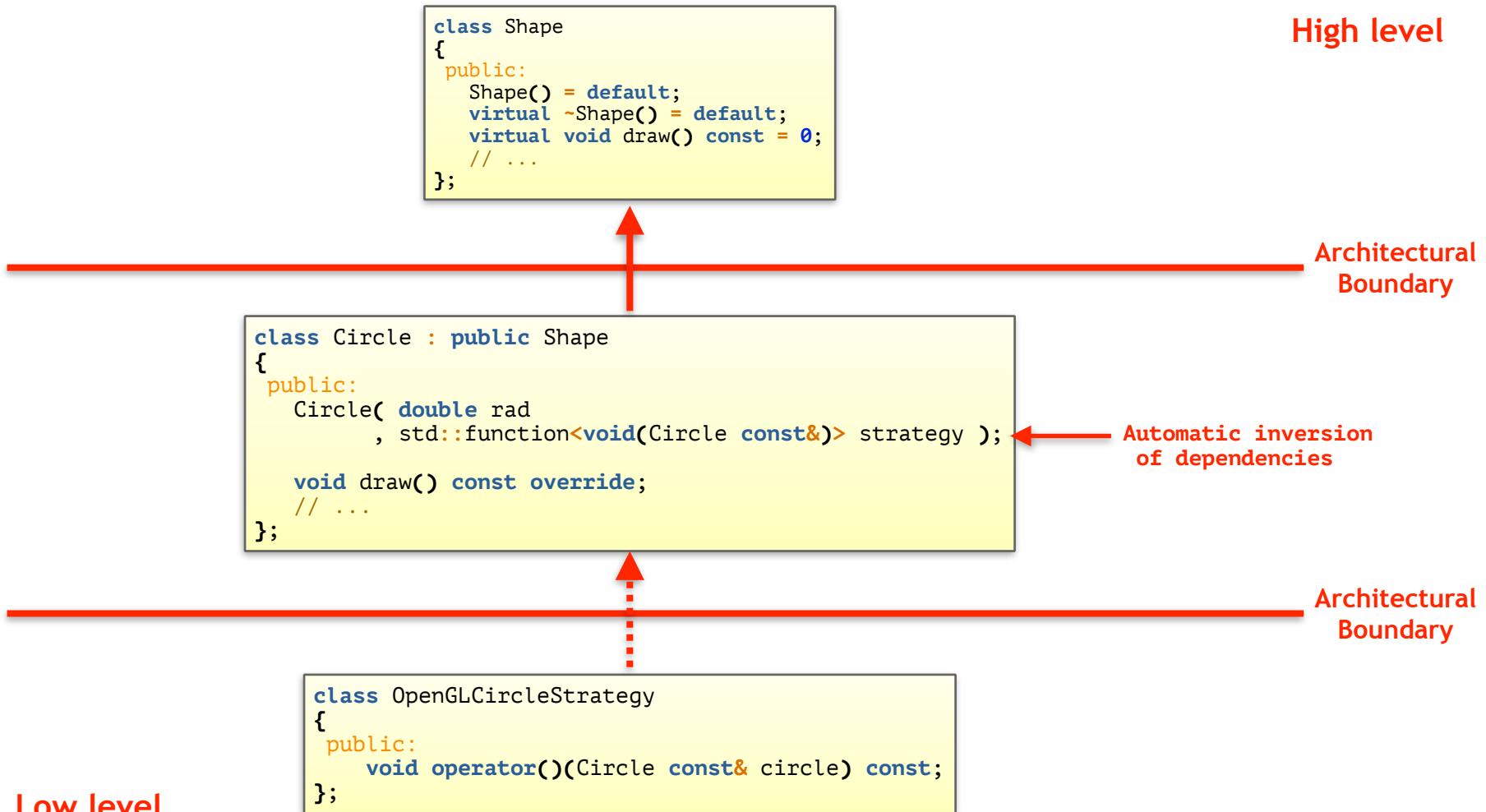
#include <Circle.h>
#include <DrawStrategy.h>
// ... Potentially some OpenGL-related header

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

Dependency Structure



Dependency Structure



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**.

The Classic Strategy Design Pattern

Task (2_Cpp_Software_Design/Strategy/UniquePtr_Strategy): Replace the hard-coded call to `delete()` with a template-based Strategy implementation.

The Classic Strategy Design Pattern

Task (2_Cpp_Software_Design/Strategy/FixedVector_Strategy):

Step 1: Replace the hard-coded logging mechanism by means of a template-based Strategy implementation. Demonstrate how this helps to invert the dependencies.

Step 2: Replace the hard-coded error handling mechanism (i.e. assert()) by means of a template-based Strategy implementation.

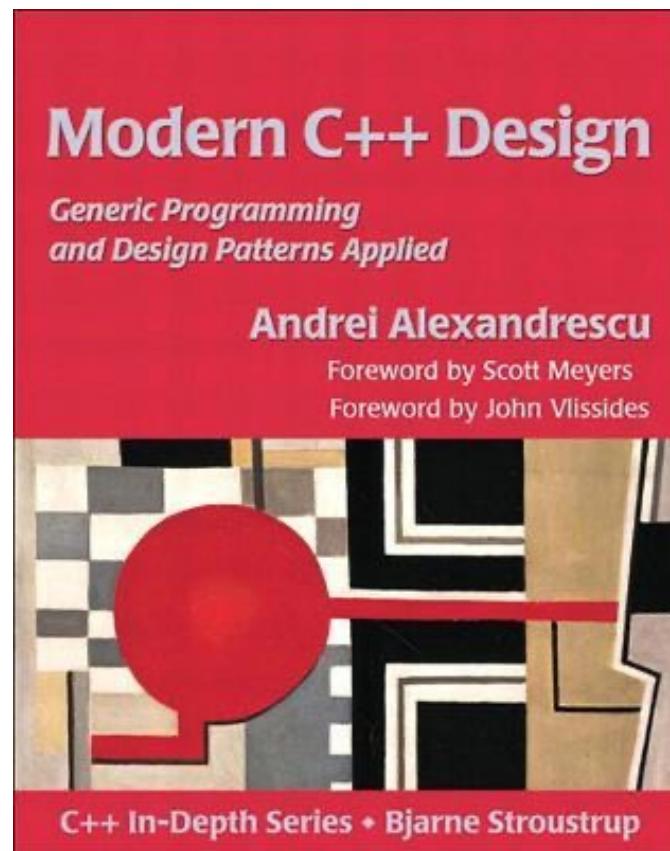
Guidelines

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;
```

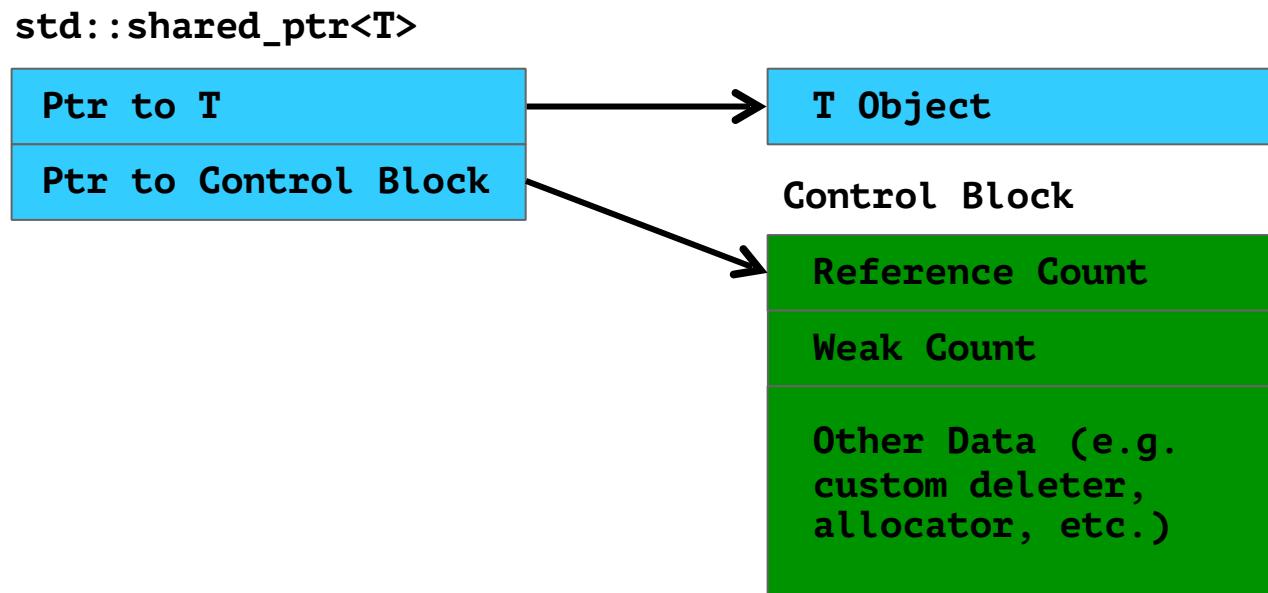
Example: Smart Pointers

```
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

32

The Expert's Attitude



A large slide with a teal header and footer. The main content area is white with the text "{OOP}" in a large, bold, black font. In the teal footer, the title "Back to Basics: Object-Oriented Programming" is displayed in white. To the right, smaller white text provides the presentation details: "Presentation by Jon Kalb", "CppCon 2019, 2019-09-17", and "thanks Andrei, Herb, & Scott". The bottom of the slide features a standard video player control bar.

The Expert's Attitude



A screenshot of a web article from THE NEW STACK. The header includes the site's logo, "THE NEW STACK", and navigation links for "Ebooks", "Podcasts", "Events", and "Newsletter". Below that is a secondary navigation bar with categories: "Architecture", "Development", and "Operations". The main headline, "CULTURE / DEVELOPMENT Why Are So Many Developers Hating on Object-Oriented Programming?", is prominently displayed in large, bold, black text. A smaller text below the headline indicates it was published on "21 Aug 2019 12:00pm" by "David Cassel". To the right of the headline is a painting of two shirtless men in a dynamic pose, one in blue pants and the other in red shorts, set against a red background.

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

Object-oriented programming is not what the cool kids are doing in C++. They are doing things at compile time, functional programming, ...
Object-oriented programming, this is so 90s ...

A painting depicting two shirtless men in a dynamic, expressive pose. One man is leaning forward, wearing blue pants, while the other is more upright, wearing red shorts. The background is a rich red with abstract, textured shapes and brushstrokes, suggesting a cityscape or industrial scene.

The Expert's Attitude

The image is a screenshot of a video from CppCon 2018. The video title is "Can a browser engine be successful with data-oriented design?". The speaker is Stoyan Nikolov, wearing a light-colored sweater and glasses. The slide text reads "OOP is dead, long live Data-oriented design". The video player interface shows a progress bar at 2:49 / 1:00:45, and the bottom right corner has the CppCon.org logo with video controls.

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

STOYAN NIKOLOV

OOP is dead, long live
Data-oriented design

CppCon 2018 | @stoyann 3

2:49 / 1:00:45

CppCon.org

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://youtu.be/QHnLmvDxGTY?t=2264>)

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?

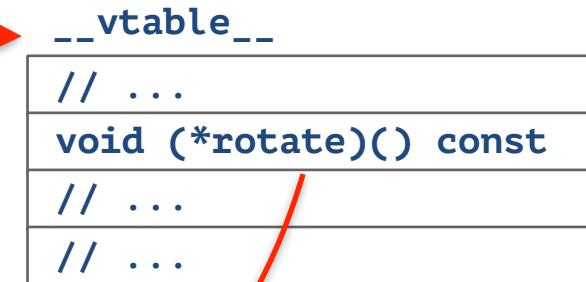
- Does not harmonize with the philosophy of the STL
- Fails to model many sub typing relations (i.e. it is hard)
- Inheritance creates a very tight coupling (second only to friendship)
 - Intrusive: Forces us to inherit, or ...
 - Verbose: Forces us to wrap perfectly good types to conform to a hierarchy
- 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!

