Back to Basics:
Designing Classes (part 1 of 2)

KLAUS IGLBERGER

Cppcon
The C++ Conference

2021

October 24-29

C++ Trainer/Consultant

Author of the blaze C++ math library

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

Chair of the CppCon B2B and SD tracks

Email: klaus.iglberger@gmx.de

Klaus Iglberger

Content

Back to Basics: Class Design (Part 1)

- The Challenge of Class Design
- Design Guidelines
  - Design for Readability
  - Design for Change and Extension
  - Design for Testability
- Implementation Guidelines
  - Resource Management

Back to Basics: Class Design (Part 2)

- Implementation Guidelines
  - Data Member Initialization
  - Implicit Conversions
  - Order of Data Members
  - Const Correctness
  - Encapsulating Design Decisions
  - Qualified/Modified Member Data
  - Visibility vs. Accessibility

The Challenge of Class Design

Back to Basics: Class Design (Part 1)

- The Challenge of Class Design
- Design Guidelines
  - Design for Readability
  - Design for Change and Extension
  - Design for Testability
- Implementation Guidelines
  - Resource Management

Back to Basics: Class Design (Part 2)

- Implementation Guidelines
  - Data Member Initialization
  - Implicit Conversions
  - Order of Data Members
  - Const Correctness
  - Encapsulating Design Decisions
  - Qualified/Modified Member Data
  - Visibility vs. Accessibility

The Challenge of Class Design

What is the root source of all problems in software development?

Change

The Challenge of Class Design

The truth in our industry:

Software must be adaptable to frequent changes

The Challenge of Class Design

The truth in our industry:

Software must be adaptable to frequent changes

The Challenge of Class Design

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

Dependencies

The Challenge of Class Design

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

The Challenge of Class Design

Guideline: Design classes for easy change.

Guideline: Design classes for easy extensions.

Design Guidelines

Back to Basics: Class Design (Part 1)

- The Challenge of Class Design
- Design Guidelines
  - Design for Readability
  - Design for Change and Extension
  - Design for Testability
- Implementation Guidelines
  - Resource Management

Back to Basics: Class Design (Part 2)

- Implementation Guidelines
  - Data Member Initialization
  - Implicit Conversions
  - Order of Data Members
  - Const Correctness
  - Encapsulating Design Decisions
  - Qualified/Modified Member Data
  - Visibility vs. Accessibility

Design for Readability

Back to Basics: Class Design (Part 1)

- The Challenge of Class Design
- Design Guidelines
  - Design for Readability
  - Design for Change and Extension
  - Design for Testability
- Implementation Guidelines
  - Resource Management

Back to Basics: Class Design (Part 2)

- Implementation Guidelines
  - Data Member Initialization
  - Implicit Conversions
  - Order of Data Members
  - Const Correctness
  - Encapsulating Design Decisions
  - Qualified/Modified Member Data
  - Visibility vs. Accessibility

Design for Readability

Guideline: Spent time to find good names for all entities.

```
template<
    class T,
    std::size_t N
> struct array;
```

What does 'N' represent?

Design for Readability

Guideline: Spent time to find good names for all entities.

```
template<
    class T,
    std::size_t Size
> struct array;
```

Now it's clear!

Design for Readability

Guideline: Spent time to find good names for all entities.

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

Container or numerical vector?

