

C++ Training

2. Modern C++ Design Patterns

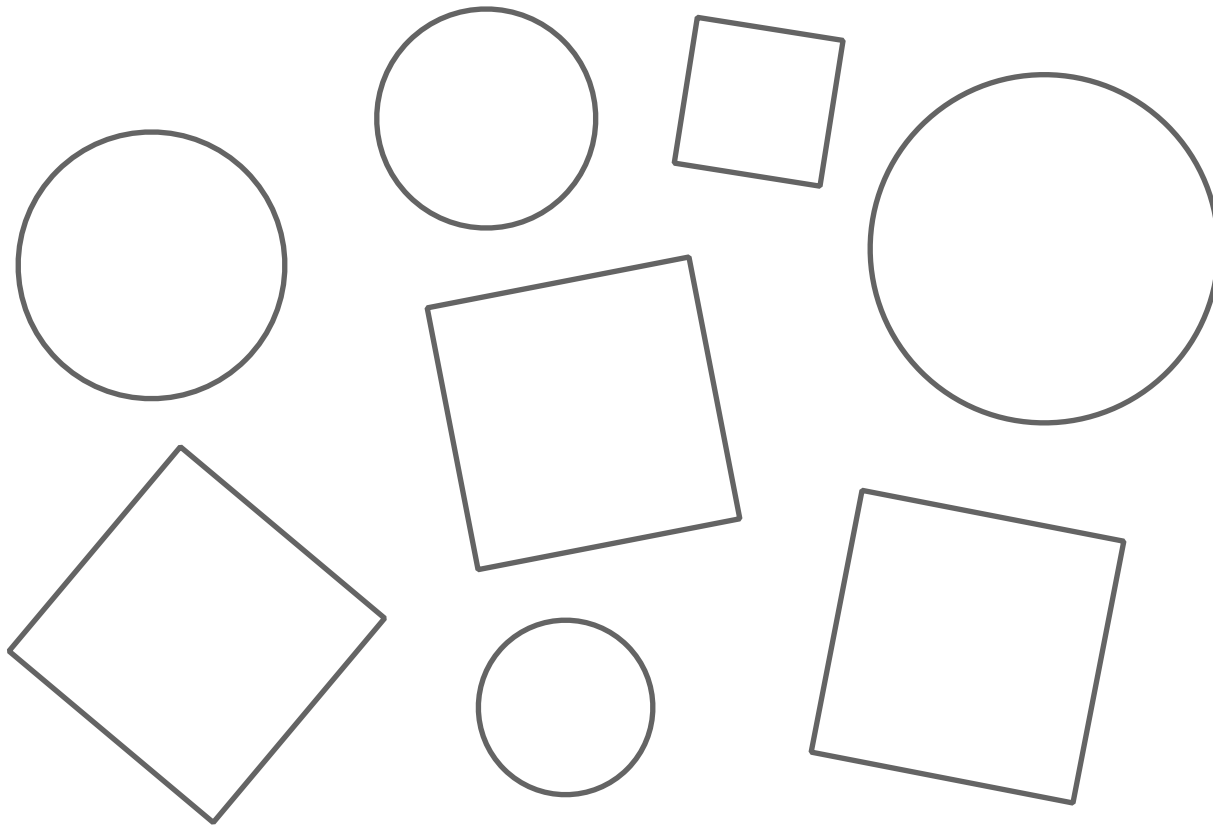
Klaus Iglberger
13.7.2020

Content

1. Motivation
2. (De-)Motivation
3. Visitor
4. State Machines
5. Command
6. Strategy
7. Observer
8. Prototype
9. Type Erasure
10. Decorator
11. Template Method
12. Singleton
13. CRTP
14. Expression Templates

2.1. Motivation

Our Toy Problem: Drawing Shapes



An Example

Task (2_Modern_Cpp_Design_Patterns/Procedural): Evaluate the given design with respect to changeability and extensibility.

A Procedural Solution

```
enum ShapeType
{
    circle,
    square
};

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

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

private:
    ShapeType type;
};

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

    virtual ~Circle() = default;
```

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
    {}

    virtual ~Circle() = default;
```

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
    {}

    virtual ~Circle() = default;
```


A Procedural Solution

```
};

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

    virtual ~Circle() = default;
    double getRadius() const noexcept;
    // ... getCenter(), getRotation(), ...

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

void translate( Circle&, Vector3D const& );
void rotate( Circle&, Quaternion 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
    {}

    virtual ~Square() = default;
    double getSide() const noexcept;
    // ... getCenter(), getRotation(), ...

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

void translate( Square&, Vector3D const& );
void rotate( Square&, Quaternion const& );
void draw( Square const& );

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

A Procedural Solution

```
};

void translate( Square&, Vector3D const& );
void rotate( Square&, Quaternion const& );
void draw( Square const& );

void draw( 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.push_back( std::make_unique<Circle>( 2.0 ) );
    shapes.push_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.push_back( std::make_unique<Circle>( 2.0 ) );
    shapes.push_back( std::make_unique<Square>( 1.5 ) );
    shapes.push_back( std::make_unique<Circle>( 4.2 ) );

    // Drawing all shapes
    draw( 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
        {}

    virtual ~Circle() = default;
    double getRadius() const noexcept;
    // ... getCenter(), getRotation(), ...

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

void translate( Circle&, Vector3D const& );
void rotate( Circle&, Quaternion 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
    {}

    virtual ~Square() = default;
    double getSide() const noexcept;
    // ... getCenter(), getRotation(), ...

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

void translate( Square&, Vector3D const& );
void rotate( Square&, Quaternion const& );
void draw( Square const& );

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

A Procedural Solution

```
void draw( Square const& );

void draw( 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.push_back( std::make_unique<Circle>( 2.0 ) );
    shapes.push_back( std::make_unique<Square>( 1.5 ) );
    shapes.push_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++)

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

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

Dependencies

The Problem

Dependencies ...

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

The Expert's Opinion

*"Dependency is the key problem in software development at all scales."
(Kent Beck, TDD by Example)*

Guidelines

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

The SOLID Principles

Single-Responsibility Principle (SRP)

Open-Closed Principle (OCP)

Liskov Substitution Principle (LSP)

Interface Segregation Principle (ISP)

Dependency Inversion Principle (DIP)

The SOLID Principles

Single-Responsibility Principle (SRP)

Open-Closed Principle (OCP)

Liskov Substitution Principle (LSP)

Interface Segregation Principle (ISP)

Dependency Inversion Principle (DIP)



Robert C. Martin



Michael Feathers

An Example

Task (2_Modern_Cpp_Design_Patterns/ObjectOriented): Evaluate the given design with respect to changeability and extensibility.

An Example

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

    virtual void translate( Vector3D const& ) = 0;
    virtual void rotate( Quaternion const& ) = 0;
    virtual void draw() const = 0;
};

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

    virtual ~Circle() = default;

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

    void translate( Vector3D const& ) override;
    void rotate( Quaternion const& ) override;
    void draw() const override;

private:
    double radius;
```

An Example

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

    virtual void translate( Vector3D const& ) = 0;
    virtual void rotate( Quaternion const& ) = 0;
    virtual void draw() const = 0;
};
```

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

    virtual ~Circle() = default;

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

    void translate( Vector3D const& ) override;
    void rotate( Quaternion 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
            {}

        virtual ~Circle() = default;

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

        void translate( Vector3D const& ) override;
        void rotate( Quaternion const& ) override;
        void draw() const override;

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

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

An Example

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

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

    virtual ~Square() = default;

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

    void translate( Vector3D const& ) override;
    void rotate( Quaternion const& ) override;
    void draw() const override;

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

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

An Example

```
void translate( vector3D const& ) override;
void rotate( Quaternion const& ) override;
void draw() const override;

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

void draw( 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.push_back( std::make_unique<Circle>( 2.0 ) );
    shapes.push_back( std::make_unique<Square>( 1.5 ) );
    shapes.push_back( std::make_unique<Circle>( 4.2 ) );

    // Drawing all shapes
    draw( 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.push_back( std::make_unique<Circle>( 2.0 ) );
    shapes.push_back( std::make_unique<Square>( 1.5 ) );
    shapes.push_back( std::make_unique<Circle>( 4.2 ) );

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


An Example

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

    virtual void translate( Vector3D const& ) = 0;
    virtual void rotate( Quaternion const& ) = 0;
    virtual void draw() const = 0;
};
```

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

    virtual ~Circle() = default;

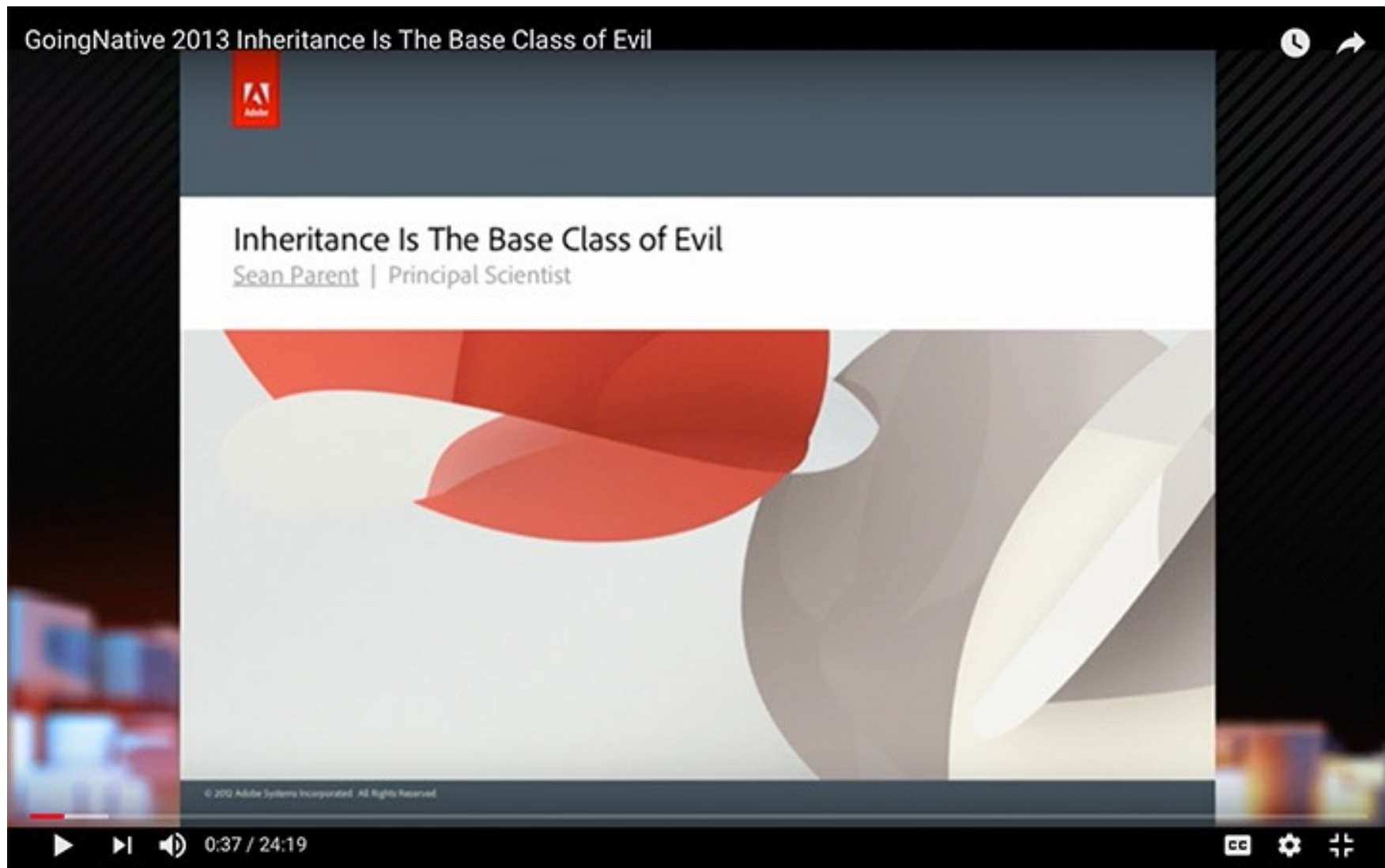
    double getRadius() const noexcept;
    // ... getCenter(), getRotation(), ...

    void translate( Vector3D const& ) override;
    void rotate( Quaternion const& ) override;
    void draw() const override;
```

```
private:
    double radius;
```

2.2. (De-)Motivation

The Expert's Attitude



The Expert's Attitude

CppCon 2019: Jon Kalb "Back to Basics: Object-Oriented Programming"