Progressive C++

Meeting C++ 2023

Prog C++ - Ivan Čukić

think-cell

Prog C++ is a broad genre of C++ code. The style was an emergence of psychedelic developers who abandoned standard C with classes traditions in favour of instrumentation and compositional techniques more frequently associated with generic, functional, or value-based object oriented coding practices.

PROG C++

INTRODUCTION
WRAPS
SWAPS
STATES
ERRORS
VALUES
SAFETY

Meeting C++ 2023

Ivan Čukić kdab.com, cukic.co



Guidelines

Guideline: Prefer value semantics over reference semantics!

Guideline: Try to reduce the use of pointers!

2.7. 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 3rd 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 3rd Pattern Languages of Programming Conference, Alton 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 manager. If I need to polymorphically treat classes from different libraries, I will want to change existing code.

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

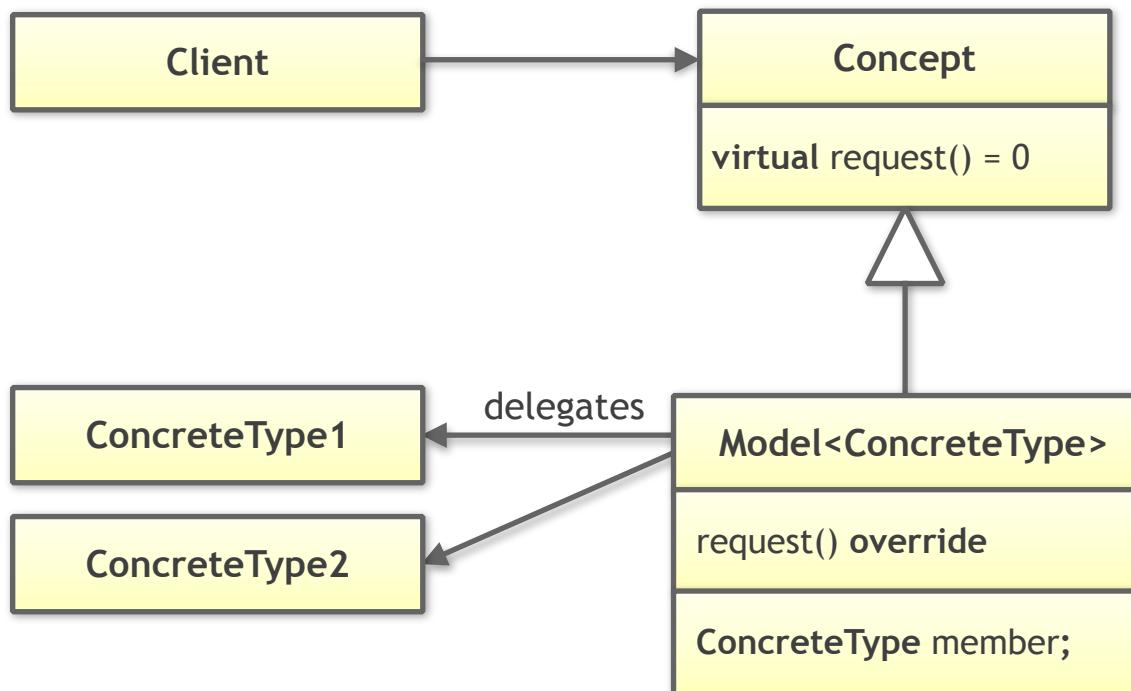
```
1. Sock_Stream::this = 0x7e7393ab, handle_ = {-1}
2. SOCK_Acceptor::this = 0x2d49a45b, handle_ = {-1}
3. int main (void) {
4.     SOCK_Stream stream; // Automatic storage
5.     INET_Addr::this = 0x3c48a432,
6.     new INET_Addr::this;
7. }
```

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:

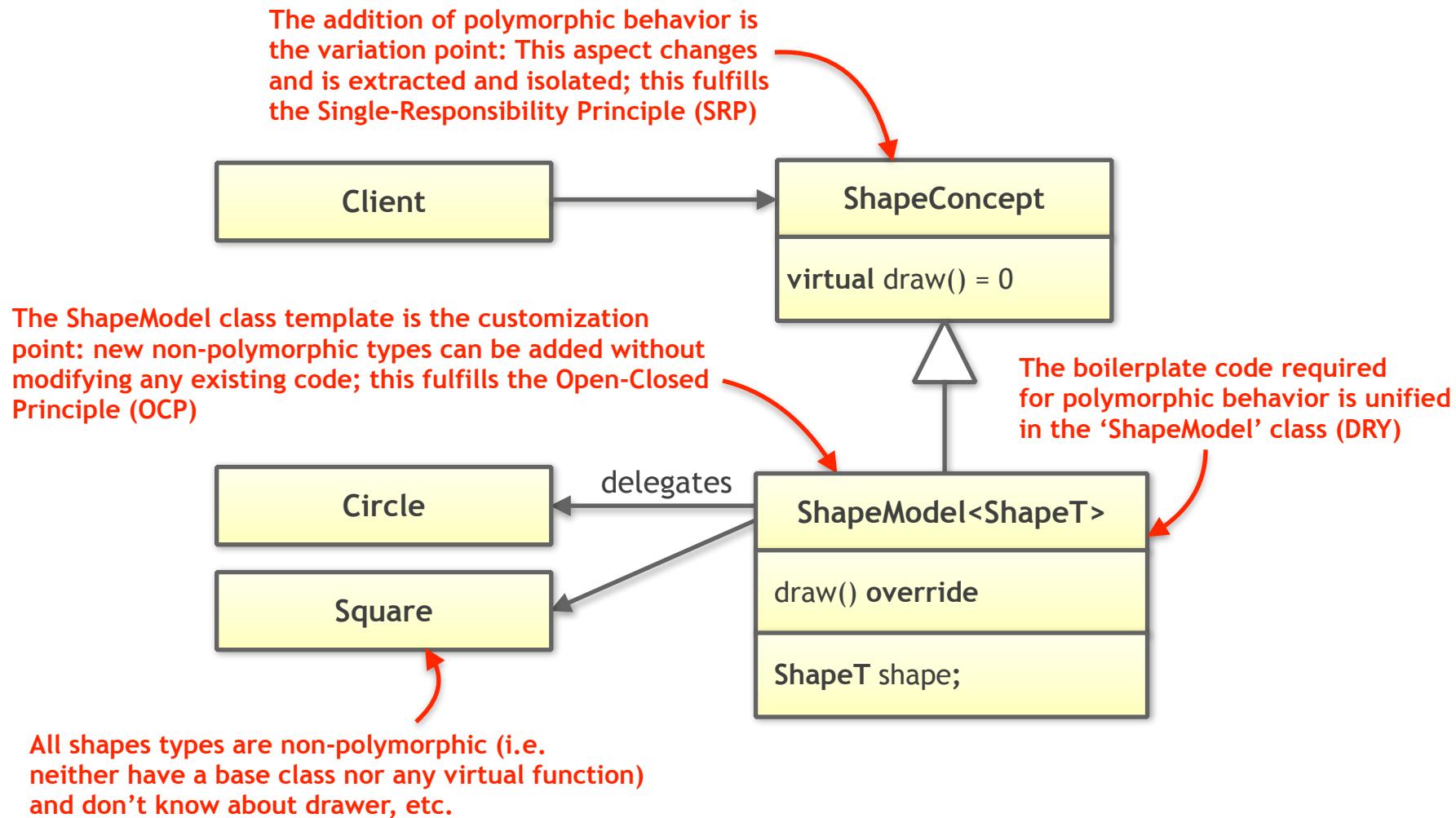
```
Sock_Stream::this = 0x7e7393ab, handle_ = {-1}
SOCK_Acceptor::this = 0x2d49a45b, handle_ = {-1}
INET_Addr::this = 0x3c48a432,
port_ = {0}, addr_ = {0.0.0.0}
```

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 (2_Cpp_Software_Design/External_Polymorphism/Animal_EP):
Create an external hierarchy for animals that represents the polymorphic behavior of animals.

External Polymorphism

**Task (2_Cpp_Software_Design/External_Polymorphism/
ExternalPolymorphism_1):**

Step 1: Refactor the given Shape hierarchy by means of the External Polymorphism design pattern to extract the draw() operation from shapes.

Step 2: Switch from one to another graphics library. Discuss the feasibility of the change: how easy is the change? How many pieces of code on which level of the architecture have to be touched?

External Polymorphism

Task (2_Cpp_Software_Design/External_Polymorphism/ExternalPolymorphism_2):

Step 1: 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.

Step 2: Switch from one to another graphics library. Discuss the feasibility of the change: how easy is the change? How many pieces of code on which level of the architecture have to be touched?

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 );