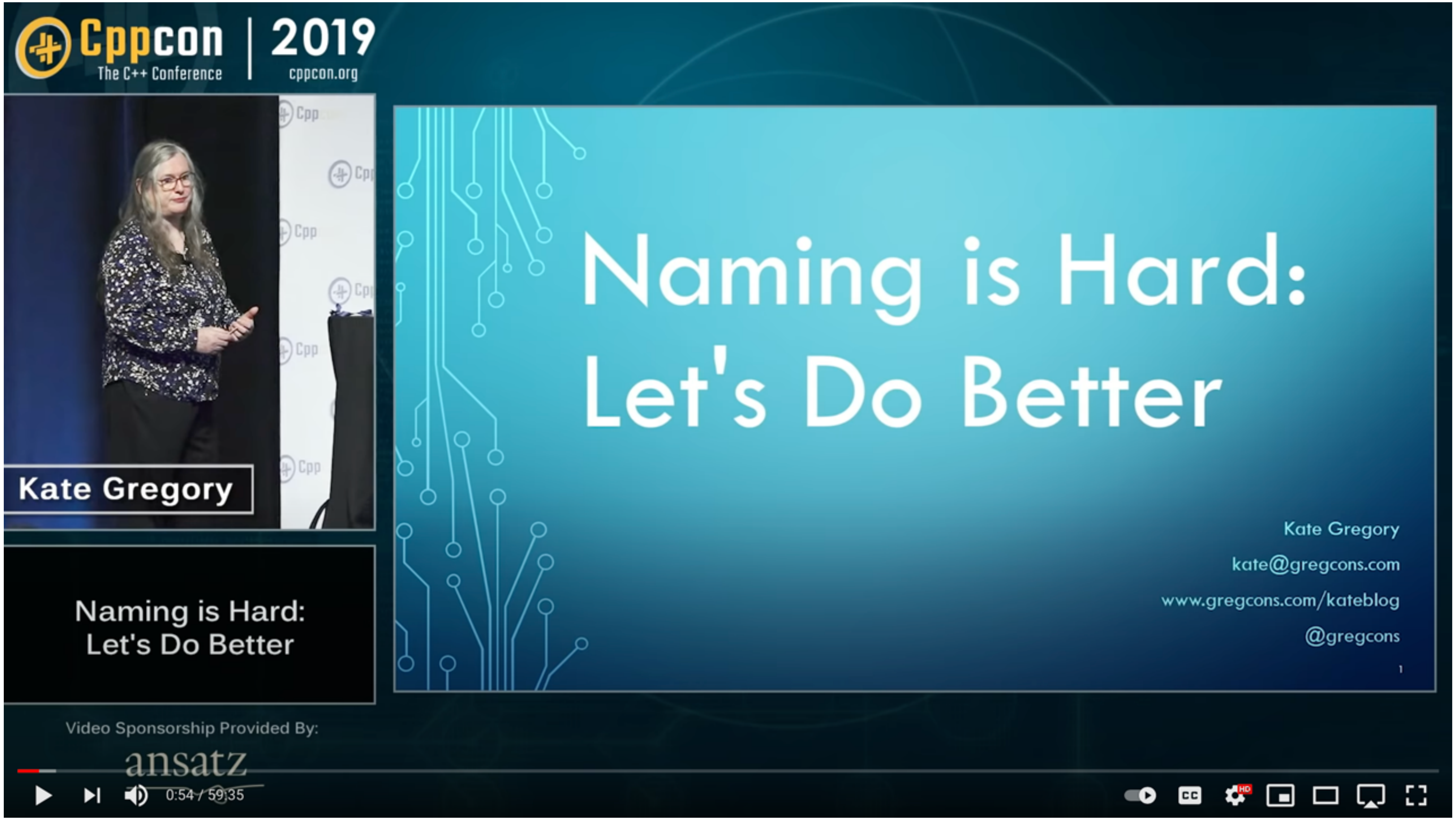
Design for Readability

Guideline: Spent time to find good names for all entities.

```
template<
    class T,
    class Allocator = std::allocator<T>
> class vector
{
 public:
   // ...

   [[nodiscard]] constexpr bool empty() const noexcept;

   // ...
};
```

Action or query?

Design for Readability



Cppcon | 2019

The C++ Conference

cppcon.org

Kate Gregory

Naming is Hard:
Let's Do Better

Video Sponsorship Provided By:

ansatz

Naming is Hard: Let's Do Better

Kate Gregory
kate@gregcons.com
www.gregcons.com/kateblog
@gregcons

1

Design for Readability

Guideline: Spent time to find good names for all entities.

”Naming requires Empathy.”