Jon Kalb

Back to Basics:
Object-Oriented
Programming

Video Sponsorship Provided By:
ansatz


THE NEW STACK Ebooks Podcasts Events Newsletter

Architecture Development Operations

CULTURE / DEVELOPMENT

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

21 Aug 2019 12:00pm, by David Cassel

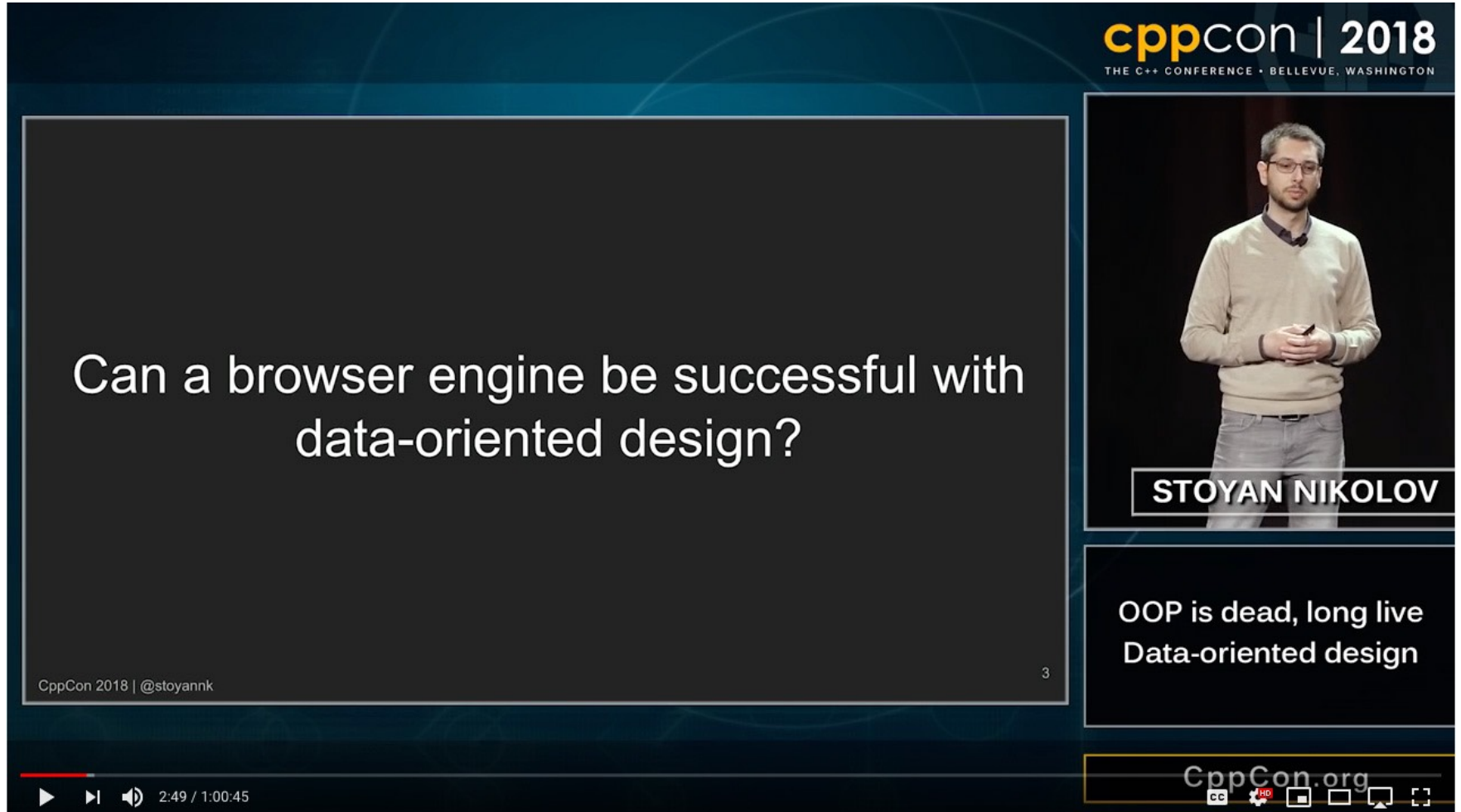


2

1:24 / 59:58

CC BY ND

The Expert's Attitude



The screenshot shows a video player interface for a CppCon 2018 presentation. The main slide is dark blue with white text asking "Can a browser engine be successful with data-oriented design?". To the right, a smaller video window shows the speaker, Stoyan Nikolov, a man with glasses and a beard wearing a light-colored sweater. Below the speaker's name, a quote reads "OOP is dead, long live Data-oriented design". The video player controls at the bottom show a progress bar at 2:49 / 1:00:45. The CppCon.org logo and social media icons are in the bottom right corner.

cppcon | 2018
THE C++ CONFERENCE • BELLEVUE, WASHINGTON

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

STOYAN NIKOLOV

OOP is dead, long live Data-oriented design

CppCon 2018 | @stoyannk 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)
- The Forgotten History of OOP (Eric Elliott)

Why is it so bad?

- Does not harmonize with the philosophy of the STL
- Inheritance creates a very tight coupling (second only to friendship)
 - Intrusive and/or very verbose to add something to the hierarchy
 - Adding functions requires modifications (violation of the OCP)
 - May cause contradictions between SRP and OCP
- Inheritance introduces overhead:
 - Heap allocation (memory management)
 - Virtual functions (no inlining)
 - Null pointers

Dynamic Allocation

```
class Animal {};
```

```
class Cat : public Animal {};
```

```
class Dog : public Animal {};
```

```
Animal make_animal();
```

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

Dynamic Allocation

```
class Animal {};
```

```
class Cat : public Animal {};
```

```
class Dog : public Animal {};
```

```
Animal make_animal();
```

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

Dynamic Allocation

```
class Animal {};
```

```
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
- Inheritance creates a very tight coupling (second only to friendship)
 - Intrusive and/or very verbose to add something to the hierarchy
 - Adding functions requires modifications (violation of the OCP)
 - May cause contradictions between SRP and OCP
- Inheritance introduces overhead:
 - Heap allocation (memory management)
 - Virtual functions (no inlining)
 - Null pointers
- Only works on a single type

```
Reaction react( Animal const& a1, Animal const& a2 );
```

2. Modern C++ Design Patterns - (De-)Motivation



Why is it so bad?

- Does not harmonize with the philosophy of the STL
- Inheritance creates a very tight coupling (second only to friendship)
 - Intrusive and/or very verbose to add something to the hierarchy
 - Adding functions requires modifications (violation of the OCP)
 - May cause contradictions between SRP and OCP
- Inheritance introduces overhead:
 - Heap allocation (memory management)
 - Virtual functions (no inlining)
 - Null pointers
- Only works on a single type