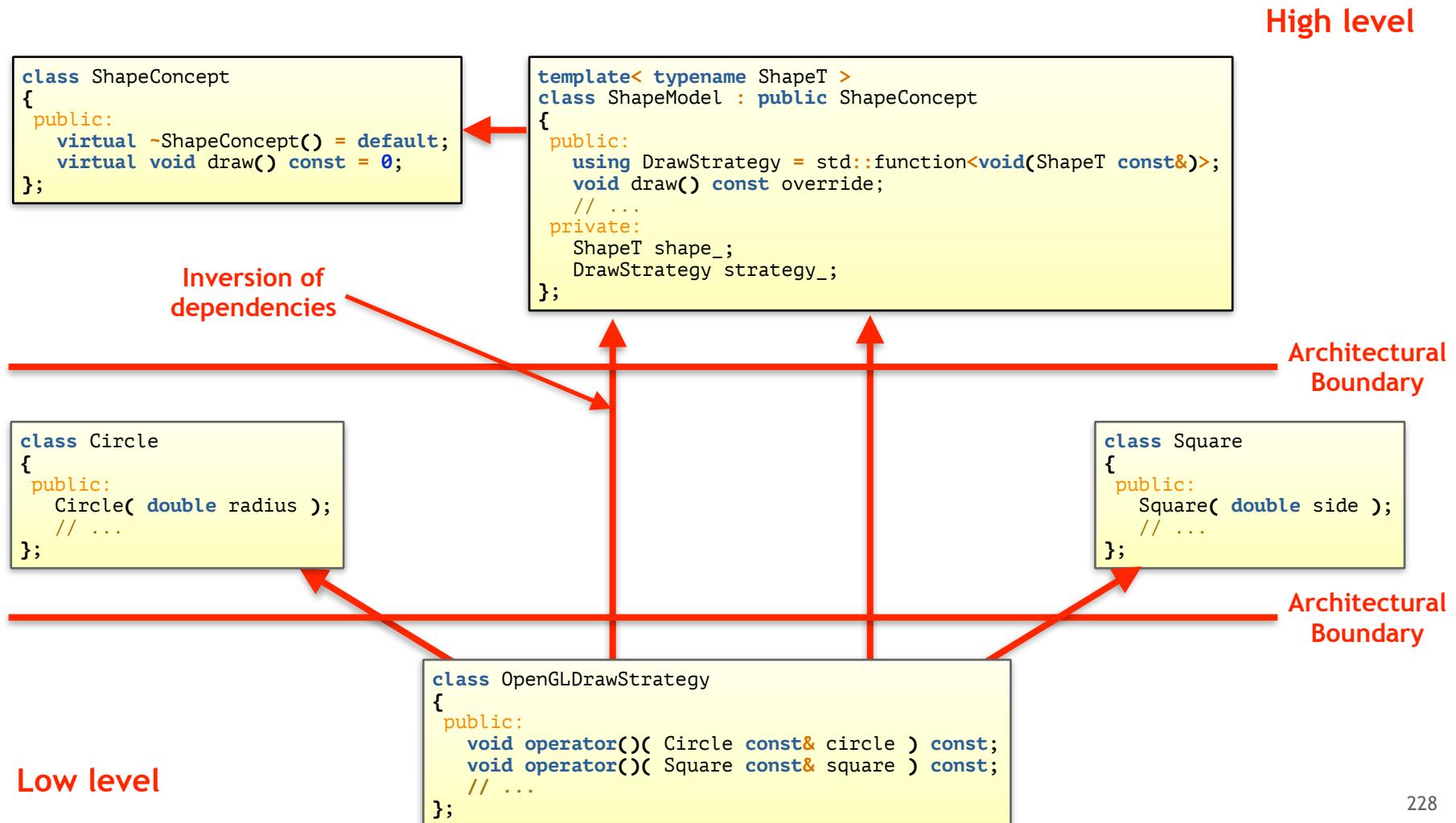
    // Drawing all shapes
    drawAllShapes( shapes );
}
```

External Polymorphism

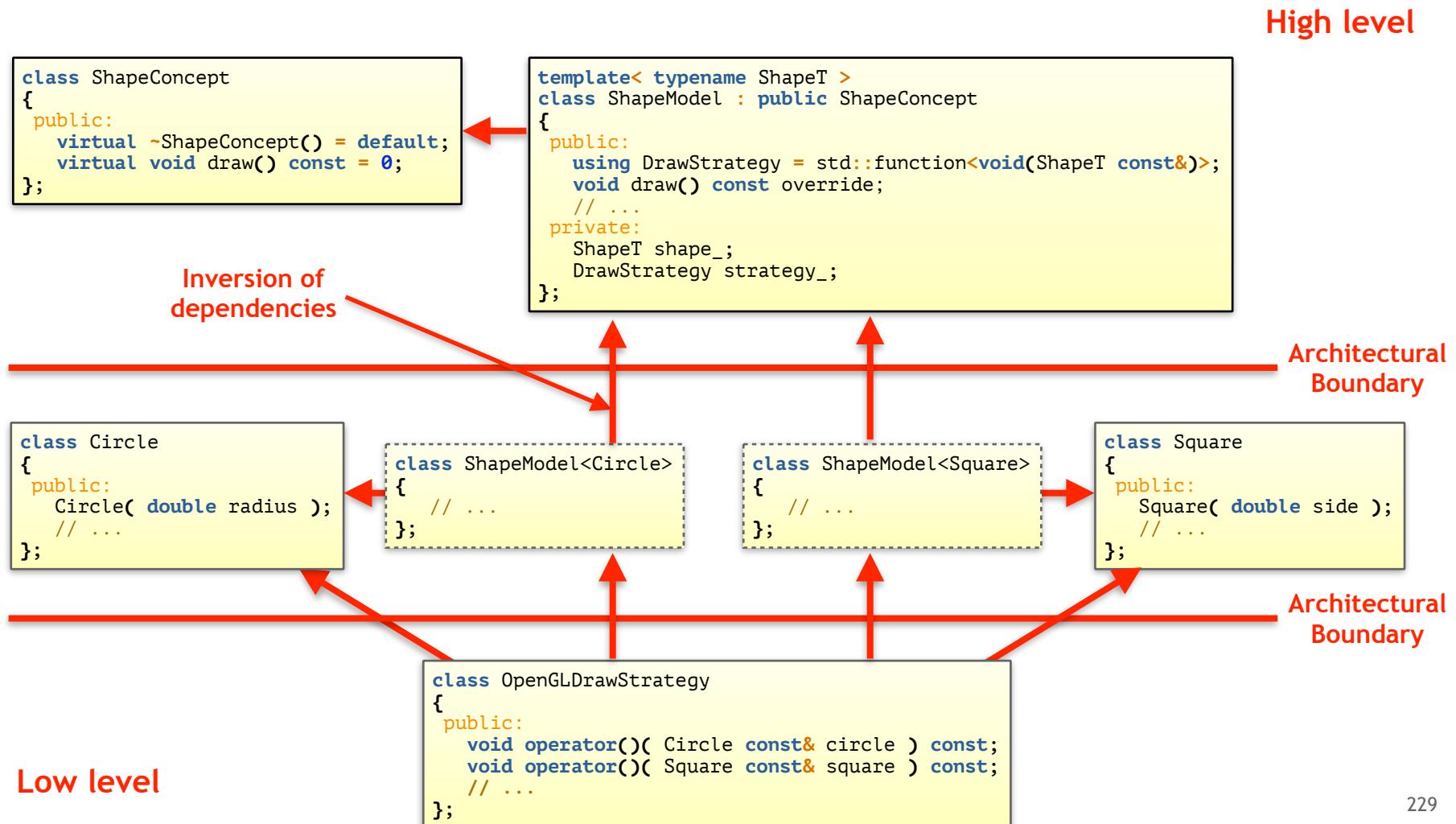
The External Polymorphism design pattern ...

- ... 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).

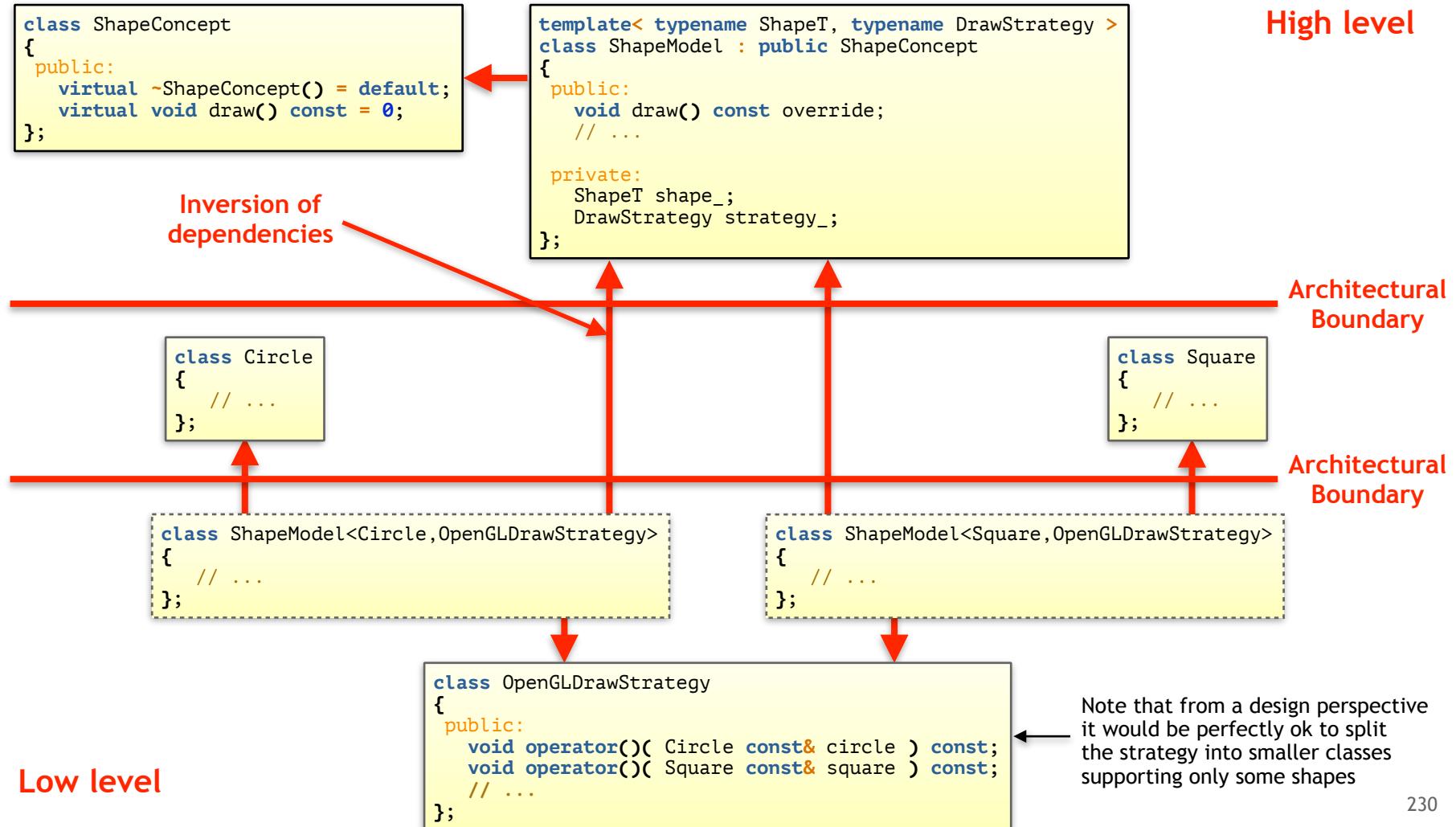
Dependency Structure



Dependency Structure



Dependency Structure



Low level

230

External Polymorphism vs. Strategy

<i>External Polymorphism</i>	<i>Strategy</i>
Extracts the complete polymorphic behaviour	Configures the behavior from outside (dependency injection)
Non-intrusive	Intrusive
Only runtime polymorphism	Runtime or compile time polymorphism
No support by mocking frameworks	Strong support by mocking frameworks

Guidelines

Guideline: Use the External Polymorphism design pattern to extract polymorphic behavior from types or to treat non-polymorphic types polymorphically.

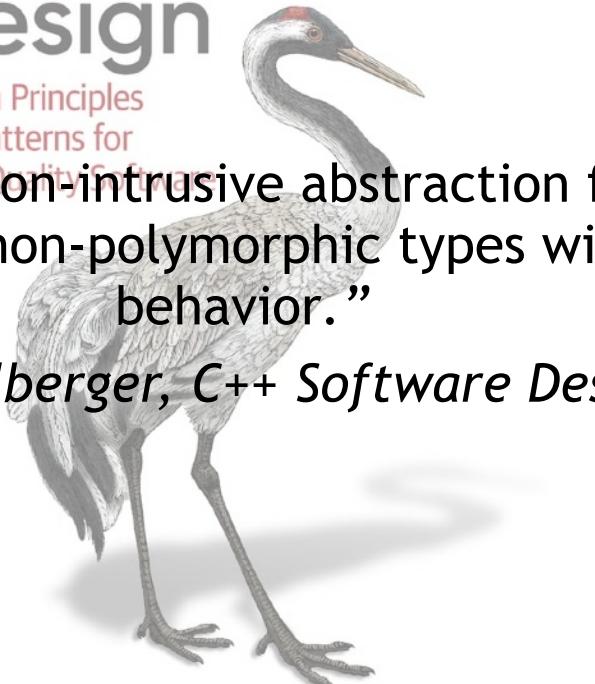
2.8. Type Erasure

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

CC

5

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 of 24:19, and standard controls for play, volume, and settings. A small number "5" 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.



FROM MECHANISM TO METHOD

Valued Conversions

HOW 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>

The screenshot shows a Twitter thread from Eric Niebler (@ericniebler). The first tweet is a quote from him, dated June 19, 2020, at 7:27 PM, using Twitter for Android. It discusses adding type erasure and concepts to C++ instead of virtual functions. The tweet has 6 retweets, 2 quote tweets, and 125 likes. Below this is a reply from Eric Niebler, also dated June 19, 2020, at 7:27 PM. He expresses surprise at the high number of likes and mentions elaborate class hierarchies.

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

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 (2_Cpp_Software_Design/Type_Erasure/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 (2_Cpp_Software_Design/Type_Erasure/Function2): Implement a simplified std::function to demonstrate the type erasure design pattern. Use the void*-based approach.

Applied Type Erasure

Task (2_Cpp_Software_Design/Type_Erasure/Animal_TE):

Step 1: Encapsulate the external AnimalConcept hierarchy in an owning Type Erasure wrapper and refactor the given solution into a value-based solution.

Step 2: Provide a non-owning Type Erasure wrapper that does not allocate dynamically and that does not create a copy of the given animal.

Applied Type Erasure

Task (2_Cpp_Software_Design/Type_Erasure/TypeErasure_1):

Step 1: Refactor the given External Polymorphism solution into Type Erasure. The Shape class may require all types to provide a free_draw() function that draws them to the screen.

Step 2: Switch from one to another graphics library. Discuss the feasibility of the change: how easy is the change? How many pieces of code on which level of the architecture have to be touched?

Applied Type Erasure

Task (2_Cpp_Software_Design/Type_Erasure/TypeErasure_2):

Step 1: 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.

Step 2: Switch from one to another graphics library. Discuss the feasibility of the change: how easy is the change? How many pieces of code on which level of the architecture have to be touched?

std::function – A Simplified Implementation

Task (2_Cpp_Software_Design/Type_Erasure/TypeErasure_SBO_1/2):
Upgrade the given Type Erasure implementation by means of the Small Buffer Optimization (SBO) technique to avoid any dynamic allocation. 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

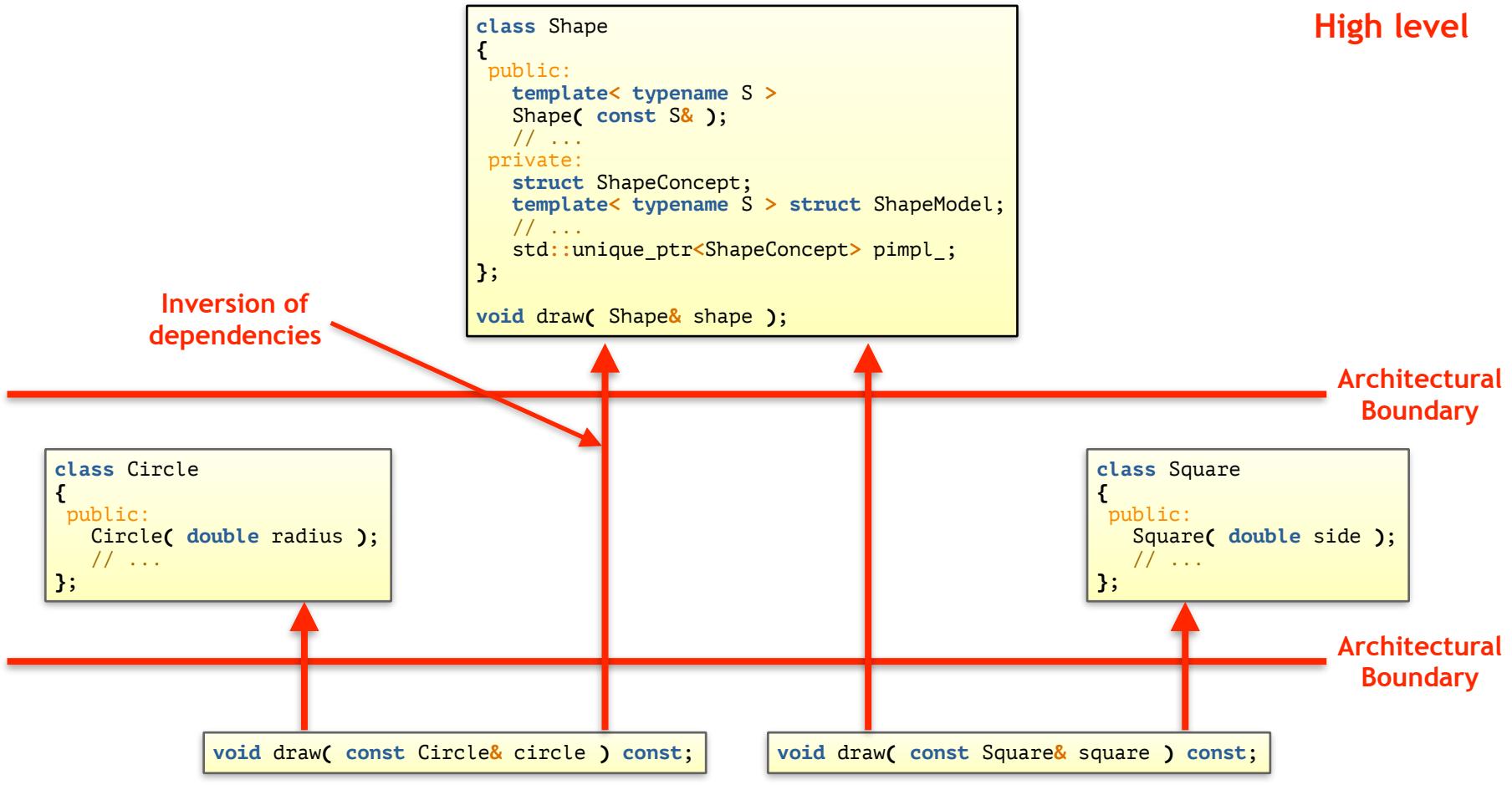
```
for( auto const& shape : shapes )
{
    draw( shape );
}