(Kate Gregory, Naming is Hard: Let’s Do Better, CppCon 2019)

Design for Change and Extension

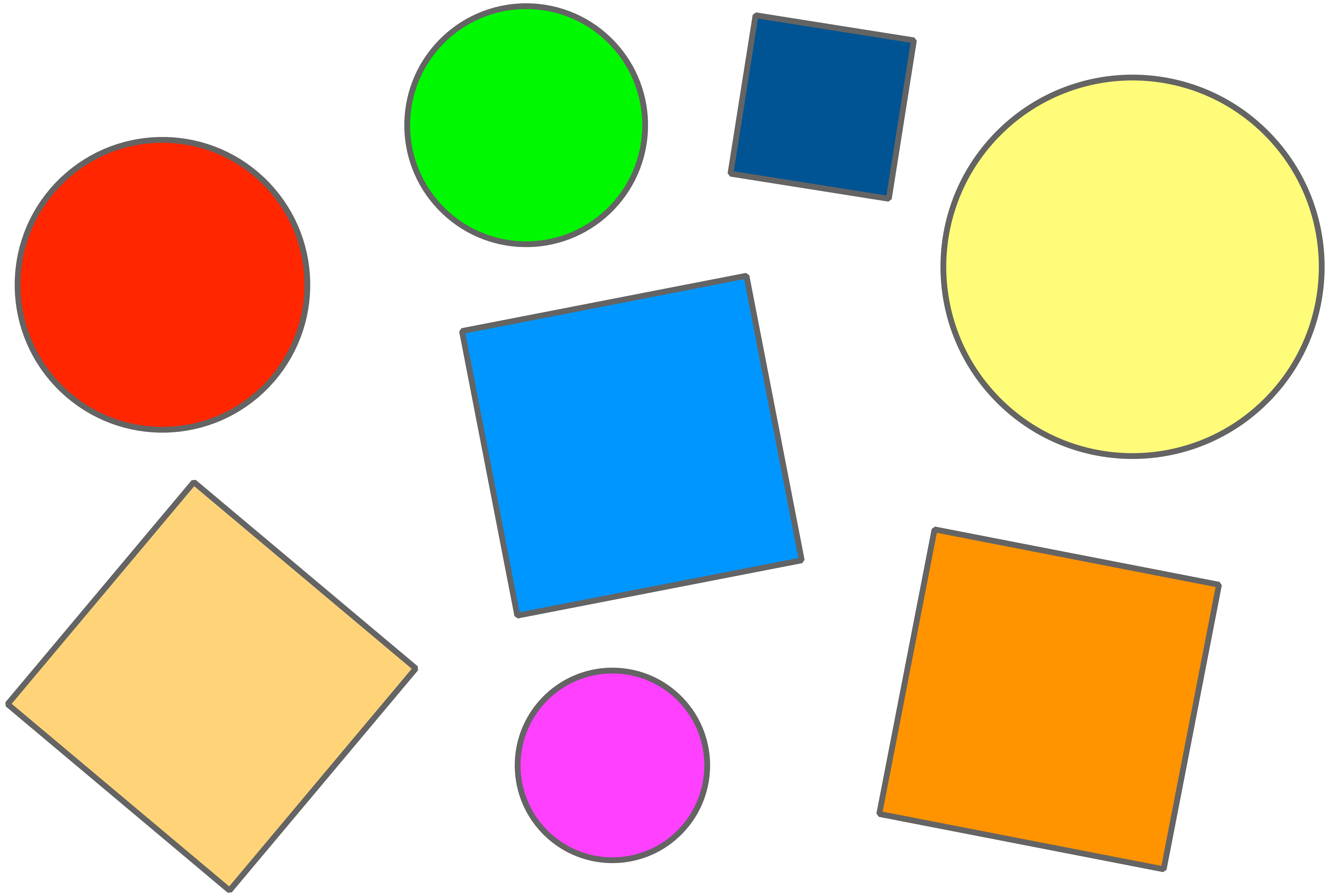
Back to Basics: Class Design (Part 1)

- The Challenge of Class Design
- Design Guidelines
  - Design for Readability
  - Design for Change and Extension
  - Design for Testability
- Implementation Guidelines
  - Resource Management

Back to Basics: Class Design (Part 2)

- Implementation Guidelines
  - Data Member Initialization
  - Implicit Conversions
  - Order of Data Members
  - Const Correctness
  - Encapsulating Design Decisions
  - Qualified/Modified Member Data
  - Visibility vs. Accessibility