```
Reaction react( Animal const& a1, Animal const& a2 );
```

- Only works with pointers and references ...

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 GC required)
- Exception-safe
- Does not change
- No side effects
- Can use the same value in multiple threads (lock-free)
- Deterministic
- Pure
- Regular

→ Value Semantics

Pointers and References

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

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

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

Pointers and References

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

→ Reference Semantics

Custom Types

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

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

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

Custom Types

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

2. Modern C++ Design Patterns - (De-)Motivation



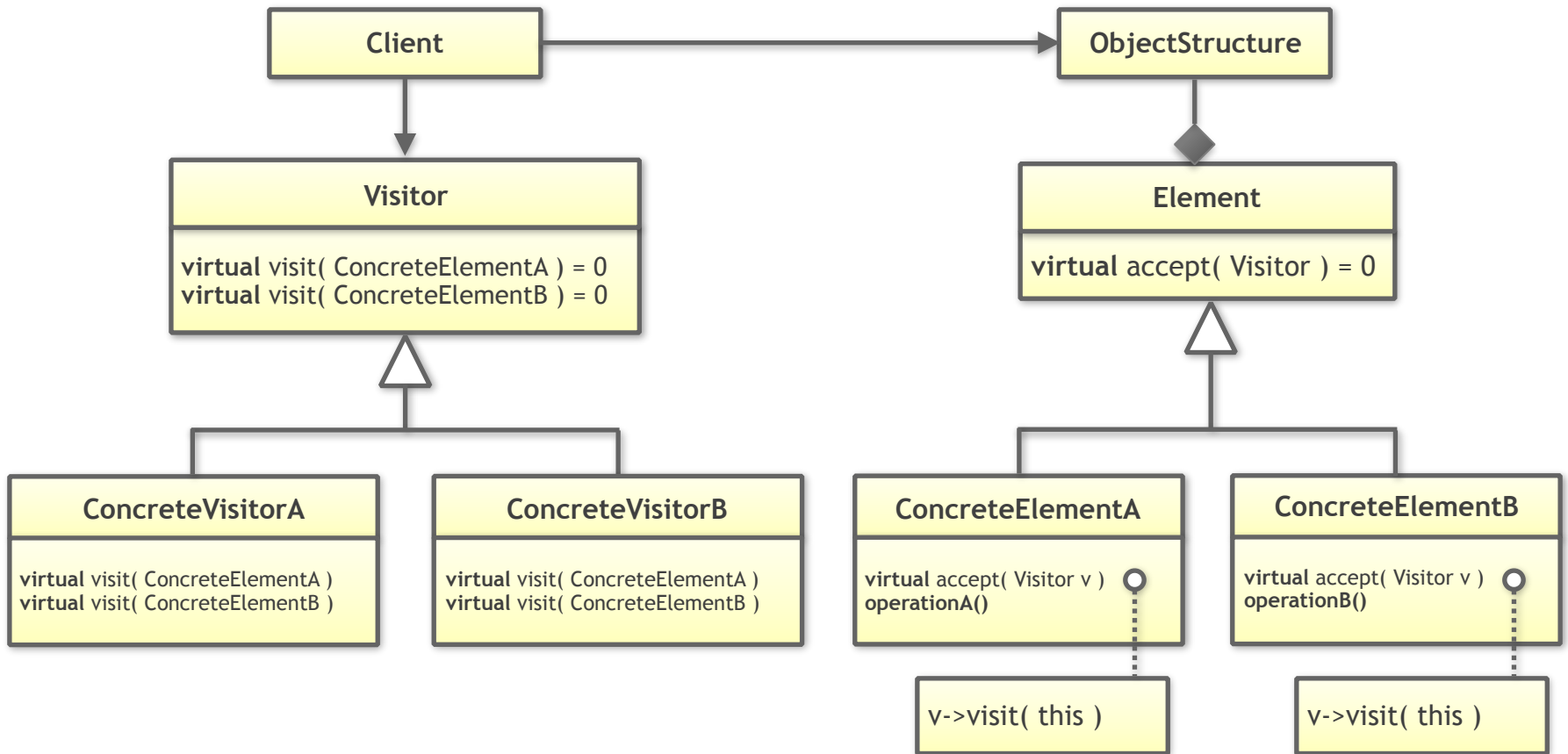
Guidelines

Guideline: Prefer value semantics over reference semantics!

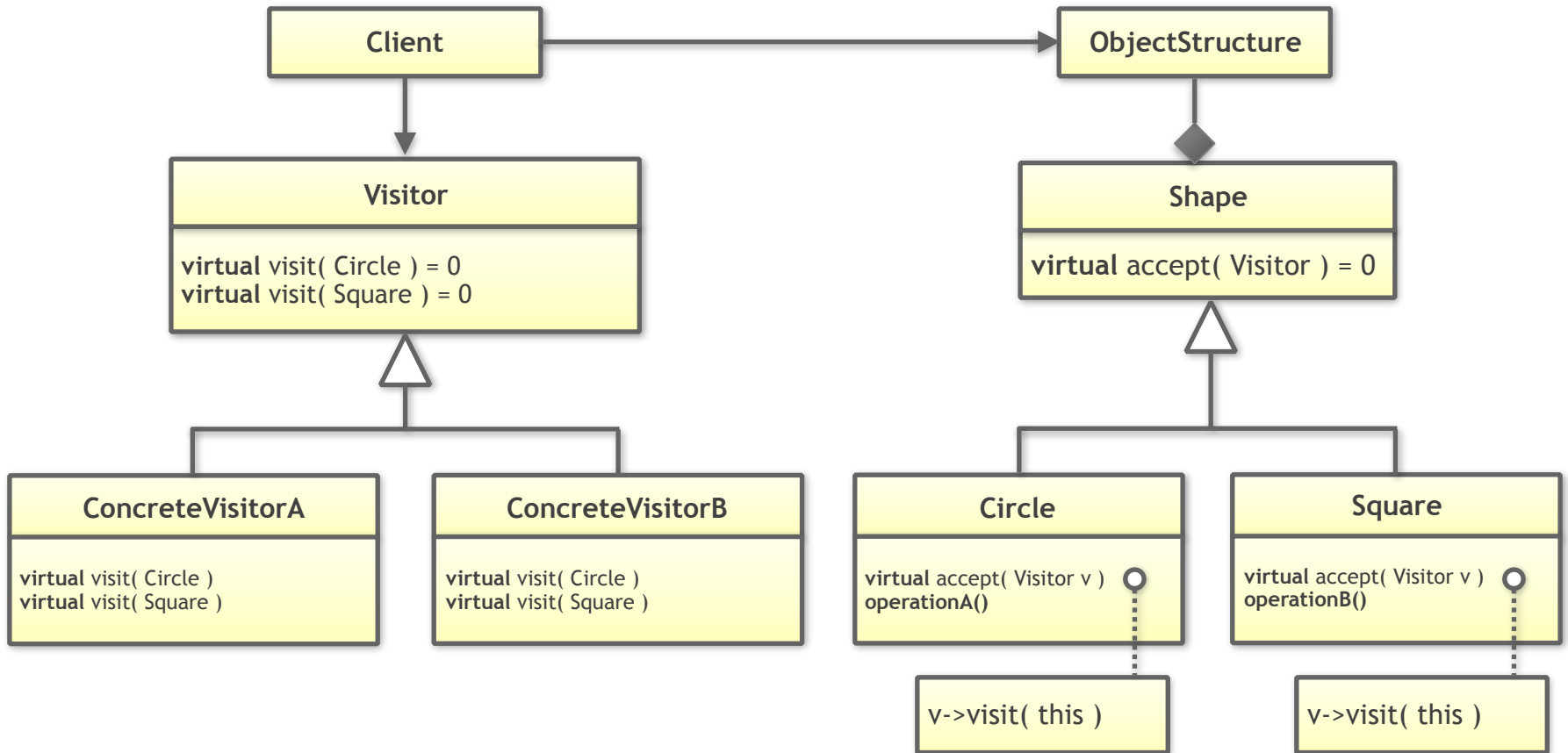
Guideline: Try to reduce the use of pointers!

2.3. Visitor

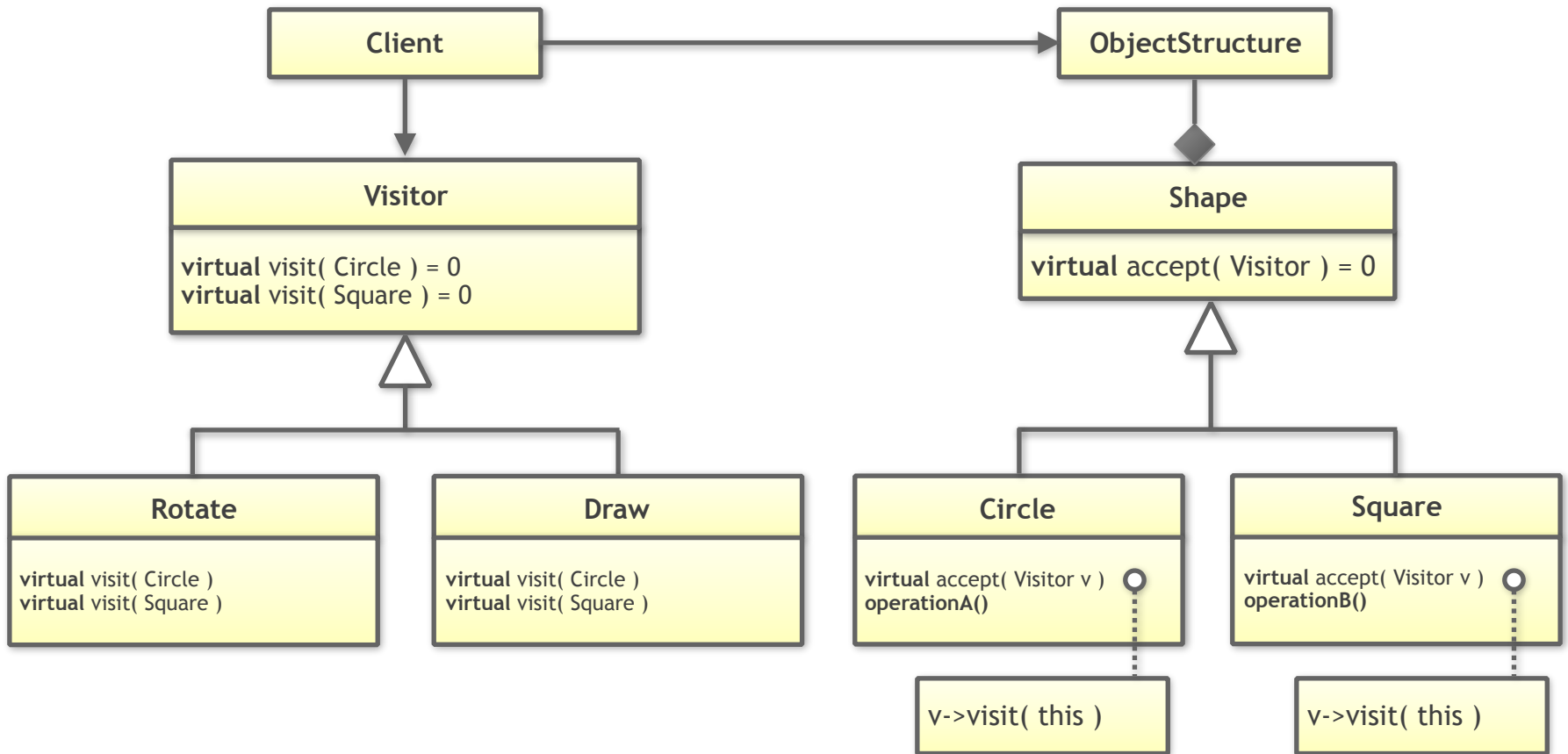
The Classical Visitor Design Pattern



The Classical Visitor Design Pattern



The Classical Visitor Design Pattern



The Classical Visitor Design Pattern

Task (2_Modern_Cpp_Design_Patterns/Visitor): Refactor the classical Visitor solution by a value semantics based solution. Note that the general behavior should remain unchanged.

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);
}
```

The Classical Visitor Design Pattern

The classical Visitor design pattern ...

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

Using `std::variant` instead of the classical 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**.

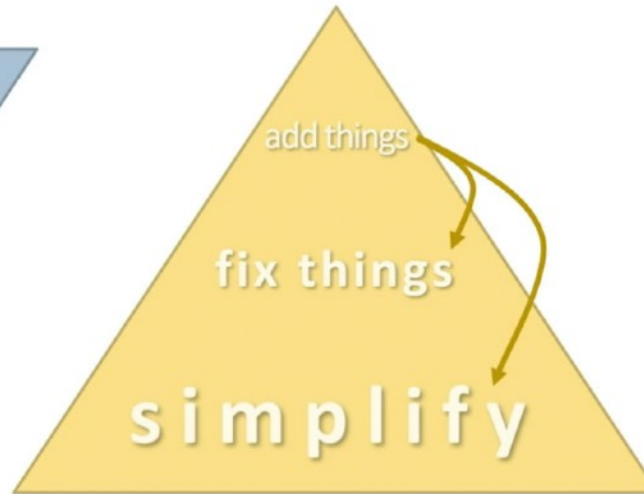
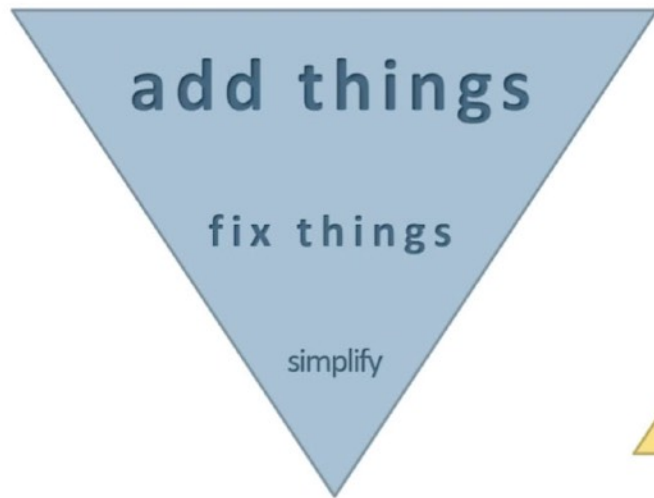
Don't use `std::variant` if ...

- ... you have an **open set** of types;
- ... you want to **extend types**, not functionality;
- ... you want to use the abstraction across **architectural boundaries**;
- ... performance is not the **primary concern**.

2. Modern C++ Design Patterns - Visitor

C++'s evolution priorities

Historical



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

Guidelines

Guideline: Avoid the over-/abuse of inheritance

Guideline: Prefer multi-paradigm solutions.

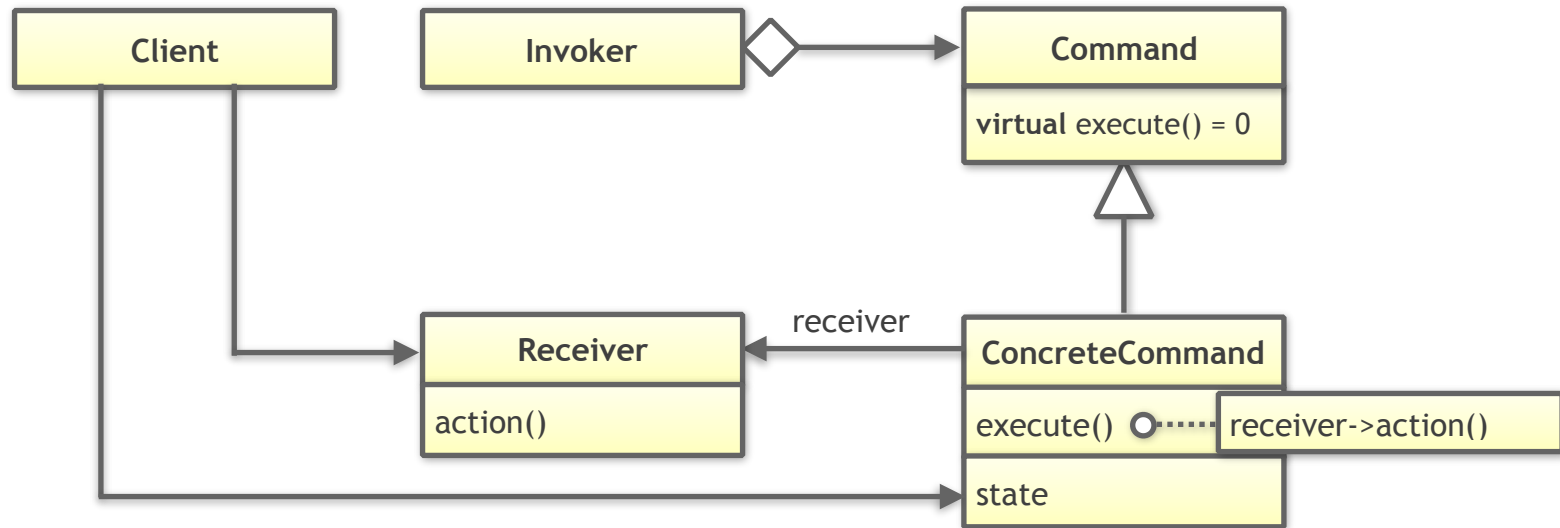
2.4. State Machines

Modern State Machine Implementations

Task (2_Modern_Cpp_Design_Patterns/StateMachine): Modify the given state machine implementation such that it is possible to reset a given state machine to the Start state.

2.5. Command

The Classical Command Pattern



The Classical Command Pattern

Question: Can we replace the Command Pattern by a simple function pointer?

```
using FP = void (*)( int, double );
```

Answer: Yes (in most cases). And this might have some advantages:

- Lower coupling (no inheritance)
- No manual memory management (no `new`, no `delete`)

The Classical Command Pattern

In “modern” C++ we would prefer to use `std::function` instead:

```
#include <functional>

using FP = std::function<void(int,double)>;
```

`std::function` is the generalization of a callable. It ...