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 );
}
```

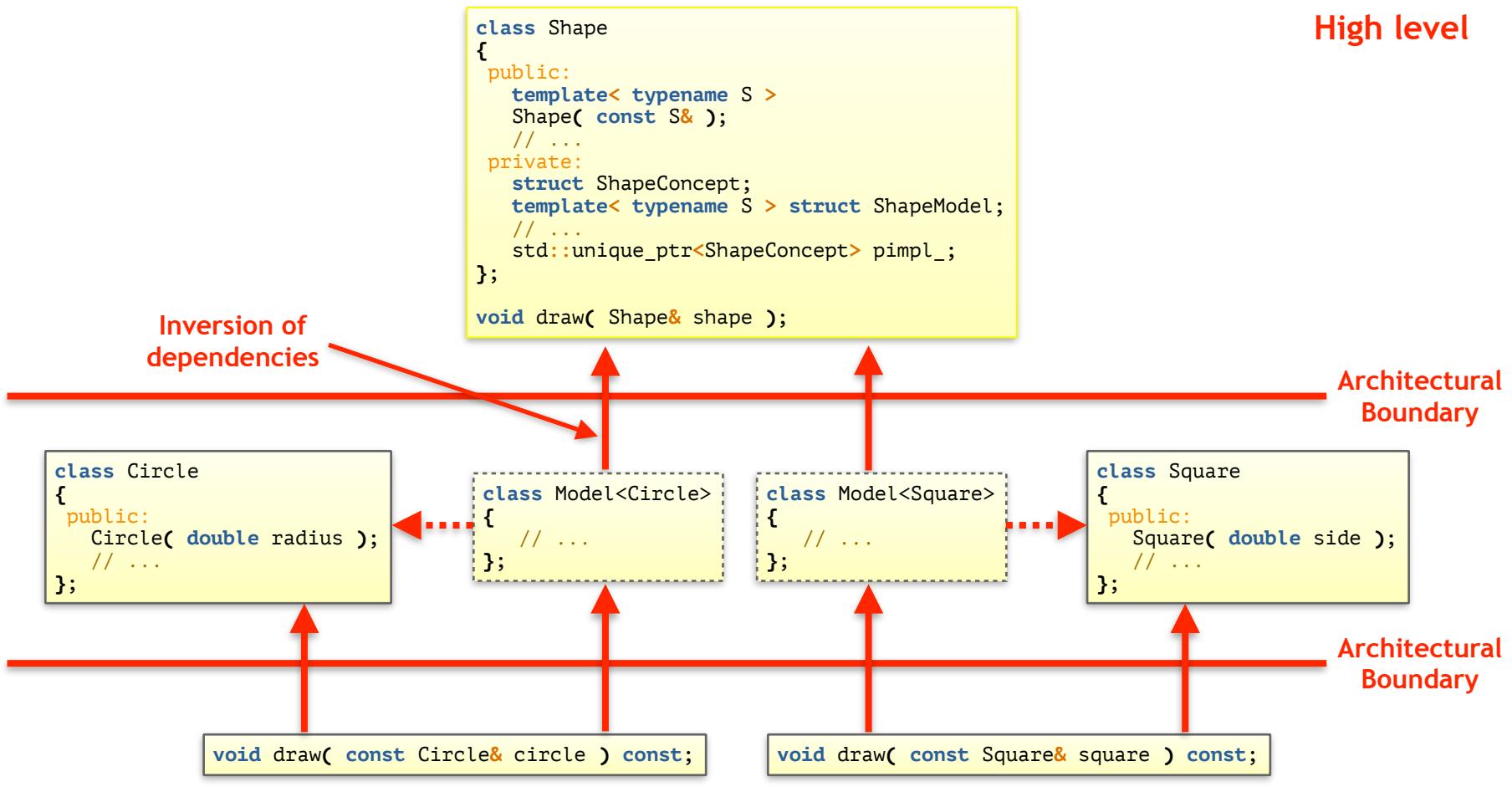
Dependency Structure



Low level

268

Dependency Structure



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( ShapeView );
          // 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 (2_Cpp_Software_Design/Type_Erasure/TypeErasure_Ref):
Implement the ShapeView class, representing a reference to a constant shape, by means of Type Erasure. ShapeView may require all types to provide a free draw() function that draws them to the screen.

Non-Intrusive API Design: Type Erasure

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

private:
    friend void draw( ShapeView 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 (2_Cpp_Software_Design/Type_Erasure/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.

Type Erasure vs. std::variant

<i>Type Erasure</i>	<i>std::variant</i>
Open set of types / Closed set of operations	Open set of operations / Closed set of types
No dynamic allocation with SBO	Never uses dynamic allocation
Manual optimization required (SBO and MVF)	No manual optimization required

A wrong choice particularly hurts in a large code base and if other people would experience a change.

Applied Type Erasure

Task (2_Cpp_Software_Design/Type_Erasure/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 extension of an open set of types.

Guideline: Don't use Type Erasure when you need to frequently extend the closed set of operations.

Guideline: Choose between Type Erasure and Visitor (aka `std::variant`) depending on whether you have an open set of types or an open set of operations.

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

There Is No Silver Bullet



"It's a general mistake to search for a silver bullet."
(Neal Ford)

2.9. Prototype

Motivation

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

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

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

class Sheep : public Animal
{
public:
    void make_sound() 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 make_sound() 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->make_sound();

    // Copying Dolly
    // ????

    return EXIT_SUCCESS;
}
```

Motivation

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

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

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

class Sheep : public Animal
{
public:
    void make_sound() 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 make_sound() 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 make_sound() 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 make_sound() 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->make_sound();  
  