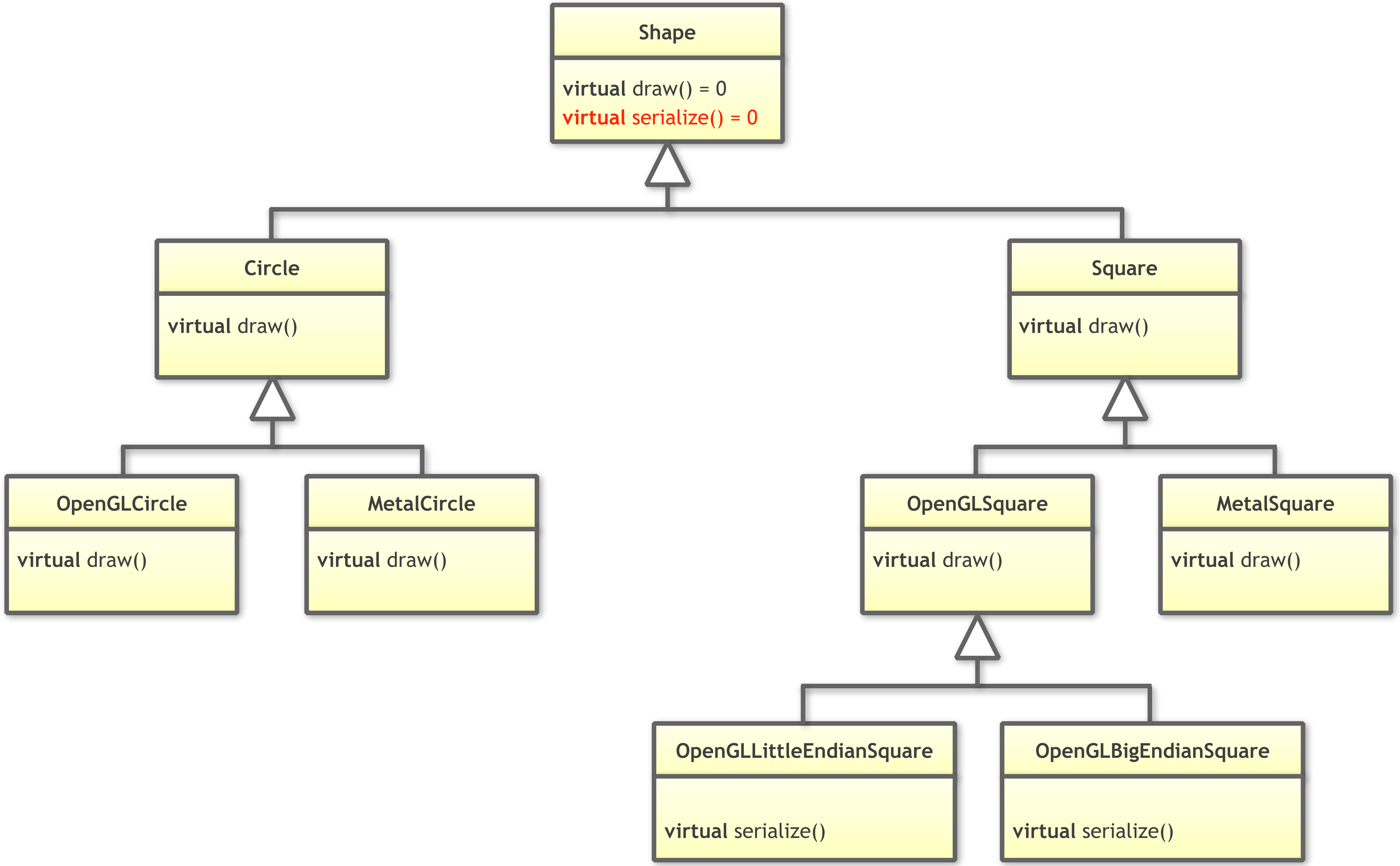
Our First Toy Problem: Shapes



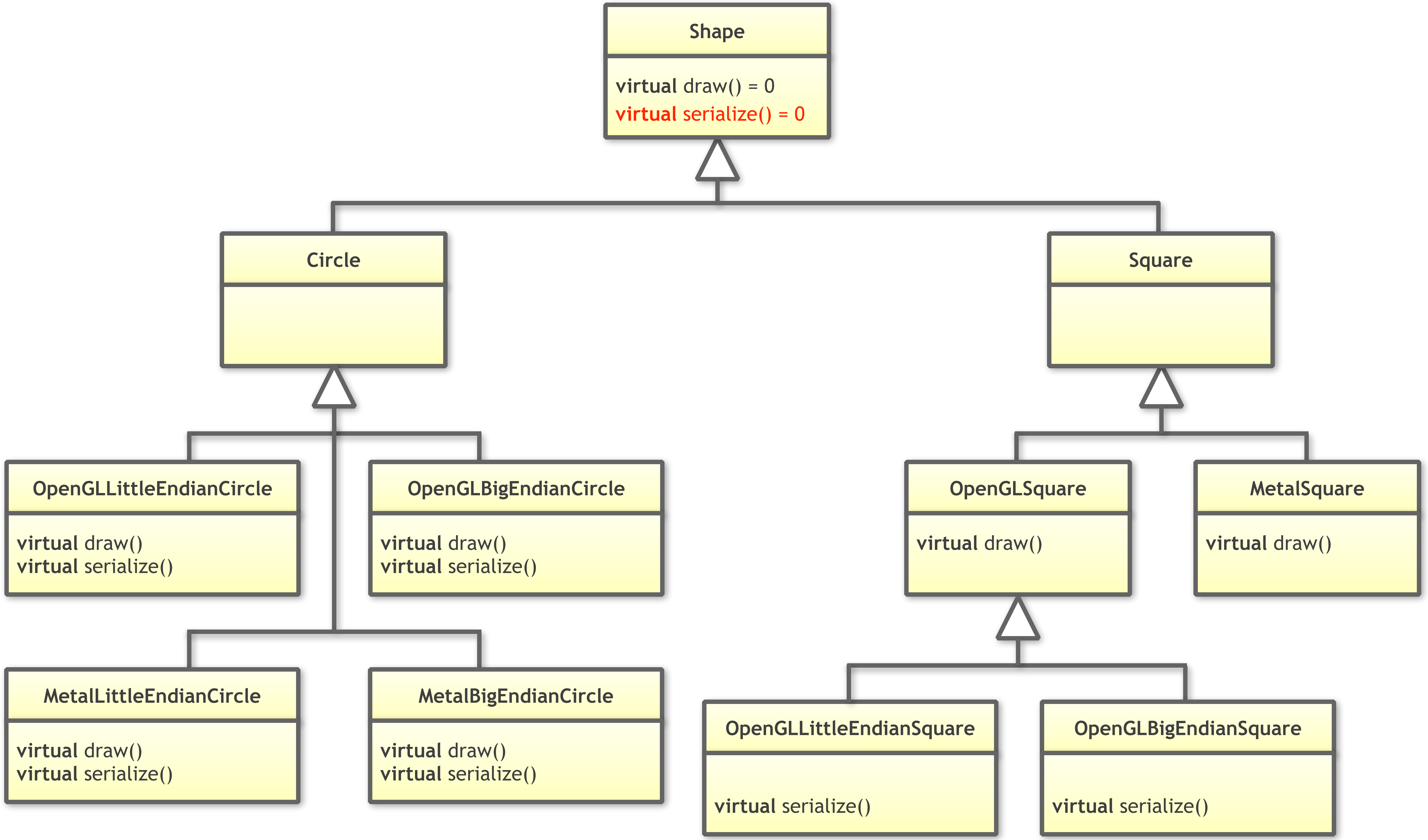
”I’m tired of this example, but I don’t know any better one.”

(Lukas Bergdoll, MUC++ organizer)

Designing the Shape Hierarchy



Designing the Shape Hierarchy



Designing the Shape Hierarchy

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

Using inheritance naively to solve our problem easily leads to …

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

Designing the Shape Hierarchy

Guideline: Resist the urge to put everything into one class. Separate concerns!

Guideline: If you use OO programming, use it properly.

Designing the Shape Hierarchy

Guideline: Design classes for easy change.

Guideline: Design classes for easy extensions.