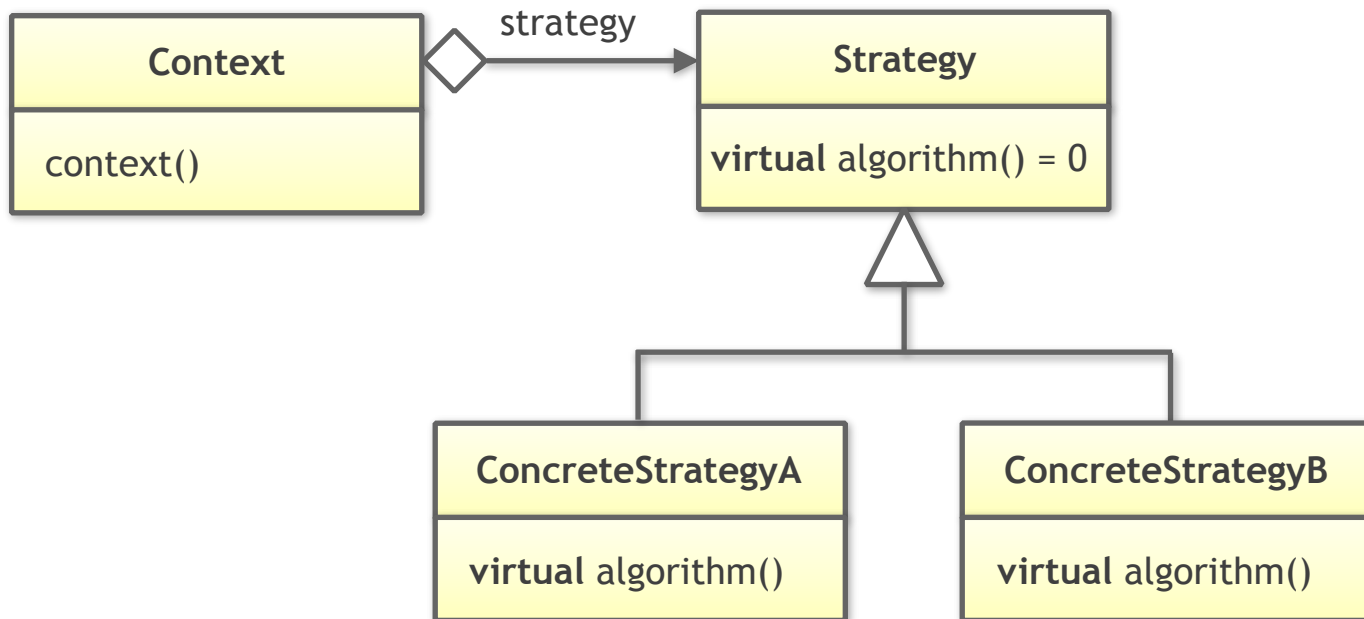
- ... takes (member) function pointers, functors, or lambdas;
- ... can have state (via functor or lambda);
- ... might allocate internally;
- ... is based on type erasure.

The Classical Command Pattern

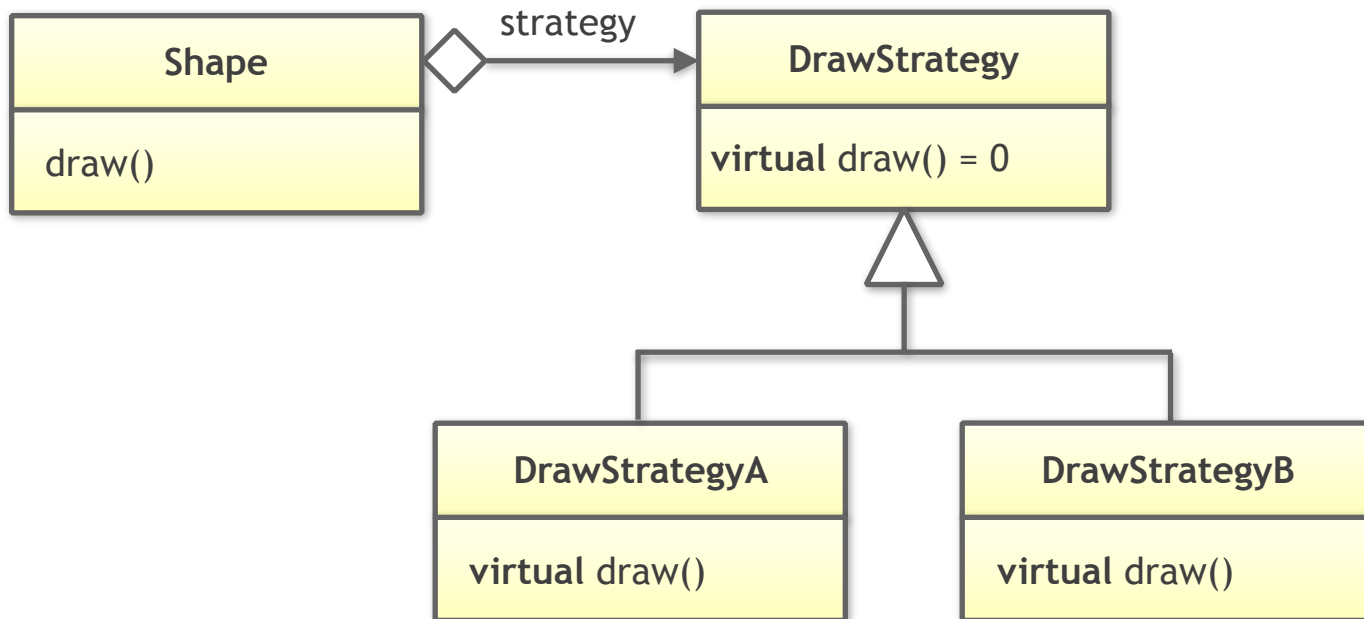
Task (2_Modern_Cpp_Design_Patterns/Command): Refactor the classical Command solution by means of `std::function`.

2.6. Strategy

The Classical Strategy Design Pattern



The Classical Strategy Design Pattern



The Classical Strategy Design Pattern

Task (2_Modern_Cpp_Design_Patterns/Strategy): Refactor the classical Strategy solution by a value semantics based solution. Note that the general behavior should remain unchanged.

The Classical Strategy Design Pattern

The classical Strategy design pattern ...

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

Using `std::function` instead of the classical Strategy design pattern ...

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

std::function — Advantages/Disadvantages

Use `std::function` if ...

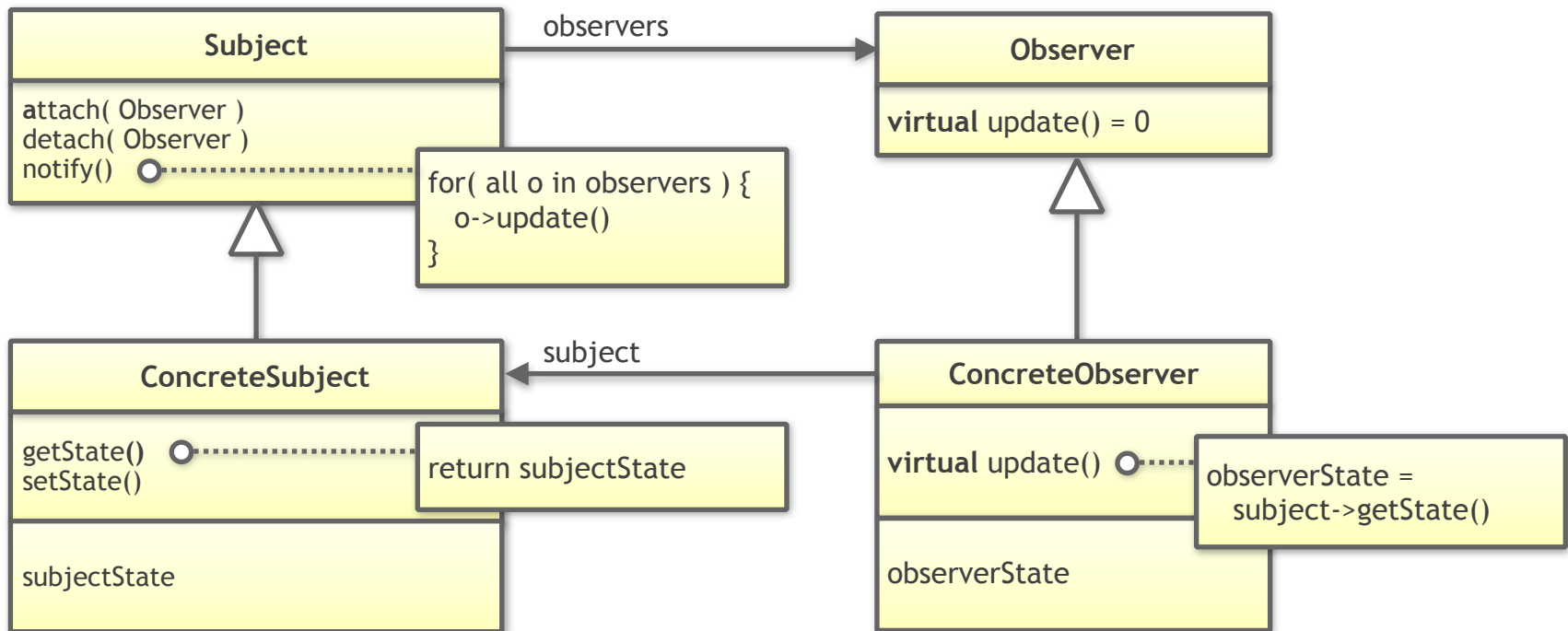
- ... you want to abstract and to **decouple a single function**;
- ... performance is not the **primary concern**.

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

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

2.7. Observer

The Classical Observer Pattern

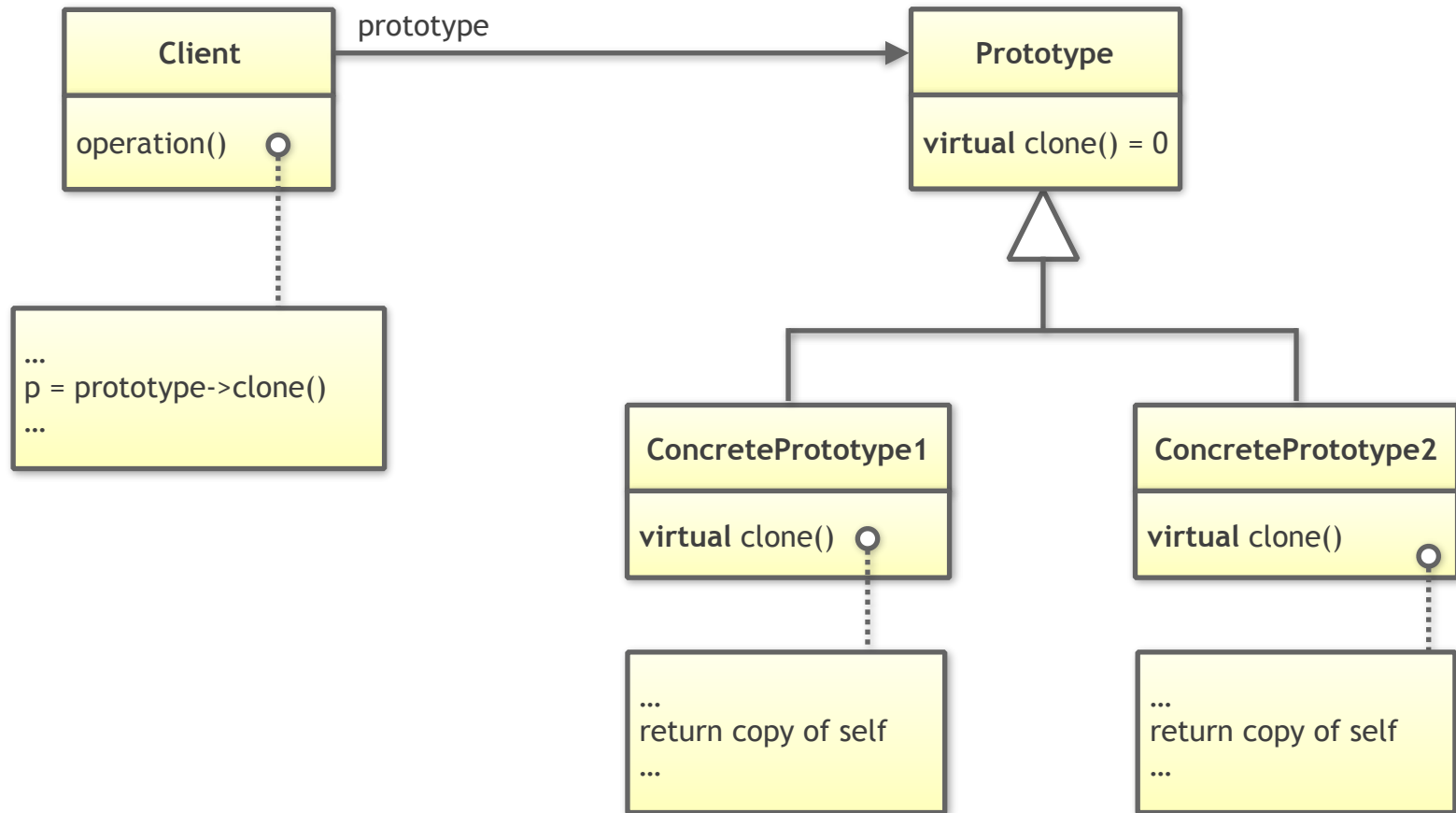


The Classical Observer Pattern

Task (2_Modern_Cpp_Design_Patterns/Observer): Refactor the classical Observer solution by means of the Command pattern. Note that it should still be possible to detach observers from their subjects.