    // Copying Dolly  
    auto dolly2 = dolly->clone();  
  
    return EXIT_SUCCESS;  
}
```

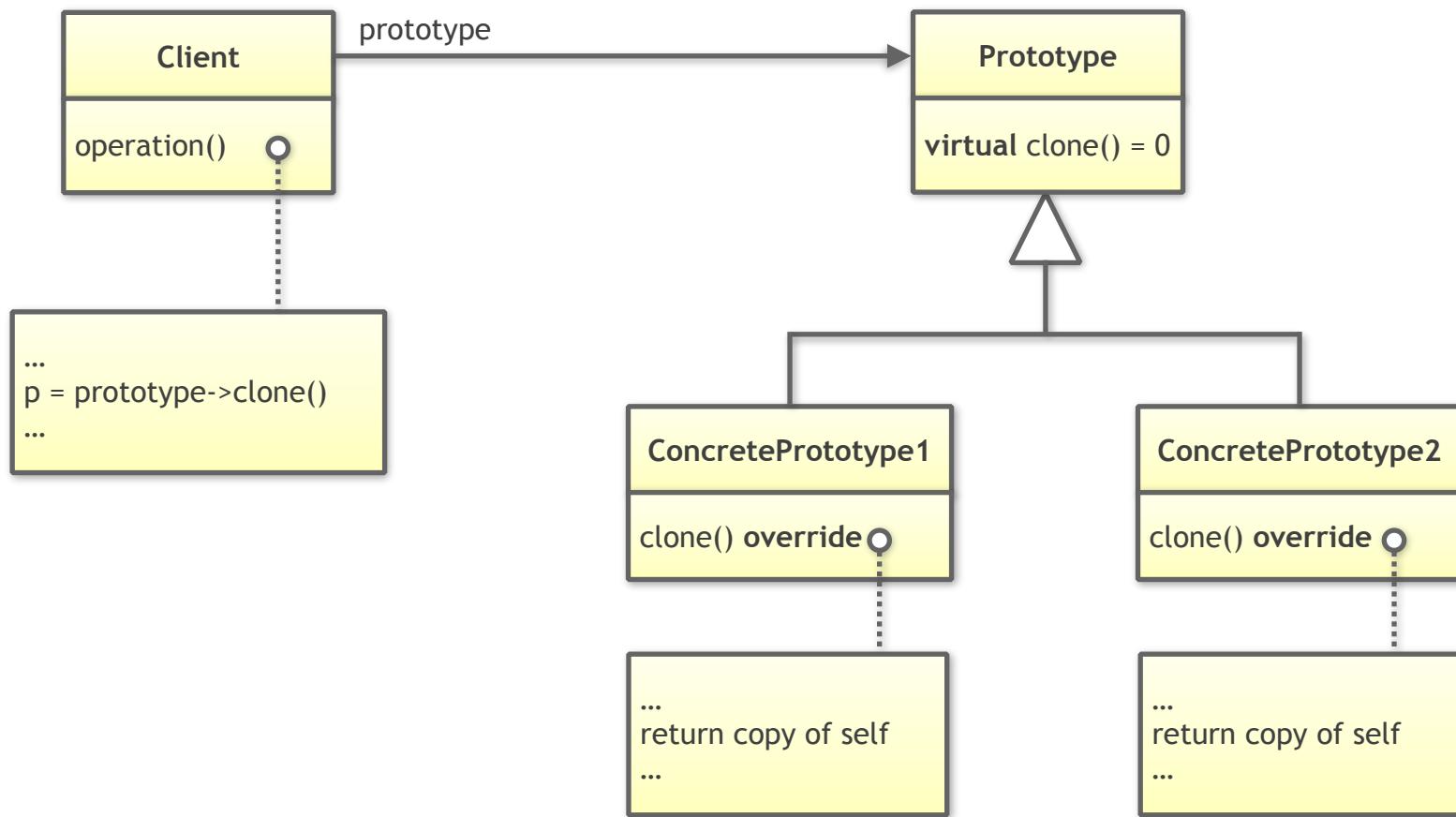
The Intent



"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)

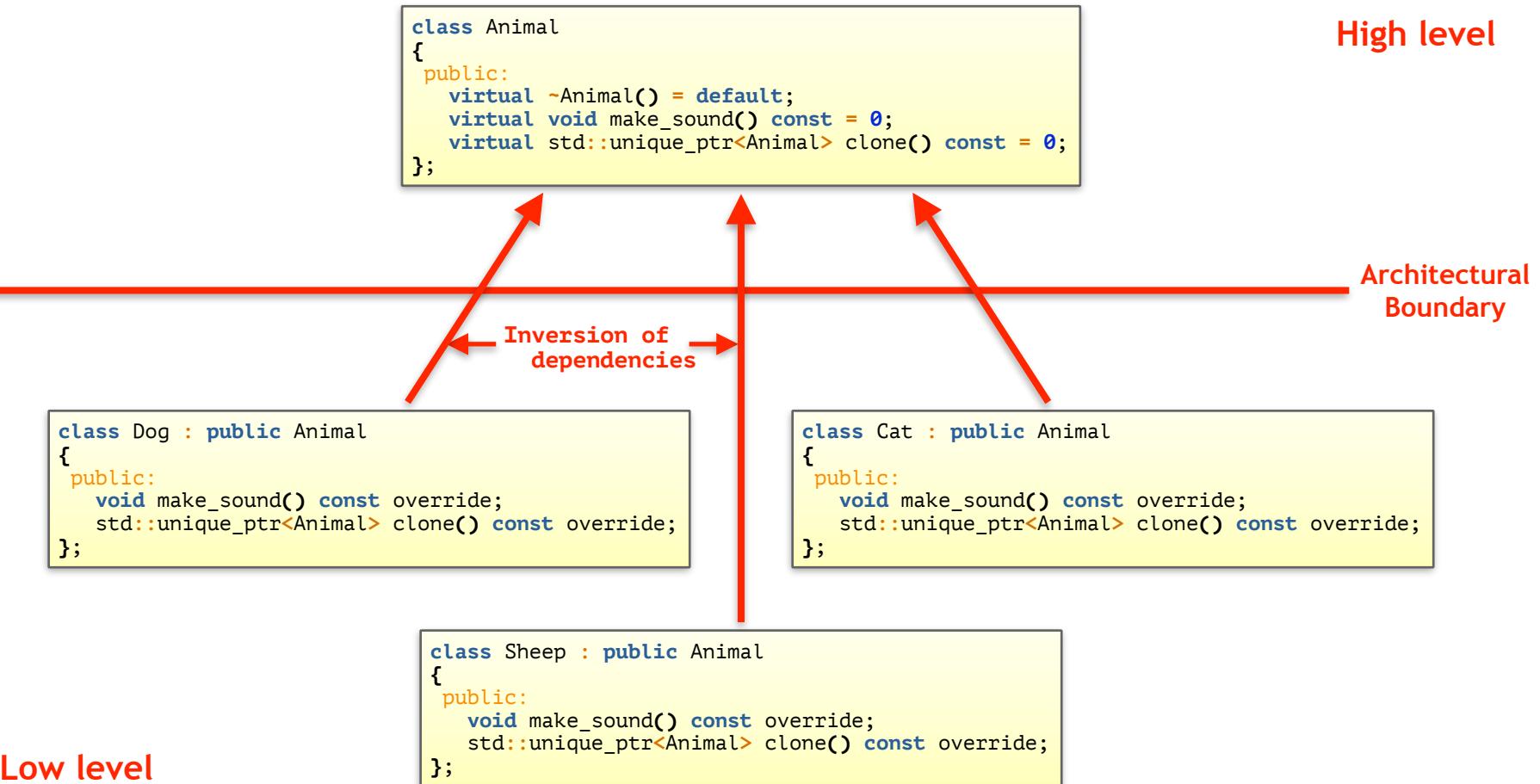
The Prototype Design Pattern



The Classic Command Pattern

Task (2_Cpp_Software_Design/Prototype/Prototype): Discuss the advantages and disadvantages of the given implementation of the classic Prototype design pattern.

Dependency Structure



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).

2.10. Bridge

Motivation

**Change
of implementation details**

Motivation



Electric Hero: The Ultimate Electric Cars



Electric Hero: The Ultimate Electric Cars

```
//---- <ElectricHero.> -----

#include <BatteryGen1.h>
#include <ElectricEngineGen1.h>
// ...

class ElectricHero
{
public:
    void drive();
    // ...
private:
    BatteryGen1 battery_;
    ElectricEngineGen1 engine_;

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

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



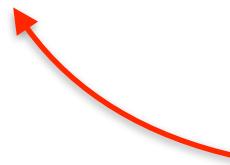
Electric Hero: The Ultimate Electric Cars

```
//---- <ElectricHero.h> -----
#include <memory>
// ...
struct BatteryGen1;           // Forward declaration
struct ElectricEngineGen1;    // Forward declaration

class ElectricHero
{
public:
    void drive();
    // ...
private:
    std::unique_ptr<BatteryGen1> battery_;
    std::unique_ptr<ElectricEngineGen1> engine_;
    // ...
};

//---- <ElectricHero.cpp> -----
#include <BatteryGen1.h>
#include <ElectricEngineGen1.h>

// ...
```



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

Electric Hero: The Ultimate Electric Cars

```
//---- <ElectricCarImpl> -----
class ElectricCarImpl
{
public:
    virtual ~ElectricCarImpl() = default;

    virtual void start() = 0;
    virtual void stop() = 0;

    virtual void drawPower() = 0;
    virtual void charge() = 0;

    // ... more car-specific functions

private:
    // ... potentially some car-specific data members (wheels, drivetrain, ...)
};

//---- <ElectricHero.h> -----
#include <ElectricCarImpl.h>
#include <memory>

class ElectricHero
{
public:
    void drive():
```



Electric Hero: The Ultimate Electric Cars

```
//---- <ElectricCarImpl> -----
class ElectricCarImpl
{
public:
    virtual ~ElectricCarImpl() = default;
    virtual void start() = 0;
    virtual void stop() = 0;
    virtual void drawPower() = 0;
    virtual void charge() = 0;
    // ... more car-specific functions
private:
    // ... potentially some car-specific data members (wheels, drivetrain, ...)
};

//---- <ElectricHero.h> -----
#include <ElectricCarImpl.h>
#include <memory>

class ElectricHero
{
public:
    void drive():
```



Electric Hero: The Ultimate Electric Cars

```
private:
    // ... potentially some car-specific data members (wheels, drivetrain
};
```

//---- <ElectricHero.h> -----

```
#include <ElectricCarImpl.h>
#include <memory>
```

```
class ElectricHero
{
public:
    void drive();
    // ...

private:
```

```
    std::unique_ptr<ElectricCarImpl> pimpl_;
    // ... potentially some non-coupling car-specific data members
};
```

//---- <ElectricHeroImpl.h> -----

```
#include <ElectricCarImpl.h>
#include <BatteryGen1.h>
#include <ElectricEngineGen1.h>
```



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



Electric Hero: The Ultimate Electric Cars

```
//---- <ElectricHeroImpl.h> -----
#include <ElectricCarImpl.h>
#include <BatteryGen1.h>
#include <ElectricEngineGen1.h>

class ElectricHeroImpl : public ElectricCarImpl
{
public:
    explicit ElectricHeroImpl( double charge )
        : battery_{charge}
        , engine_{}
    {}

    void start() override { engine_.start(); }
    void stop() override { engine_.stop(); }

    void drawPower() override { battery_.drawPower(); }
    void charge() override { battery_.charge(); }

    // ... more car or EH-specific functions

private:
    BatteryGen1 battery_;
    ElectricEngineGen1 engine_;
    // ... more EH-specific data members (wheels, drivetrain, ...)
};
```