Designing the Shape Hierarchy

”Inheritance is Rarely the Answer.

Delegate to Services: Has-A Trumps Is-A.”

(Andrew Hunt, David Thomas, The Pragmatic Programmer)

The Solution: Design Principles and Patterns

The Solution: Design Principles and Patterns

Single-Responsibility Principle (SRP)

Open-Closed Principle (OCP)

Don’t Repeat Yourself (DRY)

The Single-Responsibility Principle (SRP)

“Everything should do just one thing.”

(Common Knowledge?)

The Single-Responsibility Principle (SRP)

“The Single-Responsibility Principle advices to separate concerns to **isolate and simplify change**.”

(Klaus Iglberger)

The SRP is also known as

- Separation of Concerns
- High cohesion / low coupling
- Orthogonality

The Open-Closed Principle (OCP)

“The Open-Closed Principle advices to prefer design that **simplifies the extension** by types or operations.”

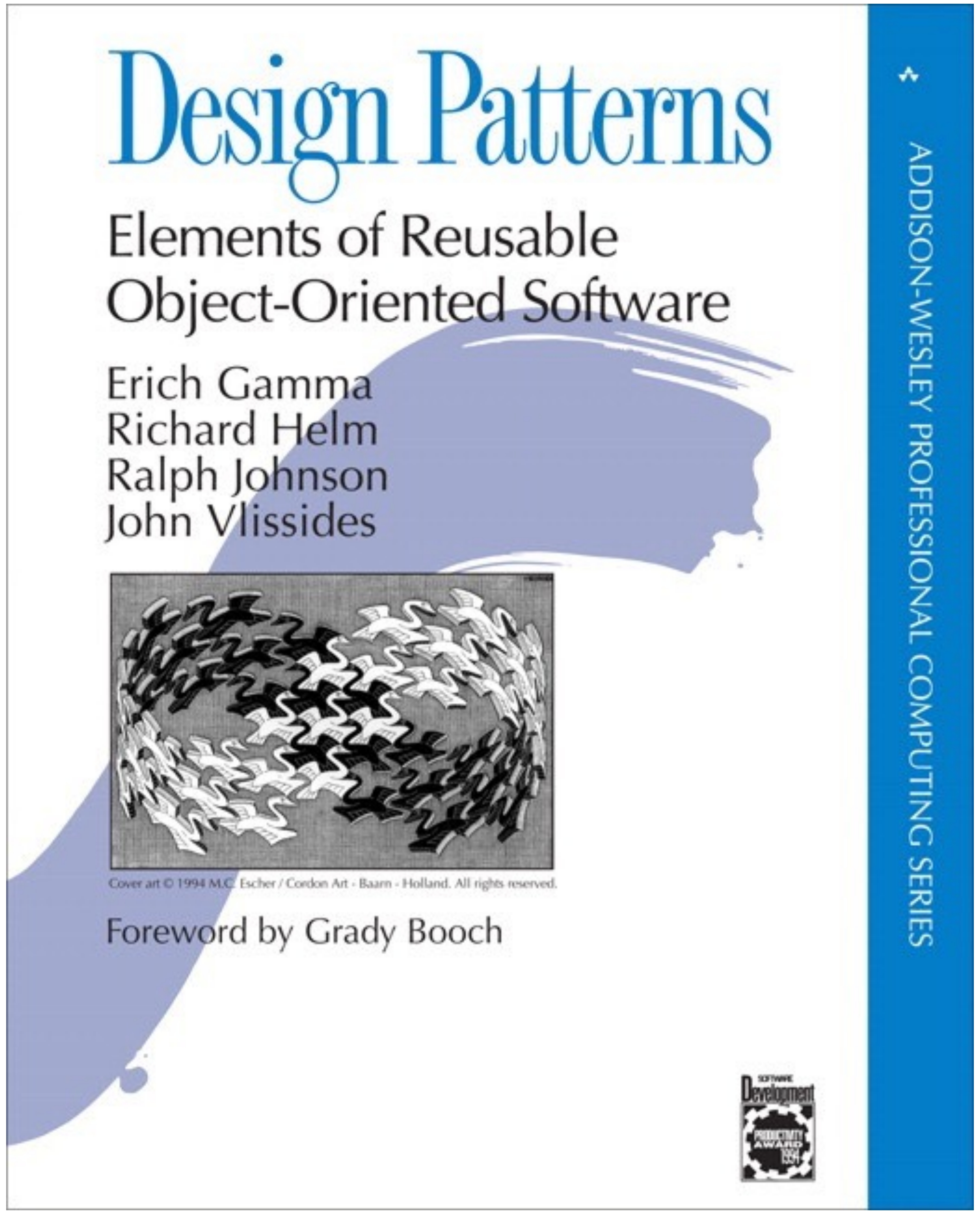
(Klaus Iglberger)