2.8. Prototype

The Prototype Design Pattern



The Prototype Design Pattern

Task (2_Modern_Cpp_Design_Patterns/Prototype): Discuss the advantages and disadvantages of the given implementation of the classical prototype design pattern.

2.9. Type Erasure

std::function – A Simplified Implementation

Task (2_Modern_Cpp_Design_Patterns/Function): Implement a simplified `std::function` to demonstrate the type erasure design pattern.

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() { 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:  
        // ...  
};
```

Applied Type Erasure

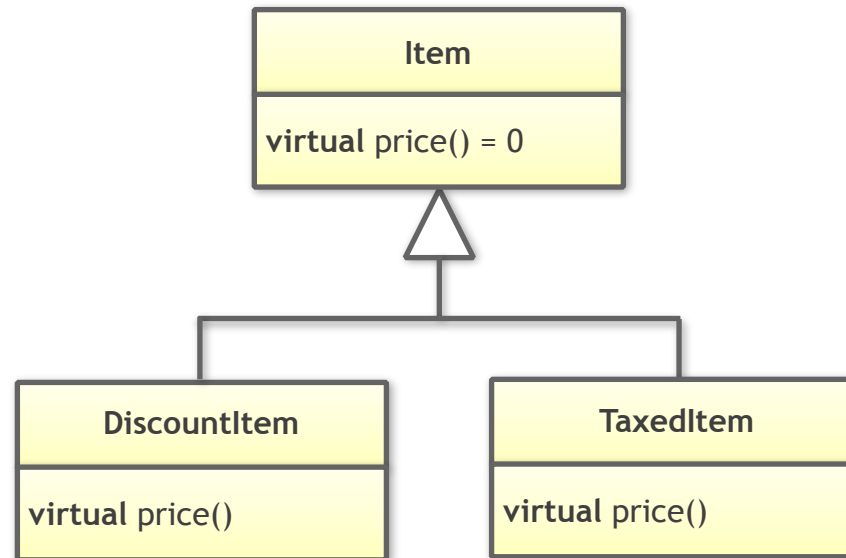
Task (2_Modern_Cpp_Design_Patterns/TypeErasure): Implement the `Shape` class by means of Type Erasure. `Shape` may require all types to provide a free `draw()` function that draws them to the screen.

Guidelines

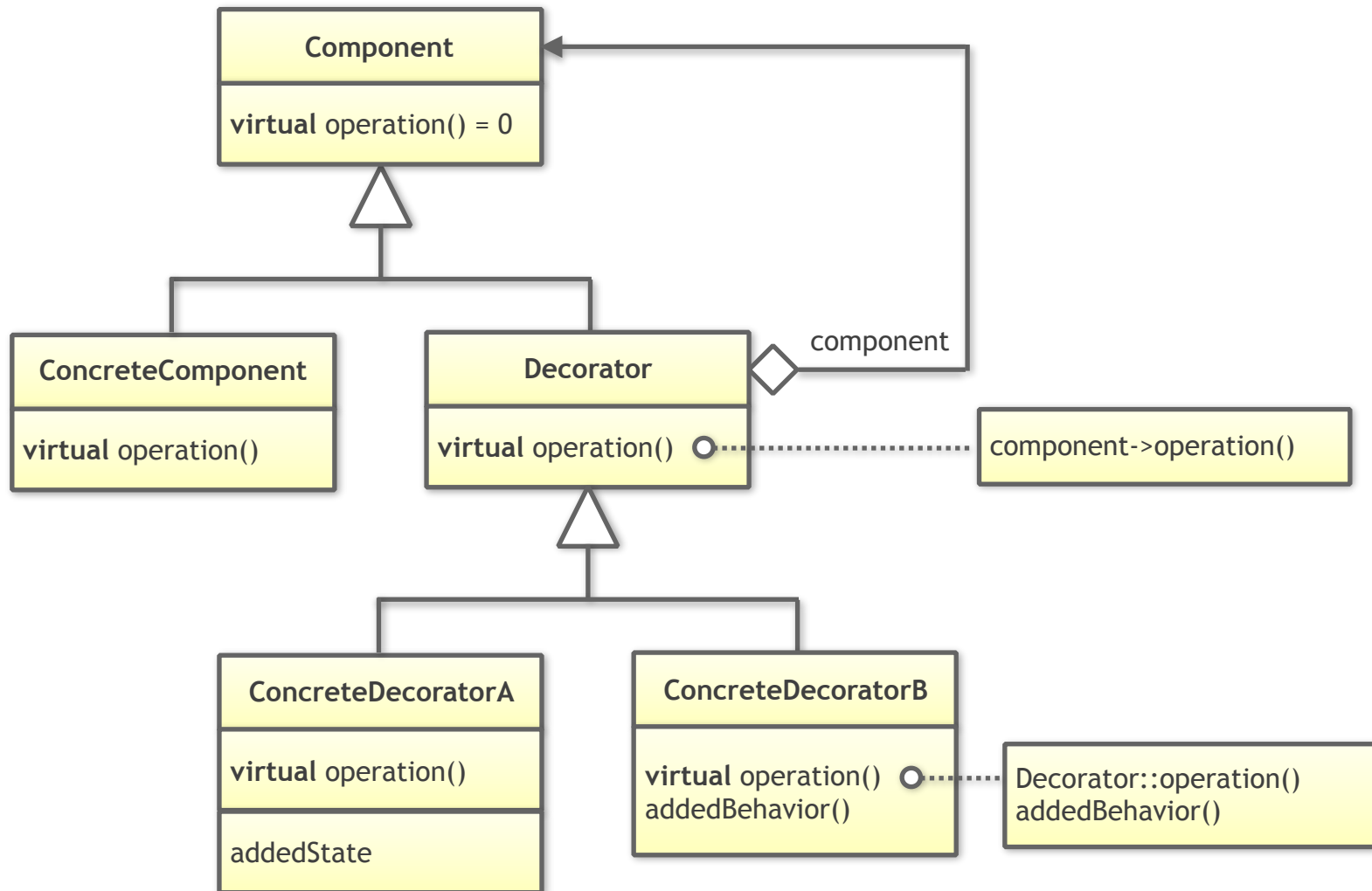
Core Guideline T.49: Where possible, avoid type-erasure

2.10. Decorator

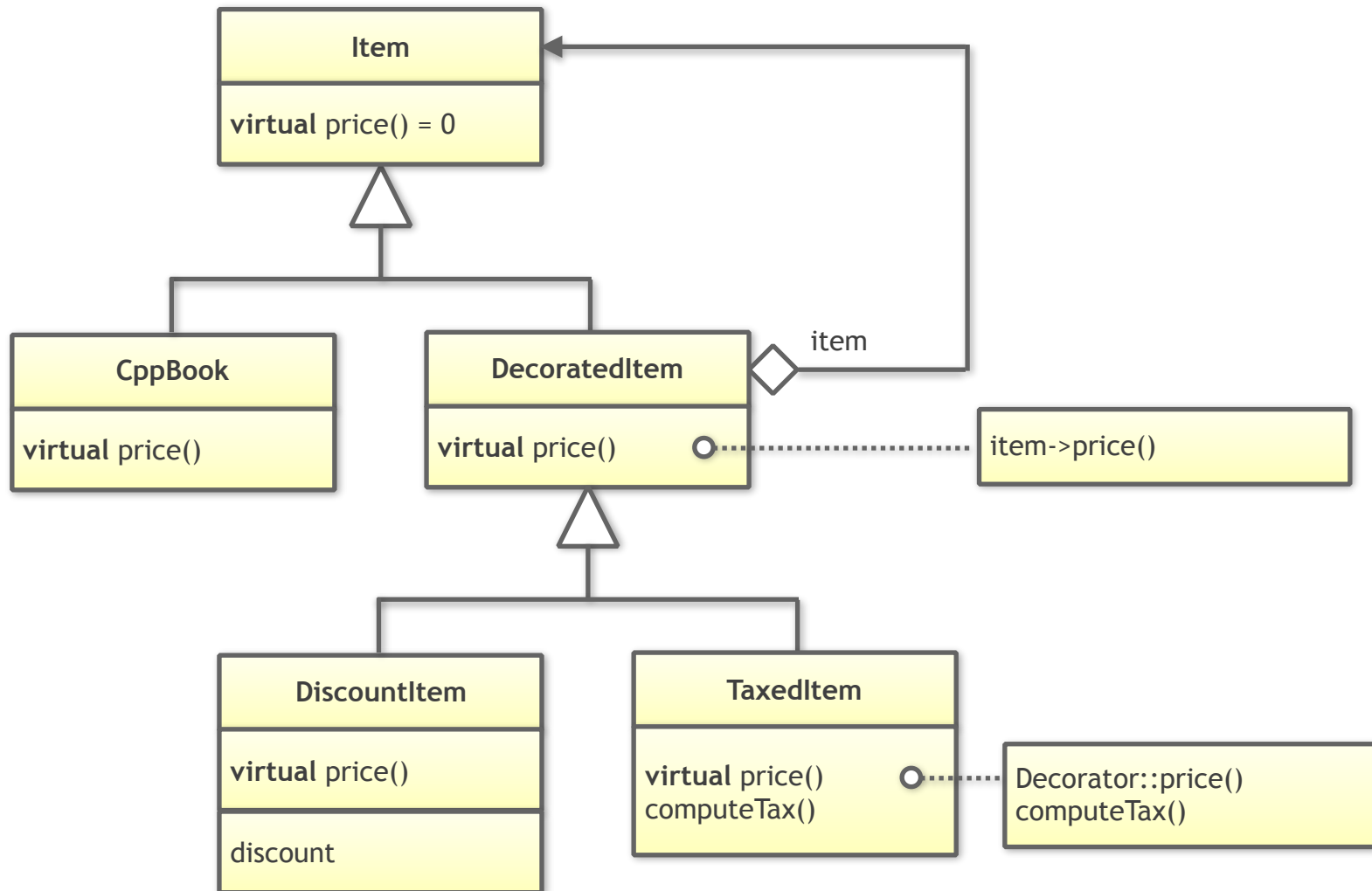
The Classical Decorator Design Pattern



The Classical Decorator Design Pattern



The Classical Decorator Design Pattern



The Classical Decorator Design Pattern

- We have to use virtual functions (no inlining)
- We have to use the heap for each single object
- We have to manage the lifetime via an `std::unique_ptr`
- In every decorator we have to deal with object lifetimes
- Due to pointers we have to deal with potential nullptr

The Classical Decorator Design Pattern

Task (2_Modern_Cpp_Design_Patterns/Decorator): Refactor the classical Decorator solution by a value semantics based solution. Note that the general behavior should remain unchanged.

The Classical Decorator Design Pattern

The classical Decorator design pattern ...

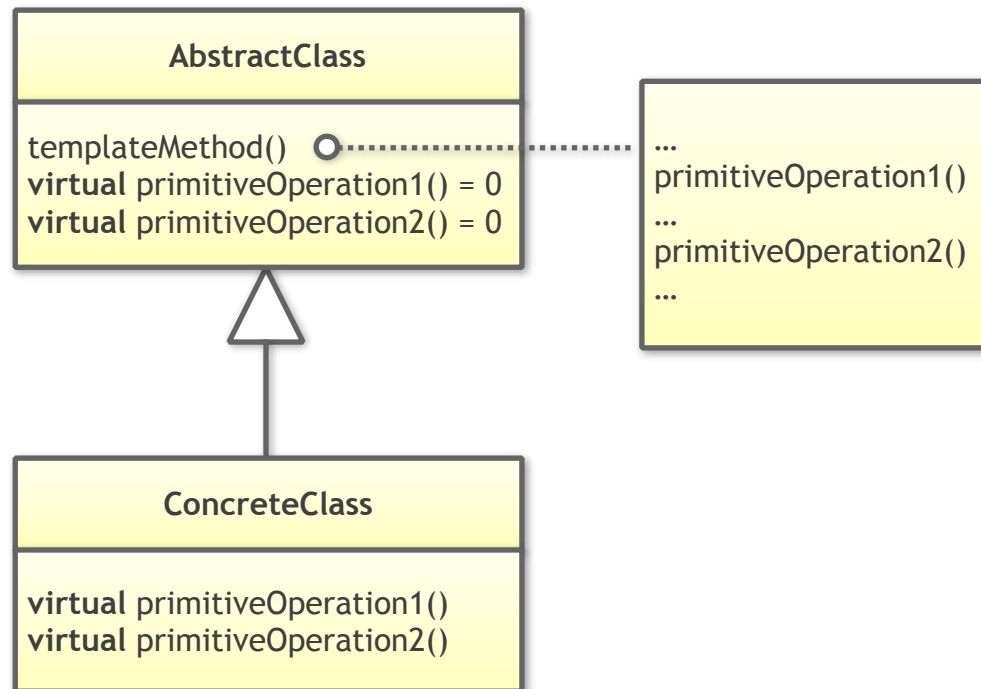
- ... requires a base class (dependency);
- ... promotes heap allocation;
- ... requires memory management;
- ... uses many pointers.

Using type erasure instead of the classical Decorator design pattern ...

- ... simplifies code;
- ... facilitates comprehension;
- ... reduces dependencies;
- ... removes all pointers.

2.11. Template Method

The Classical Template Method Design Pattern



Example