Electric Hero: The Ultimate Electric Cars

```
void start() override { engine_.start(); }
void stop() override { engine_.stop(); }

void drawPower() override { battery_.drawPower(); }
void charge() override { battery_.charge(); }

// ... more car or EH-specific functions

private:
    BatteryGen1 battery_;
    ElectricEngineGen1 engine_;
    // ... more EH-specific data members (wheels, drivetrain, ...)
};

//---- <ElectricHero.cpp> ----

#include <ElectricHero.h>
#include <ElectricHeroImpl.h>

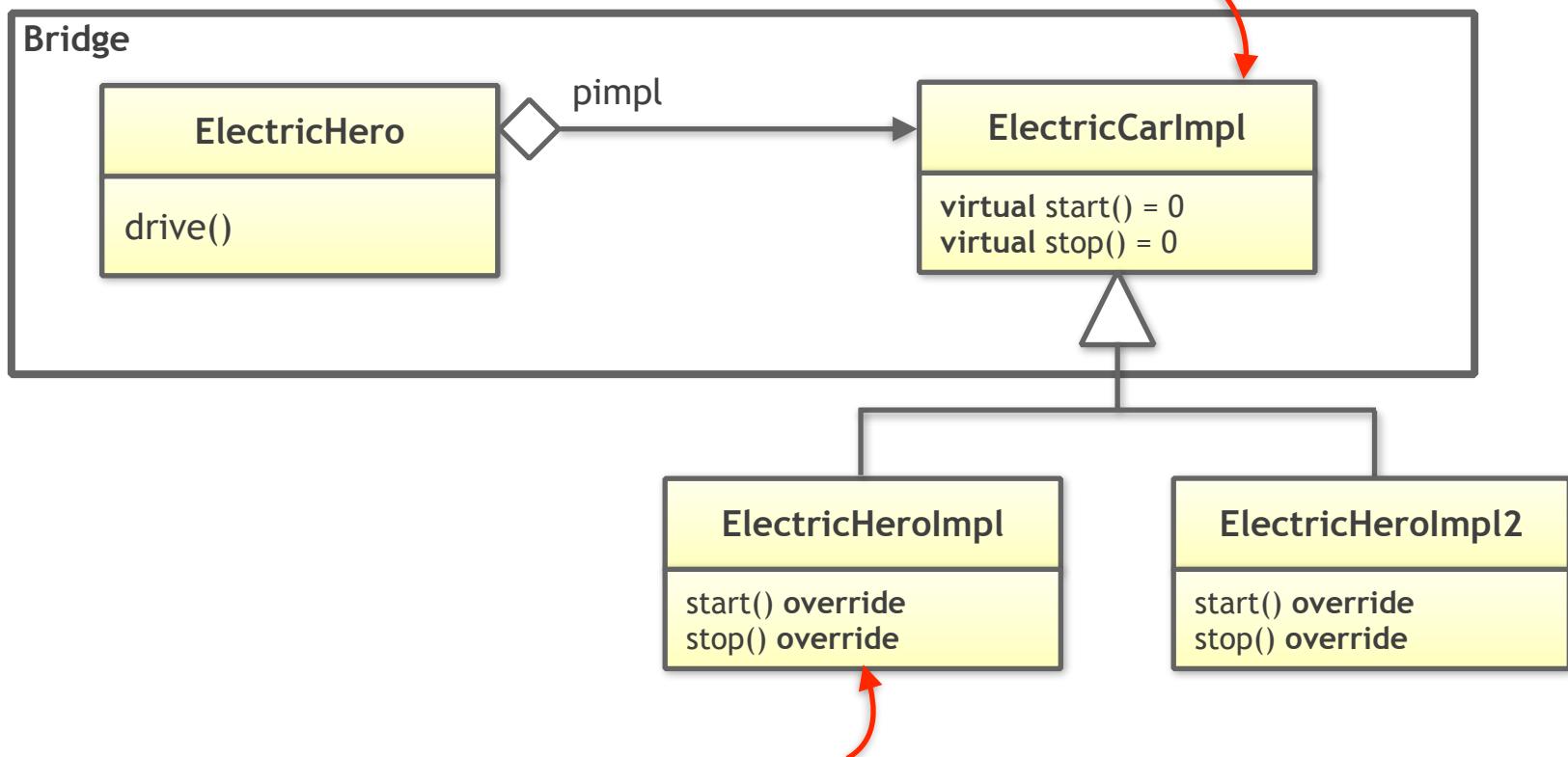
ElectricHero::ElectricHero( /*maybe some battery and/or engine arguments*/ )
    : pimpl_{ std::make_unique<ElectricHeroImpl>( 100.0 ) }
{}

// ... Other 'ElectricHero' member functions, primarily using the battery
//      and engine.
```



The Classic Bridge Design Pattern

The implementation details of electric hero cars is the variation point. This aspect changes and is extracted and isolated; this fulfills the Single-Responsibility Principle (SRP)



Adding new implementations is the customization point: new implementations can be added without modifying any existing code; this fulfills the Open-Closed Principle (OCP)

Motivation

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

class ElectricCar
{
protected:
    explicit ElectricCar( std::unique_ptr<ElectricCarImpl> impl )
        : pimpl_{ std::move(impl) }
    {}