Don’t Repeat Yourself (DRY)

“The DRY Principle advices to reduce duplication in order to **simplify change**.”

(Klaus Iglberger)

The Solution: Design Principles and Patterns


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES
Software Development Productivity Award

The **Gang-of-Four** (**GoF**) book: Origin of 23 of the most commonly used design patterns.

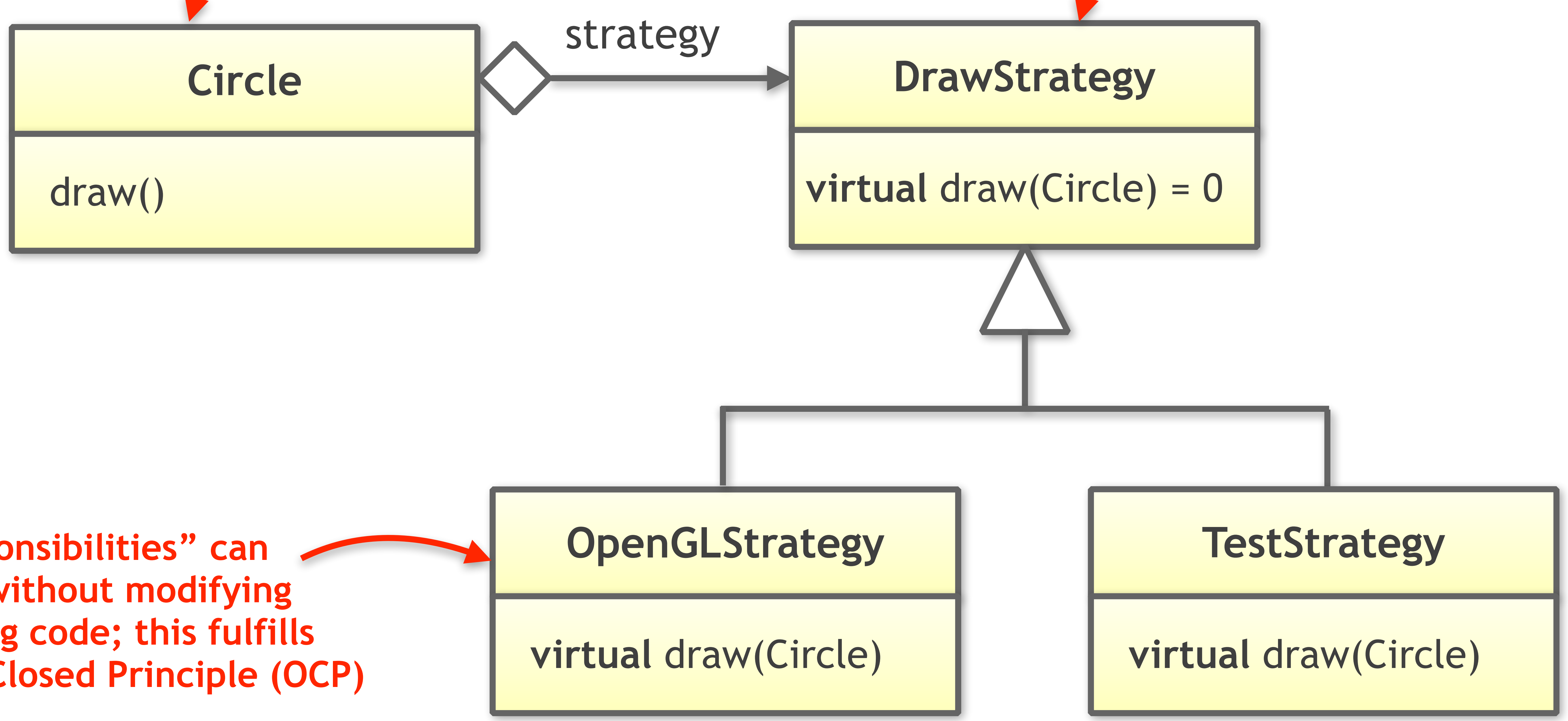
A design pattern …

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

The Strategy Design Pattern

This represents any concrete shape, i.e. Circle, Square, etc., and not a base class