```
class PersistenceInterface
{
public:
    PersistenceInterface();

    virtual ~PersistenceInterface();

    bool write( const Blob& blob );
    bool write( const Blob& blob, WriteCallback callback );
    bool read ( Blob& blob, uint timeout );
    bool read ( Blob& blob, ReadCallback callback, uint timeout );
    // ...

private:
    virtual bool doWrite( const Blob& blob ) = 0;
    virtual bool doWrite( const Blob& blob, WriteCallback callback ) = 0;
    virtual bool doRead ( Blob& blob, uint timeout ) = 0;
    virtual bool doRead ( Blob& blob, ReadCallback callback, uint timeout ) = 0;
    // ...
};
```

Example

```
bool PersistenceInterface::write( const Blob& blob )
{
    TRACE_INFO( "PersistenceInterface::write( Blob ), name = " <<
                blob.name() << ": starting..." );

    if ( blob.name().empty() )
    {
        TRACE_ERROR( "PersistenceInterface::write( Blob ): Attempt to"
                    " write unnamed blob failed" );
        return false;
    }

    const uint32_t start = ThreadHelper::GetTickCount();
    const bool success = doWrite( blob );
    const uint32_t time = ThreadHelper::GetTickCount() - start;

    TRACE_INFO( "PersistenceInterface::write( Blob ), name = " <<
                blob.name() << ": Writing blob of size " << blob.size() <<
                " bytes " << ( success ? "succeeded" : "failed" ) << " in"
                " duration = " << time << "ms" );

    return success;
}
```

Non-Virtual Interface Idiom

Advantages:

- The "wrapper" function can properly set up and tear down the context
- "before stuff" can include verifying class invariants, check function preconditions, lock a mutex, log information, start time measurements, ...
- "after stuff" can include verifying class invariants, check postconditions, unlocking mutexes, log information, stop time measurements, ...
- No performance disadvantage in case of an `inline` function

Disadvantages:

- none

The Expert's View

"The Non-Virtual Interface idiom is the most structured way of using inheritance I know of."

(Stephan T. Lavavej, Core C++, 4 of n)

Non-Virtual Interface Idiom

Note: It is perfectly ok for deriving classes to override a `private` virtual function of the base class!

Usually the virtual function should be implemented in the `private` section. In case they need to be accessed by deriving classes, they may be in the `protected` section.

The sole exception to the NVI: a virtual destructor.

Visibility vs. Accessibility

Task: Which of the following two functions is called in the subsequent function call?

```
class Object
{
    public:
        void doSomething( int );        // (1)
    private:
        void doSomething( double );    // (2)
};

Object obj;
obj.doSomething( 1.0 );
```

The compiler tries to call function (2), but quits the compilation process with an error about an **access violation**: function (2) is declared private!

Visibility vs. Accessibility

Task: Which of the following two functions is called in the subsequent function call?

```
class Object
{
    public:
        void doSomething( int );        // (1)
    private:
        void doSomething( double );    // (2)
};

Object obj;
obj.doSomething( 1U );
```

This results in an **ambiguous function call**. The compiler still sees both functions and cannot decide which conversion to perform!

The Call Resolution Algorithm

Remember the four steps of the compiler to resolve a function call:

1. **Name resolution:** Select all (visible) candidate functions with a certain name within the current scope. If none is found, proceed into the next surrounding scope.
2. **Overload resolution:** Find the best match among the selected candidate functions. If necessary, apply the necessary argument conversions.
3. **Accessibility resolution:** Check if the best match is accessible from the given call site.
4. **Delete resolution:** Check if the best match has been explicitly deleted.

Guidelines

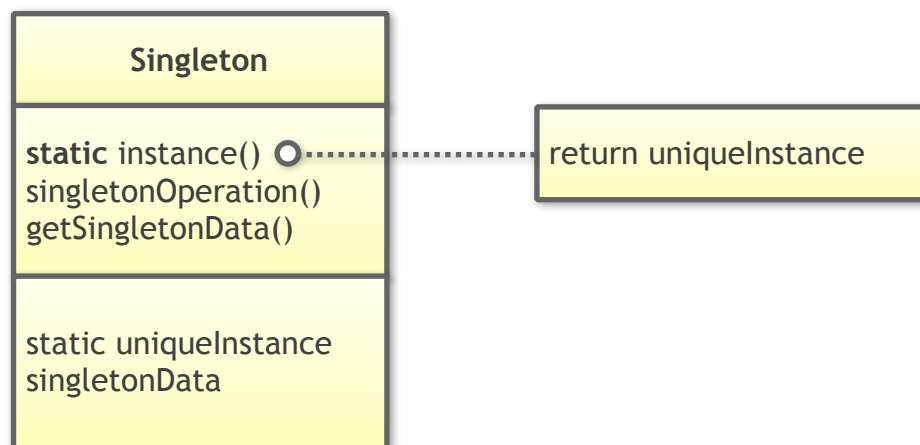
Guideline: Prefer to make interfaces non-virtual, using the Non-Virtual Interface Idiom (sole exception: the destructor).

Guideline: Prefer to make virtual functions private.

Guideline: Only if derived classes need to invoke the base implementation of a virtual function, make the virtual function protected.

2.12. Singleton

The Classical Singleton Design Pattern



The GOF Singleton







The GOF Singleton

```
class Foo
{
public:
    static Foo* Instance()
    {
        if (m_instance == NULL) {
            m_instance = new Foo();
        }
        return m_instance;
    }
    // ...
protected:
    Foo();
    // ...
private:
    static Foo* m_instance;
};
```

Advantages:

 Can be accessed anytime

Disadvantages:

-  Resource leak
-  Unclear responsibilities
-  Bad semantics
-  Not thread safe
-  Cannot handle dependencies
-  Testing very difficult

An „Improved“ Singleton Implementation

An „Improved“ Singleton Implementation

```
class Foo {
public:
    // ...
    static Foo* CreateInstance( int i, double d )
    {
        if (m_instance == NULL) {
            m_instance = new Foo( i, d );
        }
        return m_instance;
    }
    static Foo* Instance()
    {
        return m_instance;
    }
    static void DeleteInstance()
    {
        if (m_instance != NULL) {
            delete m_instance;
            m_instance = NULL;
        }
    }
    // ...
private:
    static Foo* m_instance;
};
```







An „Improved“ Singleton Implementation

```
class Foo {
public:
    // ...
    static Foo* CreateInstance( int i, double d )
    {
        if (m_instance == NULL) {
            m_instance = new Foo( i, d );
        }
        return m_instance;
    }
    static Foo* Instance()
    {
        return m_instance;
    }
    static void DeleteInstance()
    {
        if (m_instance != NULL) {
            delete m_instance;
            m_instance = NULL;
        }
    }
    // ...
private:
    static Foo* m_instance;
};
```

Advantages:

 Can be accessed anytime

Disadvantages:

-  Resource leak
-  Unclear responsibilities
-  Bad semantics
-  Not thread safe
-  Cannot handle dependencies
-  Testing very difficult








An „Improved“ Singleton Implementation

```
class Foo {  
public:  
    // ...  
    static Foo* CreateInstance( int i, double d )  
    {  
        if (m_instance == NULL) {  
            m_instance = new Foo( i, d );  
        }  
        return m_instance;  
    }  
    static Foo* Instance()  
    {  
        return m_instance;  
    }  
    static void DeleteInstance()  
    {  
        if (m_instance != NULL) {  
            delete m_instance;  
            m_instance = NULL;  
        }  
    }  
    // ...  
private:  
    static Foo* m_instance;  
};
```

Advantages:

 Can be accessed anytime

Disadvantages:

-  Resource leak
-  Unclear responsibilities
-  Bad semantics
-  Not thread safe
-  Cannot handle dependencies
-  Testing very difficult
-  Cannot be accessed anytime








An „Improved“ Singleton Implementation

```
class Foo {  
public:  
    // ...  
    static Foo* CreateInstance( int i, double d )  
    {  
        if (m_instance == NULL) {  
            m_instance = new Foo( i, d );  
        }  
        return m_instance;  
    }  
    static Foo* Instance()  
    {  
        return m_instance;  
    }  
    static void DeleteInstance()  
    {  
        if (m_instance != NULL) {  
            delete m_instance;  
            m_instance = NULL;  
        }  
    }  
    // ...  
private:  
    static Foo* m_instance;  
};
```

Advantages:

 Can be accessed anytime

Disadvantages:

-  Potential resource leak
-  Unclear responsibilities
-  Bad semantics
-  Not thread safe
-  Cannot handle dependencies
-  Testing very difficult
-  Cannot be accessed anytime

Meyer's Singleton







Meyer's Singleton

```
class Foo {  
public:  
    // ...  
    static Foo& Instance()  
    {  
        static Foo m_instance;  
        return m_instance;  
    }  
    // ...  
};
```

Advantages:

 Can be accessed anytime

Disadvantages:

-  Resource leak
-  Unclear responsibilities
-  Bad semantics
-  Not thread safe
-  Cannot handle dependencies
-  Testing very difficult

Meyer's Singleton

```
class Foo {  
public:  
    // ...  
    static Foo& Instance()  
    {  
        static Foo m_instance;  
        return m_instance;  
    }  
    // ...  
};
```

Advantages:

- ☑ Can be accessed anytime
- ☑ No resource leak

Disadvantages:

- ✗ Resource leak
- ✗ Unclear responsibilities
- ✗ Bad semantics
- ✗ Not thread safe
- ✗ Cannot handle dependencies
- ✗ Testing very difficult

Meyer's Singleton

```
class Foo {  
public:  
    // ...  
    static Foo& Instance()  
    {  
        static Foo m_instance;  
        return m_instance;  
    }  
    // ...  
};
```

Advantages:

- ✓ Can be accessed anytime
- ✓ No resource leak
- ✓ Clear responsibilities

Disadvantages:

- ✗ Resource leak
- ✗ Unclear responsibilities
- ✗ Bad semantics
- ✗ Not thread safe
- ✗ Cannot handle dependencies
- ✗ Testing very difficult

Meyer's Singleton

```
class Foo {  
public:  
    // ...  
    static Foo& Instance()  
    {  
        static Foo m_instance;  
        return m_instance;  
    }  
    // ...  
};
```

Advantages:

- ✓ Can be accessed anytime
- ✓ No resource leak
- ✓ Clear responsibilities/ semantics

Disadvantages:

- ✗ Resource leak
- ✗ Unclear responsibilities
- ✗ Bad semantics
- ✗ Not thread safe
- ✗ Cannot handle dependencies
- ✗ Testing very difficult

Meyer's Singleton

```
class Foo {  
public:  
    // ...  
    static Foo& Instance()  
    {  
        static Foo m_instance;  
        return m_instance;  
    }  
    // ...  
};
```

Advantages:

- ☑ Can be accessed anytime
- ☑ No resource leak
- ☑ Clear responsibilities/ semantics
- ☑ Thread safe (in C++11)

Disadvantages:

- ✗ Resource leak
- ✗ Unclear responsibilities
- ✗ Bad semantics
- ✗ Not thread safe
- ✗ Cannot handle dependencies
- ✗ Testing very difficult

Observations

- Singletons create an “invisible” dependency
- “Singletons are pathological liars” (M. Hevery)
- Singletons are inherently difficult to test
- Singletons make testing much more difficult

Further Observations

- But so does the Monostate design pattern