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

protected:
    ElectricCarImpl* getImpl() { return pimpl_.get(); }

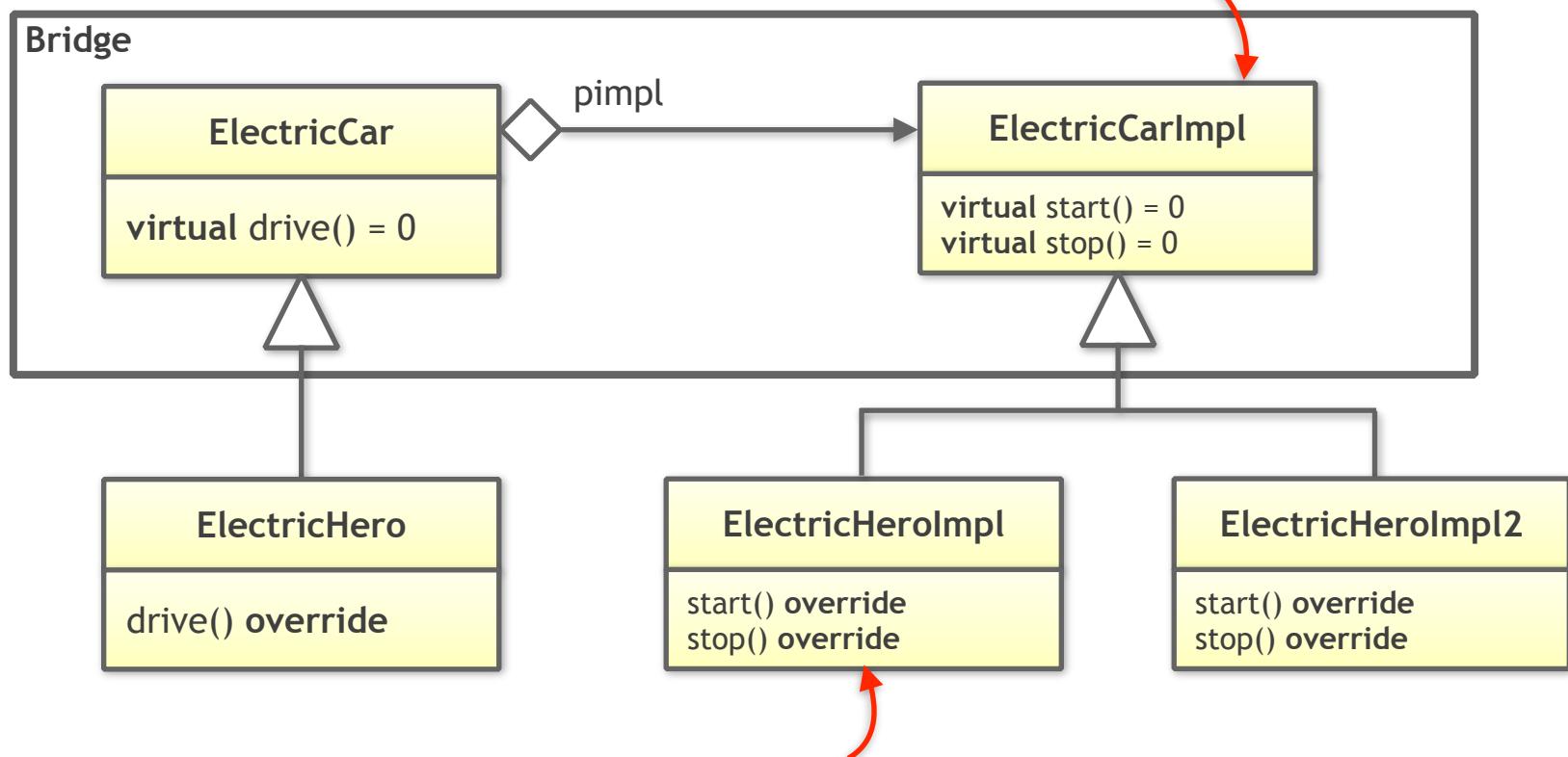
private:
    std::unique_ptr<ElectricCarImpl> pimpl_; // Pointer-to-implementation
};
```



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

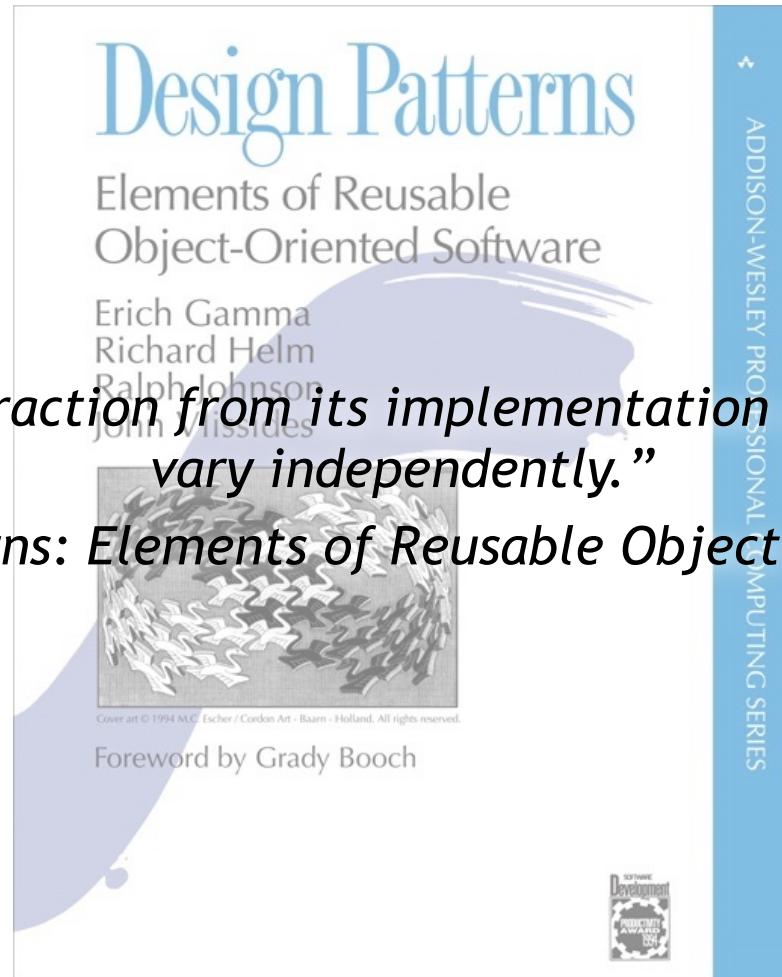
The Classic Bridge Design Pattern

The implementation details of electric hero cars is the variation point. This aspect changes and is extracted and isolated; this fulfills the Single-Responsibility Principle (SRP)



Adding new implementations is the customization point: new implementations can be added without modifying any existing code; this fulfills the Open-Closed Principle (OCP)

The Intent



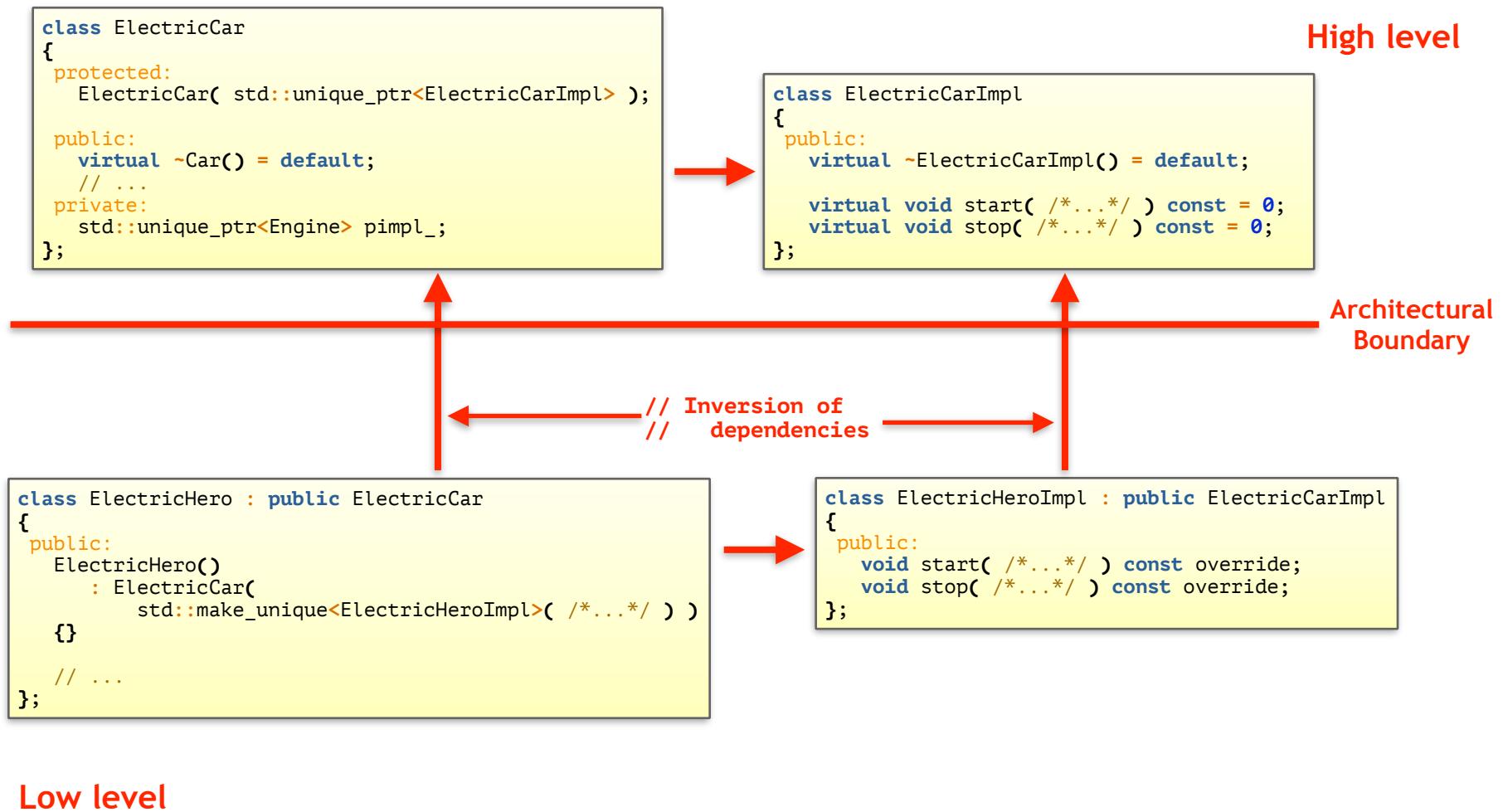
"Decouple an abstraction from its implementation so that the two can vary independently."

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

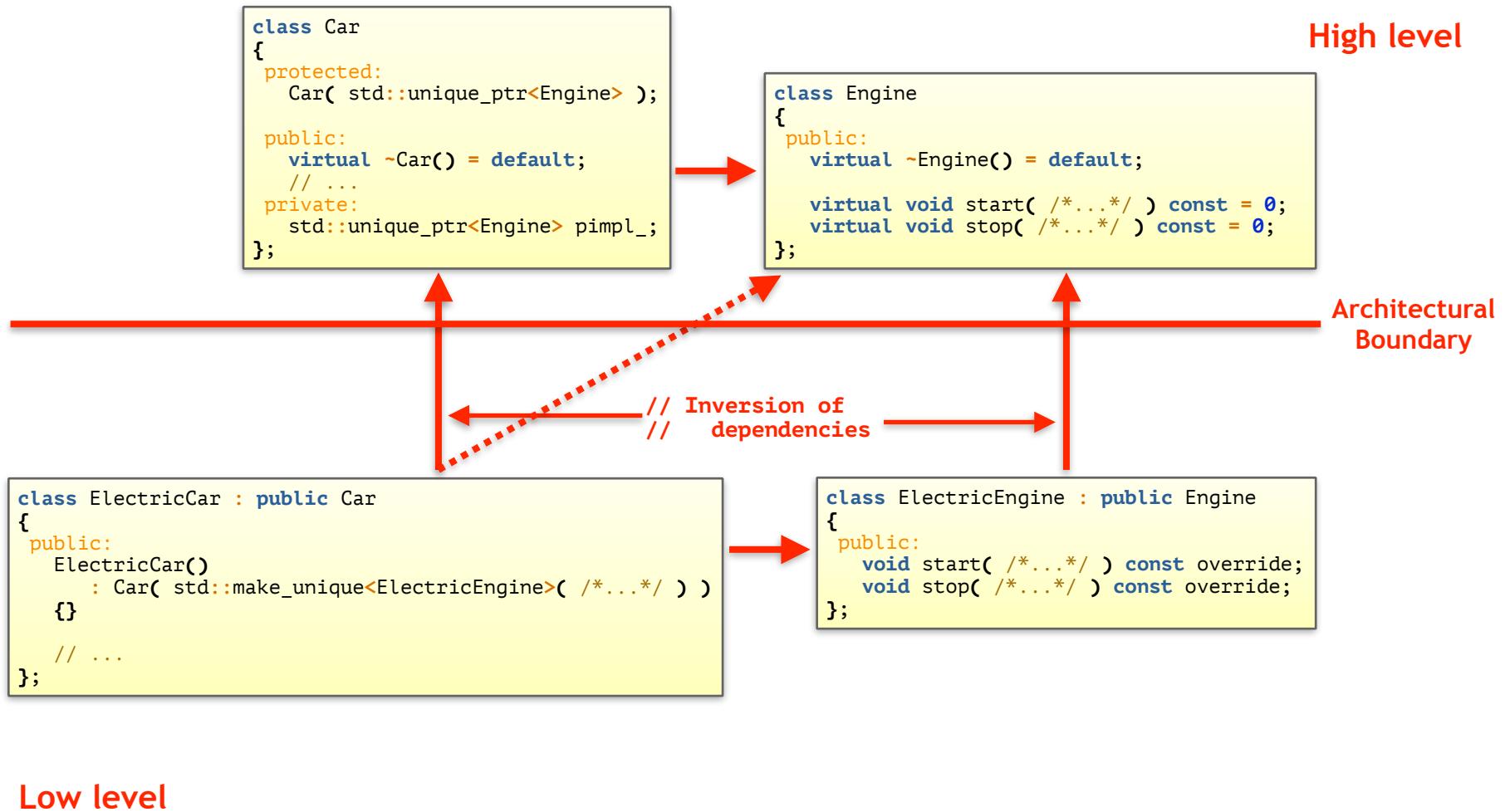
Programming Task

Task (2_Cpp_Software_Design/Bridge/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?

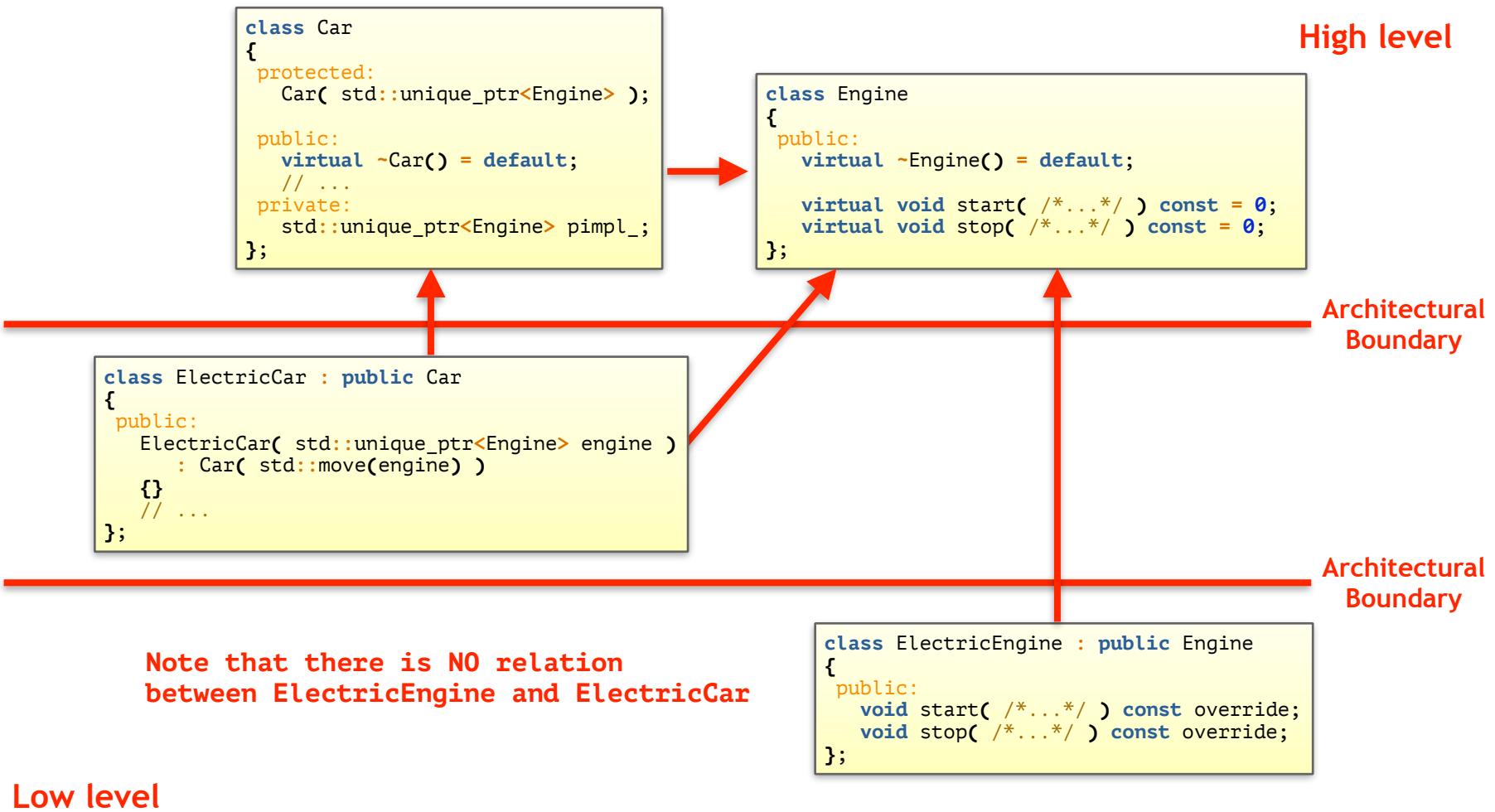
Dependency Structure



Dependency Structure



Dependency Structure of Strategy

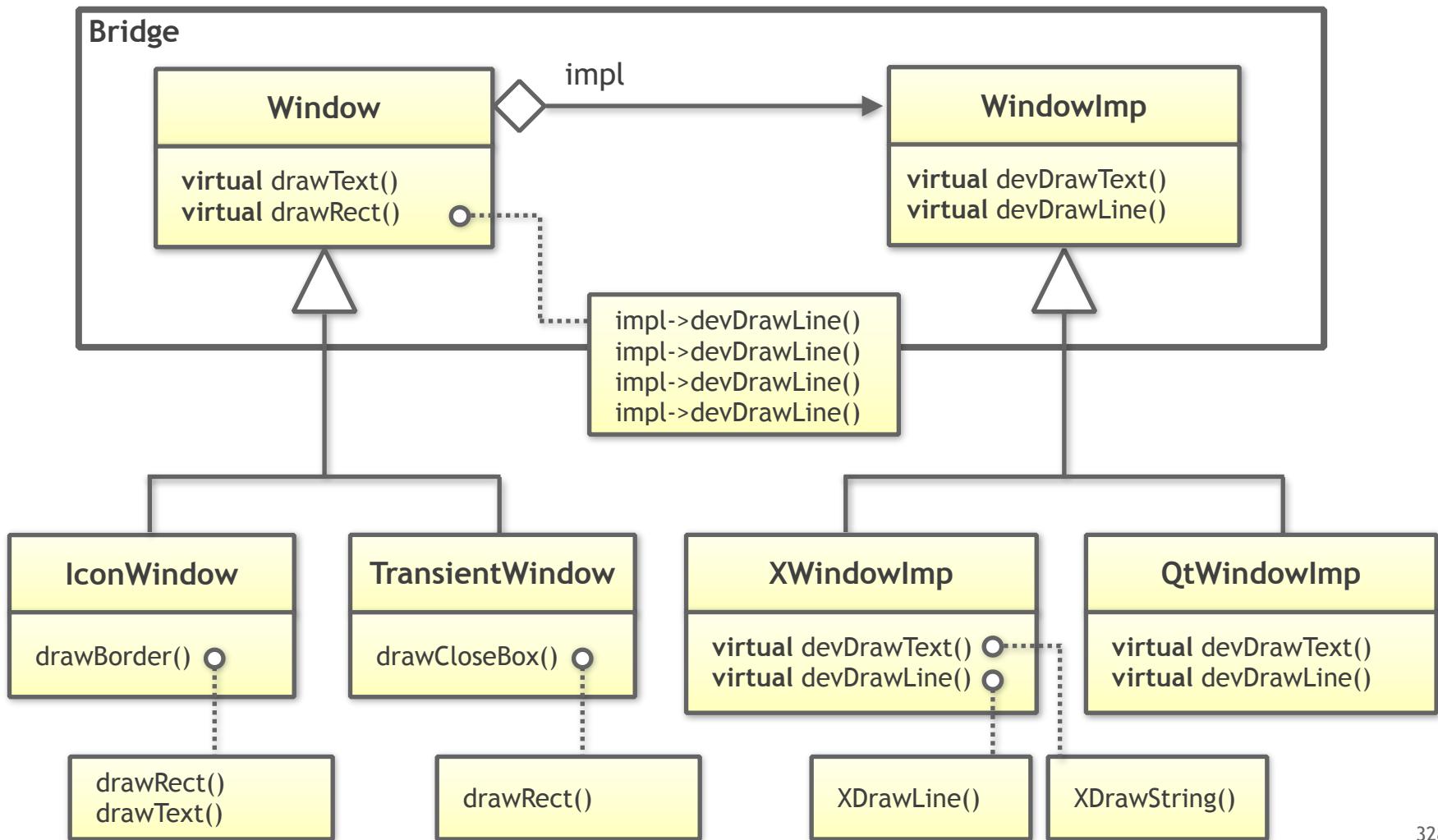


Low level

Bridge vs. Strategy – Telltale Signs

<i>Bridge</i>	<i>Strategy</i>
Reduces physical dependencies	Reduces logical dependencies
No dependency injection	Always dependency injection
Internal customization point	External customization point
Does not improve testability	Improves testability

No Dependency Injection – Really?



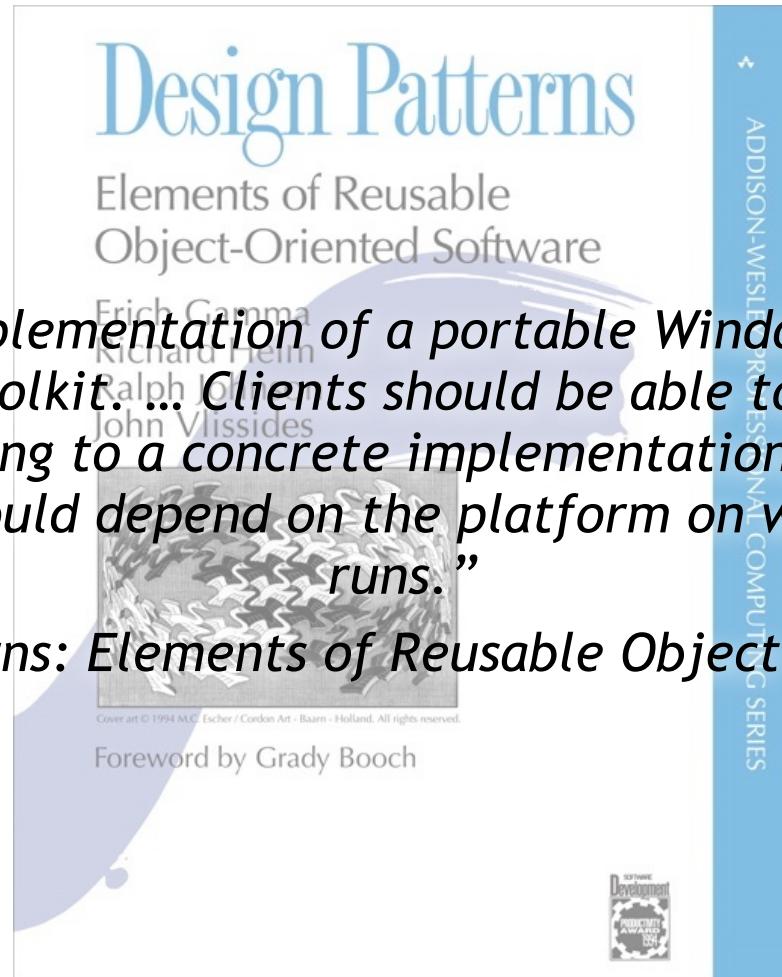
No Dependency Injection – Really?