```
class Monostate
{
public:
    Monostate(); ← // No limitation to the number of
                  objects

private:
    static A a_; ← // Only static data members
    static B b_;
    // ...
};
```

- The root cause of the singleton problem is global data
- Global data must be avoided as much as possible

Guidelines

Guideline: Avoid global data as much as possible. Global data severely impedes testability and heavily increases the dependencies within code.

“Right” Use of Singletons

- Singletons with data flow in one direction (e.g. logger) may be acceptable
- Singleton implementations may require a built-in mechanism to reset the singleton:

```
class Singleton()  
{  
    private:  
        Singleton();  
        ~Singleton();  
  
    public:  
        static Singleton& Instance()  
  
        static void ReinstantiateSingleton();  
}
```

“Right” Use of Singletons

- Testability can be improved by separating logic from singleton use:

```
void function_to_test()
{
    // ...
    double const value = Singleton::Instance()->GetValue();
    // ...
}
```

“Right” Use of Singletons

- Testability can be improved by separating logic from singleton use:

```
    // Note that the function name may be changed, but
    // may also participate in overload resolution
void function_to_test( double value )
{
    // ...
    // No need to know about the singleton
    // Dependency to 'Singleton' is completely removed
    // ...
}

void function_to_test()
{
    double const value = Singleton::Instance()->GetValue();
    function_to_test( value );
}
```


A “Modern” Singleton Implementation

Task (2_Modern_Cpp_Design_Patterns/Singleton): Evaluate the following Singleton implementation with respect to life-time guarantees.

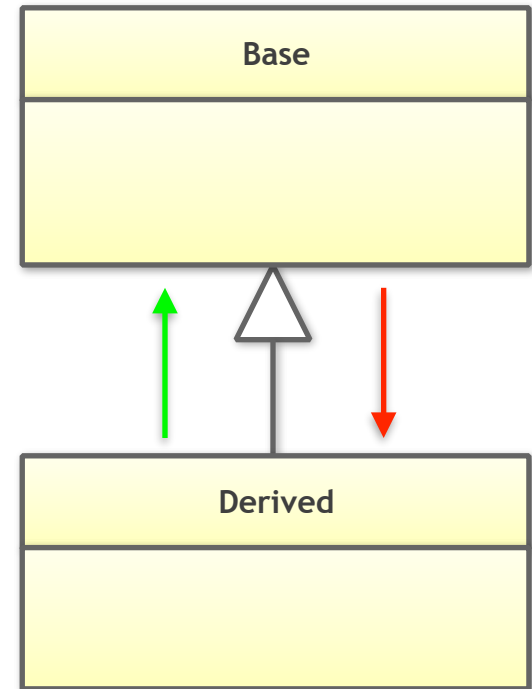
2.13. CRTP

The CRTP Design Pattern

This is an example for the **CRTP** (Curiously Recurring Template Pattern).

```
template< typename T >
struct Base
{
    ~Base();
};

struct Derived
{
    ~Derived();
    // ...
    void print() { std::cout << "Derived"; }
};
```

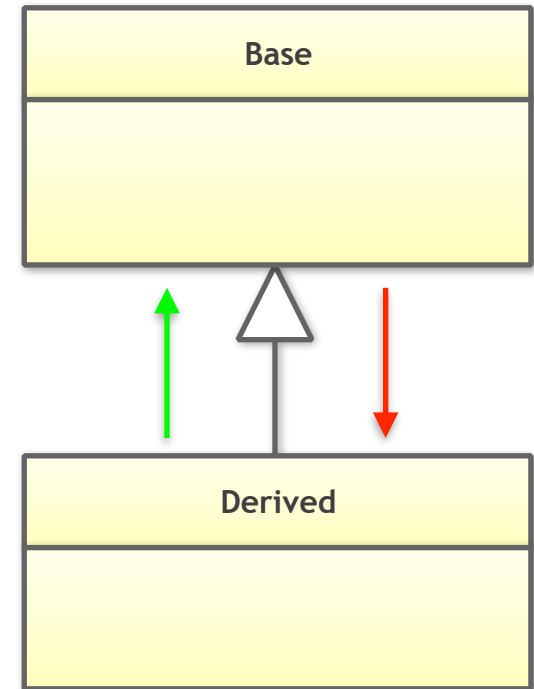


The CRTP Design Pattern

This is an example for the **CRTP** (Curiously Recurring Template Pattern).

```
template< typename T >
struct Base
{
    ~Base();
};

struct Derived : public Base<Derived>
{
    ~Derived();
    // ...
    void print() { std::cout << "Derived"; }
};
```



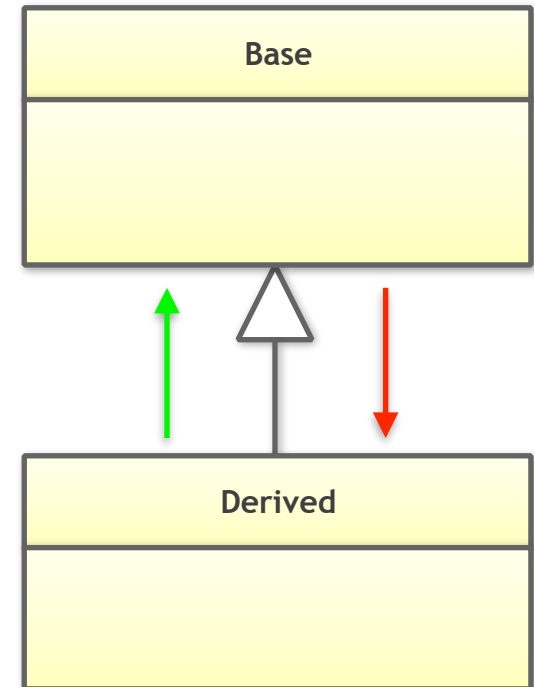
The CRTP Design Pattern

This is an example for the **CRTP** (Curiously Recurring Template Pattern).

```
template< typename T >
struct Base
{
    ~Base();

    void print() {
        static_cast<T*>(*this).print();
    }
};

struct Derived : public Base<Derived>
{
    ~Derived();
    // ...
    void print() { std::cout << "Derived"; }
};
```



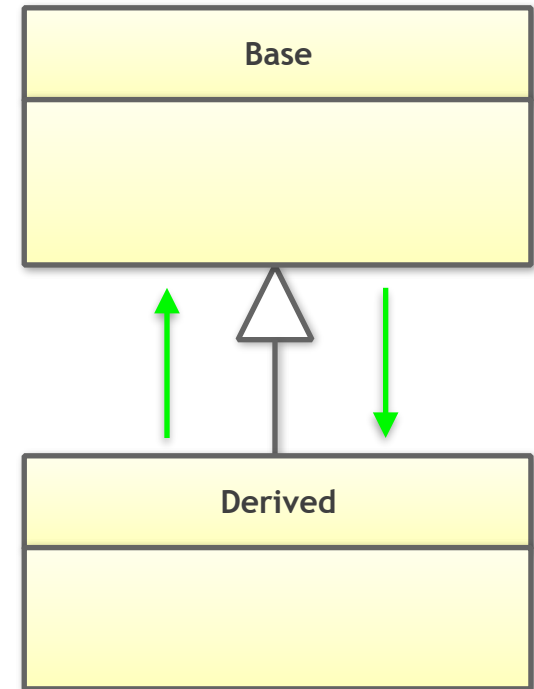
The CRTP Design Pattern

This is an example for the **CRTP** (Curiously Recurring Template Pattern).

```
template< typename T >
struct Base
{
    ~Base();

    void print() {
        static_cast<T*>(*this).print();
    }
};

struct Derived : public Base<Derived>
{
    ~Derived();
    // ...
    void print() { std::cout << "Derived"; }
};
```



The CRTP Design Pattern

Task (2_Modern_Cpp_Design_Patterns/CRTP): Implement a common output operator for both `DynamicVector` and `StaticVector` by means of the CRTP design pattern.

The CRTP Design Pattern

This formulation lacks any kind of abstraction:

```
template< typename T >  
std::ostream&  
    operator<<( std::ostream&, const DynamicVector<T>& );
```

This formulation is too abstract (takes everything):

```
template< typename VectorType >  
std::ostream&  
    operator<<( std::ostream&, const VectorType& );
```


The CRTP Design Pattern

```
template< typename Derived >
class DenseVector
{
public:
    constexpr size_t size() const noexcept {
        return static_cast<const Derived&>( *this ).size();
    }
    // ...
};
```

```
template< typename T >
class DynamicVector
    : public DenseVector< DynamicVector<T> >
{
public:
    size_t size() const noexcept { return ...; }
};
```

The CRTP Design Pattern

This formulation lacks any kind of abstraction:

```
template< typename T >
std::ostream&
    operator<<( std::ostream&, const DynamicVector<T>& );
```

This formulation is too abstract (takes everything):

```
template< typename VectorType >
std::ostream&
    operator<<( std::ostream&, const VectorType& );
```

CRTP-based formulation works (only) for all kinds of dense vector:

```
template< typename Derived >
std::ostream&
    operator<<( std::ostream&, const DenseVector<Derived>& );
```

CRTP – Advantages/Disadvantages

Use CRTP if ...

- ... you don't need a **common base class**;
- ... you functions that use the base class may be **templates**;
- ... you don't need to **explicitly use** the base class;
- ... the **virtual call overhead** is not acceptable.

Don't use CRTP if ...

- ... you need a **common base class**;
- ... you need code to reside in **source files**;
- ... you need to use the base class across **architectural boundaries**.

2.14. Expression Templates

Expression Templates

```
DynamicVector<double> a, b, c;  
// ... Initialization of vector a and b  
c = a + b;
```

Expression Templates

```
inline const Vector
    operator+( const Vector& a, const Vector& b )
{
    Vector tmp( a.size() );
    for( std::size_t i=0; i<a.size(); ++i )
        tmp[i] = a[i] + b[i];
    return tmp;
}
```

Expression Templates

```
for( size_t i=0; i<size; ++i )  
    c[i] = a[i] + b[i];
```

Expression Templates

The idea of expression templates:

- Defer the computation until it is required (lazy evaluation)
- Instead of returning a temporary vector, return a proxy object
- The proxy object represents the result of the addition
- The result is computed when accessing the proxy

Expression Templates

```
template< typename A, typename B >
class Sum {
public:
    Sum( const A& a, const B& b ) : a_( a ), b_( b )
    {}

    std::size_t size() const { return a_.size(); }

    double operator[]( std::size_t i ) const
    { return a_[i] + b_[i]; }

private:
    const A& a_; // Reference to the left-hand side operand
    const B& b_; // Reference to the right-hand side operand
};
```

Expression Templates

```
template< typename A, typename B >
const Sum<A,B> operator+( const A& a, const B& b )
{
    return Sum<A,B>( a, b );
}
```

Expression Templates

```
class Vector {  
public:  
    // ...  
    template< typename A >  
    Vector& operator=( const A& expr ) {  
        resize( expr.size() );  
        for( std::size_t i=0; i<expr.size(); ++i )  
            v_[i] = expr[i];  
        return *this;  
    }  
    // ...  
};
```

Programming Task

Task (2_Modern_Cpp_Design_Patterns/ExpressionTemplates): Use Expression Templates to implement the addition operation. Benchmark the addition of two small (in-cache) and two large (out-of-cache) vectors.

Expression Templates

Let's consider the STL ...

Programming Task

Task (2_Modern_Cpp_Design_Patterns/Ranges_v3): Modify the given example of the `ranges_v3` library to compute the first four squares of odd numbers.



BATTLE OF THE ALGORITHMS

Challenge 1

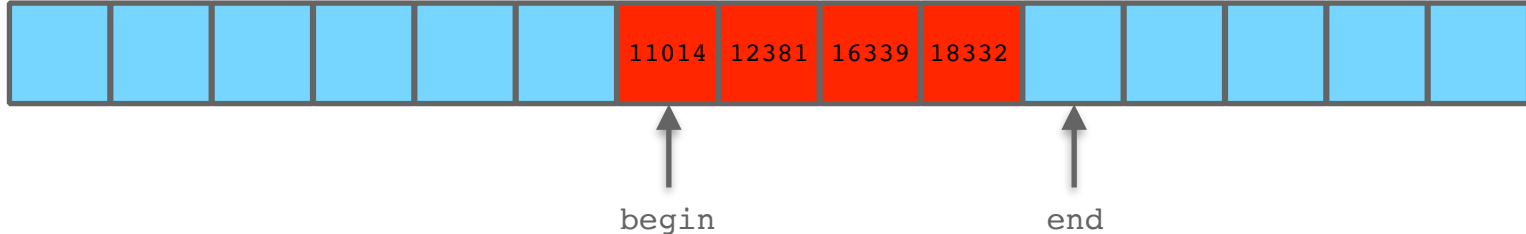
You need the first occurrence of the word `mega` in a long string.



`std::search`

Challenge 2

You have a large collection of players, sorted by score, and need to get all players with a score between 10000 and 20000.



`std::equal_range`

Challenge 3

You want to exclude all duplicate players contained in a sorted vector that also appear in a sorted list.



Challenge 3

You want to exclude all duplicate players contained in a sorted vector that also appear in a sorted list.



Challenge 3

You want to exclude all duplicate players contained in a sorted vector that also appear in a sorted list.



`std::set_difference`

Challenge 4

You have an unsorted vector of players and need the first player with a score higher than 10000.



`std::find_if`

Challenge 5

You want to know if one string is the prefix of another string.

p	r	e	f	i	x
---	---	---	---	---	---

p	r	e	f	i	x	r	e	s	t
---	---	---	---	---	---	---	---	---	---

`std::mismatch`

Challenge 6

You want to initialize raw memory with strings of a specific value.



`std::uninitialized_fill`

Challenge 7

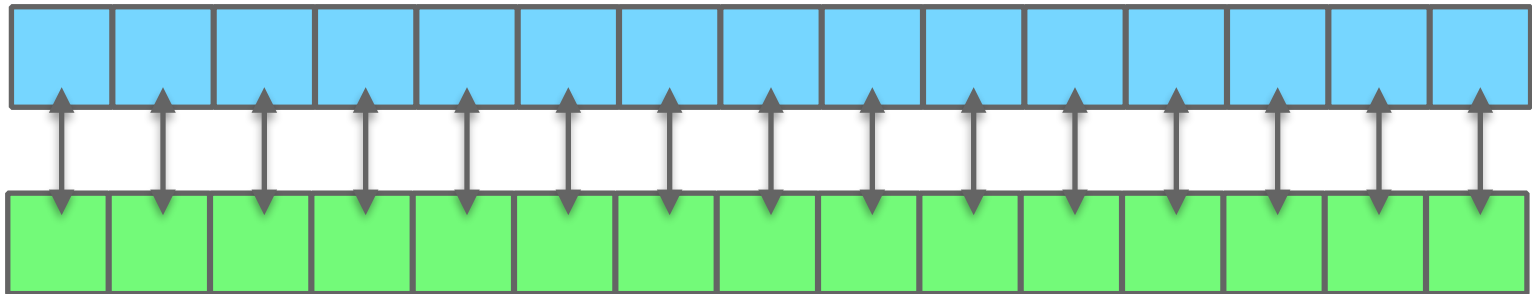
You want to know if any department stored in a deque has more than 100 employees.



`std::any_of`

Challenge 8

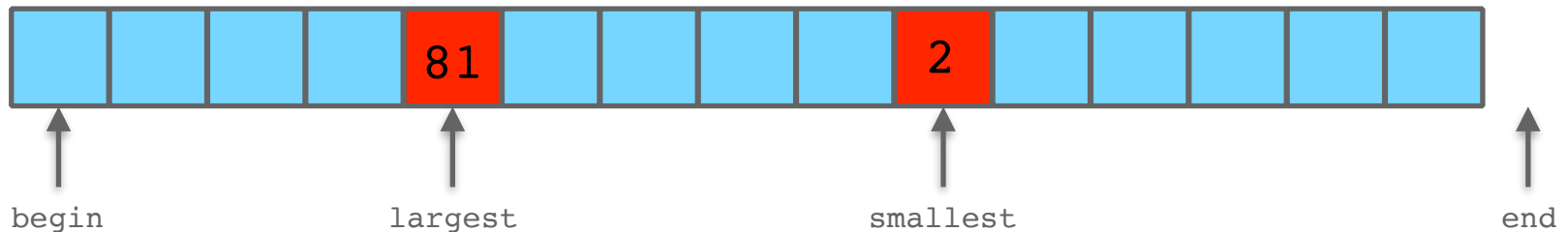
You want to swap the elements of a vector with the elements of an array.



`std::swap_ranges`

Challenge 9

You need to calculate the difference between the smallest and largest element of a collection.



`std::minmax_element`

Challenge 10

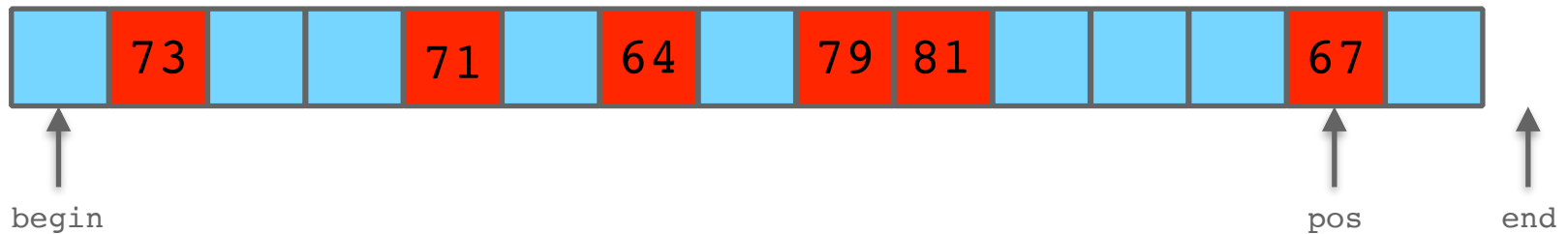
You want to know if all employees in one department are properly assigned the correct department flag.



`std::all_of`

Challenge 11

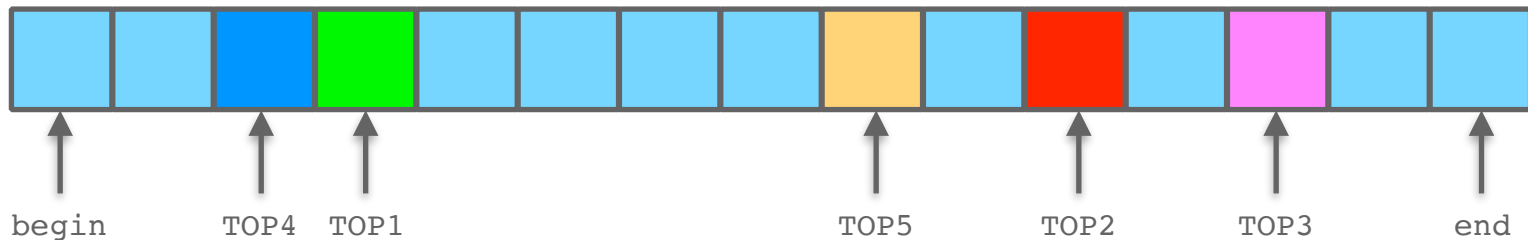
You want to find the last customer in an unsorted queue that is above a certain age.



std::find_end

Challenge 12

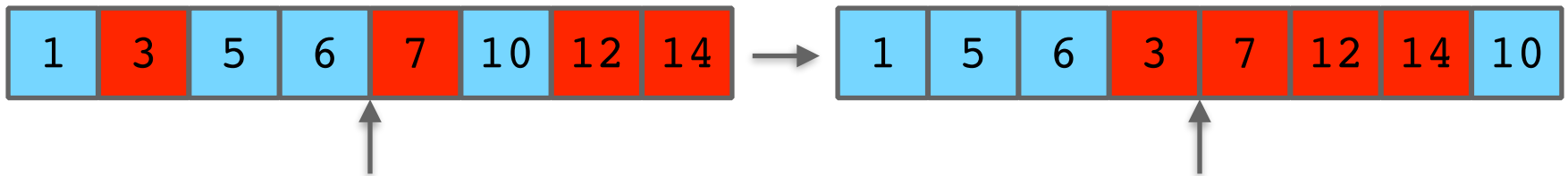
You want to display the TOP5 players in-order based on their high score.



`std::partial_sort`

Challenge 13

You have manually selected several items and want to collect them at another, specific position.



`std::partition`

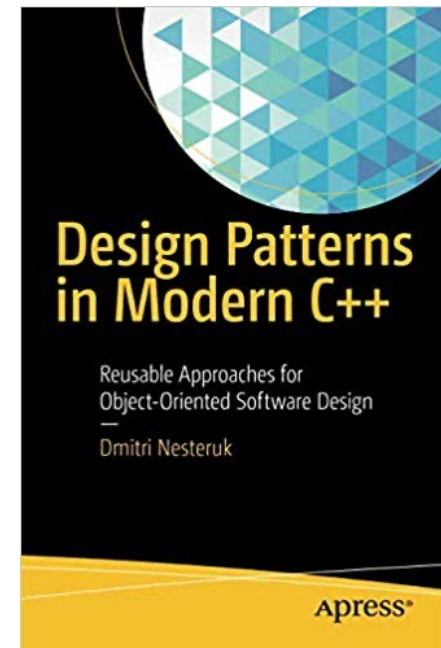
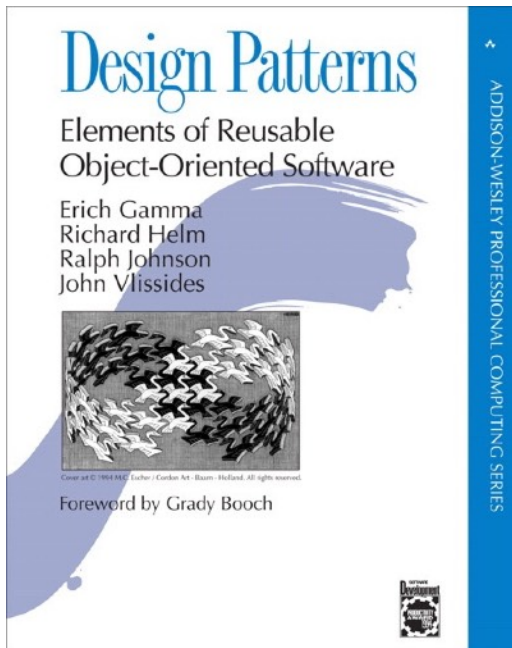
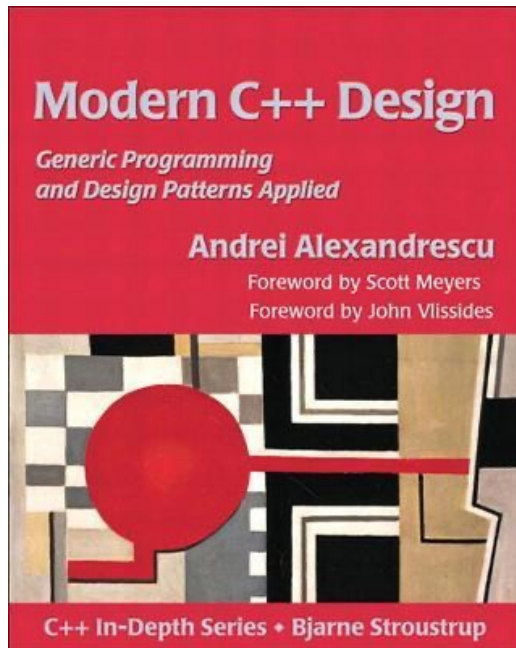
Programming Task

Task (2_Modern_Cpp_Design_Patterns/Ranges): Fix the implementation of the `TransformExpr` expression to apply the given operation on the given range.

Things to Remember

- Minimize couplings wherever possible
- Consider modern programming techniques to break inheritance relationships
- Prefer value semantics based solutions
- Try to reduce the use of pointers

Literature



References

- E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns - Elements of Reusable Object-Oriented Software, Addison-Wesley
- A. Alexandrescu: Modern C++ Design - Generic Programming and Design Patterns Applied, Addison-Wesley
- M. Hevery: Singletons are Pathological Liars (<http://misko.hevery.com/2008/08/17/singletons-are-pathological-liars/>)
- M. Hevery: Where Have All the Singletons Gone? (<http://misko.hevery.com/2008/08/21/where-have-all-the-singletons-gone/>)
- M. Hevery: Root Cause of Singletons (<http://misko.hevery.com/2008/08/25/root-cause-of-singletons/>)
- S. Meyers and A. Alexandrescu: C++ and the Perils of Double-Checked Locking, Sept. 2004 (http://www.aristeia.com/Papers/DDJ_Jul_Aug_2004_revised.pdf)

Online Resources

- C++ Reference: www.cppreference.com
- C++ Core Guidelines: isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines
- Stackoverflow: www.stackoverflow.com
- Compiler Explorer: www.godbolt.org
- Quick-Bench: www.quickbench.com
- C++ Insights: www.cppinsights.io
- Intel Intrinsics Guide: software.intel.com/sites/landingpage/IntrinsicsGuide

klaus.iglberger@gmx.de