```
class WindowImp {  
public:  
    virtual void ImpTop() = 0;  
    virtual void ImpBottom() = 0;  
    virtual void SetExtent(const Point&) = 0;  
    virtual void SetOrigin(const Point&) = 0;  
  
    // ...  
    // lots more functions for drawer on windows  
};  
  
class Window {  
public:  
    Window(View* contents);  
  
    virtual void DrawContents();  
    // ...  
  
private:  
    View* _contents; // the Window's contents  
    WindowImp* _imp;  
};
```

This is a Strategy (dependency injection, external customization point), ...

... this is a Bridge (no dependency injection, internal customization point).

No Dependency Injection – Really?



"Consider the implementation of a portable Window abstraction in a user interface toolkit. ... Clients should be able to create a window without committing to a concrete implementation. Only the window implementation should depend on the platform on which the application runs."

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

Programming Task

Task (2_Cpp_Software_Design/Bridge/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 (2_Cpp_Software_Design/Bridge/BridgedMembers): Copy-and-paste the following code into quick-bench.com. Benchmark the time to compute the total age of all persons contained in a std::vector.

Static Memory Instead of Dynamic Memory

```
//---- <X.h> -----
#include <type_traits>
// ...

class X : public A
{
public:
    X( C const& c );
    // ...

private:
    struct Impl;

    Impl* pimpl() { return reinterpret_cast<Impl*>(&buffer); }

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

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

//---- <X.cpp> -----
#include <X.h>
```

Static Memory Instead of Dynamic Memory

```
//---- <X.h> -----
#include <type_traits>
// ...

class X : public A
{
public:
    X( C const& c );
    // ...

private:
    struct Impl;

    Impl* pimpl() { return reinterpret_cast<Impl*>(&buffer); }

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

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

//---- <X.cpp> -----
#include <X.h>
```

Static Memory Instead of Dynamic Memory

```
//---- <X.h> -----
#include <type_traits>
// ...

class X : public A
{
public:
    X( C const& c );
    // ...

private:
    struct Impl;

    Impl* pimpl() { return reinterpret_cast<Impl*>(&buffer); }

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

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

//---- <X.cpp> -----
#include <X.h>
```

Static Memory Instead of Dynamic Memory

```
//---- <X.h> -----
#include <type_traits>
// ...

class X : public A
{
public:
    X( C const& c );
    // ...

private:
    struct Impl;

    Impl* pimpl() { return reinterpret_cast<Impl*>(&buffer); }

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

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

//---- <X.cpp> -----
#include <X.h>
```

Static Memory Instead of Dynamic Memory

```
static constexpr size_t bufferalign = 16;

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

};

//---- <X.cpp> ----

#include <X.h>
#include <memory>
// ...

struct Impl { /*...*/ };

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

    std::construct_at( pimpl(), c );
    //::new (&buffer_) Impl{ c };
}

// ...

```

Static Memory Instead of Dynamic Memory

```
//---- <X.h> -----
#include <type_traits>
// ...

class X : public A
{
public:
    X( C const& c );
    // ...

private:
    struct Impl;

    Impl* pimpl() { return reinterpret_cast<Impl*>(&buffer); }

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

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

//---- <X.cpp> -----
#include <X.h>
```

Note that these two values can never change without breaking the ABI of class X!

Programming Task

Task (2_Cpp_Software_Design/Bridge/FastPimpl): Refactor the given ElectricCar class by means of the "Fast Pimpl Idiom":

- Evaluate the required size and alignment of the 'Impl' class
- Replace the ElectricEngineGen1 and BatteryGen1 data members with an Impl class
- Refactor the special member functions of the ElectricCar class

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 Bridge design pattern to make client code independent of platform specific code (no dependency injection).

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

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 (<https://www.youtube.com/watch?v=QthAU-t3PQ4>)
- Jon Kalb, “Back to Basics: Object-Oriented Programming”. CppCon 2019 (<https://www.youtube.com/watch?v=32tDTD9UJCE>)
- 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>)

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>)
- Klaus Iglberger, “Breaking Dependencies: Type Erasure - The Implementation Details”. CppCon 2022 (<https://www.youtube.com/watch?v=qn6OqefuH08>)
- 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)
- Eduardo Madrid, “Not Leaving Performance On The Jump Table”, CppCon 2020 (https://www.youtube.com/watch?v=e8SyxB3_mnw)
- Scott Wlaschin, “Function Programming Design Patterns”. NDC London 2014 (<https://www.youtube.com/watch?v=E8I19uA-wGY>)

email: klaus.iglberger@gmx.de

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

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