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



Circle

draw()

strategy

DrawStrategy

virtual draw(Circle) = 0

OpenGLStrategy

virtual draw(Circle)

TestStrategy

virtual draw(Circle)

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

A Strategy-Based Solution

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

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


class Circle;

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

   virtual void draw( Circle const& circle, /*...*/ ) const = 0;
};


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

A Strategy-Based Solution

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

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


class Circle;

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

   virtual void draw( Circle const& circle, /*...*/ ) const = 0;
};


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

A Strategy-Based Solution

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

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


class Circle;

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

   virtual void draw( Circle const& circle, /*...*/ ) const = 0;
};


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

A Strategy-Based Solution

```
};


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

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

   void draw( /*...*/ ) const override
   {
      drawing->draw( this, /*...*/ );
   }
   void serialize( /*...*/ ) const override;

   // ...

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


class Square:
```

Dependency Injection

A Strategy-Based Solution

```
   // ...

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


class Square;

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

   virtual void draw( Square const& square, /*...*/ ) const = 0;
};


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

   double getSide() const noexcept;
```

A Strategy-Based Solution

```
};


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

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

   void draw( /*...*/ ) const override
   {
      drawing->draw( this, /*...*/ );
   }
   void serialize( /*...*/ ) const override;

   // ...

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

```
class OpenGLCircleStrategy : public DrawCircleStrategy
```

A Strategy-Based Solution

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



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

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

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

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



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

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

A Strategy-Based Solution

```
class OpenGLSquareStrategy : public DrawSquareStrategy
{
 public:
   virtual ~OpenGLStrategy() {}

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



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

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

A Strategy-Based Solution — Summary

By means of the Strategy design pattern we have …

- … extracted implementation details (SRP);
- … created the opportunity for easy change;
- … created the opportunity for easy extension (OCP);
- … reduced duplication (DRY);
- … limited the depth of the inheritance hierarchy;
- … simplified maintainance.

A Strategy-Based Solution — Guidelines

Guideline: Design classes for easy change.

Guideline: Design classes for easy extensions.

Guideline: Don’t guess! If you expect change, prefer design that makes this change easy. If you don’t expect any change, learn from the next change.

A Strategy-Based Solution

The guidelines make sense, but still you complain …

”That’s the style of the 90s and early 2000s, not Modern C++!”
(You)

And you are correct. Today we favor a value-semantics style…

A Strategy-Based Solution

```
class Circle;

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


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

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

   void draw( /*...*/ ) const override
   {
      drawing( this, /*...*/ );
   }
   void serialize( /*...*/ ) const override;

   // ...

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

A Strategy-Based Solution

```
template< typename DrawStrategy >
class Circle : public Shape
{
 public:
   Circle( double rad )
      : radius{ rad }
      , // ... Remaining data members
   {}

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

   void draw( /*...*/ ) const override
   {
      DrawStrategy{}( this, /*...*/ );
   }
   void serialize( /*...*/ ) const override;

   // ...

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

It's still the same intent: separation of concerns (SRP)

A Strategy-Based Solution — Guidelines

Guideline: Design classes for easy change.

Guideline: Design classes for easy extensions.

Our Second Toy Problem: Persistence Systems

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

   virtual ~PersistenceInterface();

   virtual bool write( const Blob& blob ) = 0;
   virtual bool write( const Blob& blob, WriteCallback callback ) = 0;
   virtual bool read ( Blob& blob, uint timeout ) = 0;
   virtual bool read ( Blob& blob, ReadCallback callback, uint timeout ) = 0;
   // ...
};
```

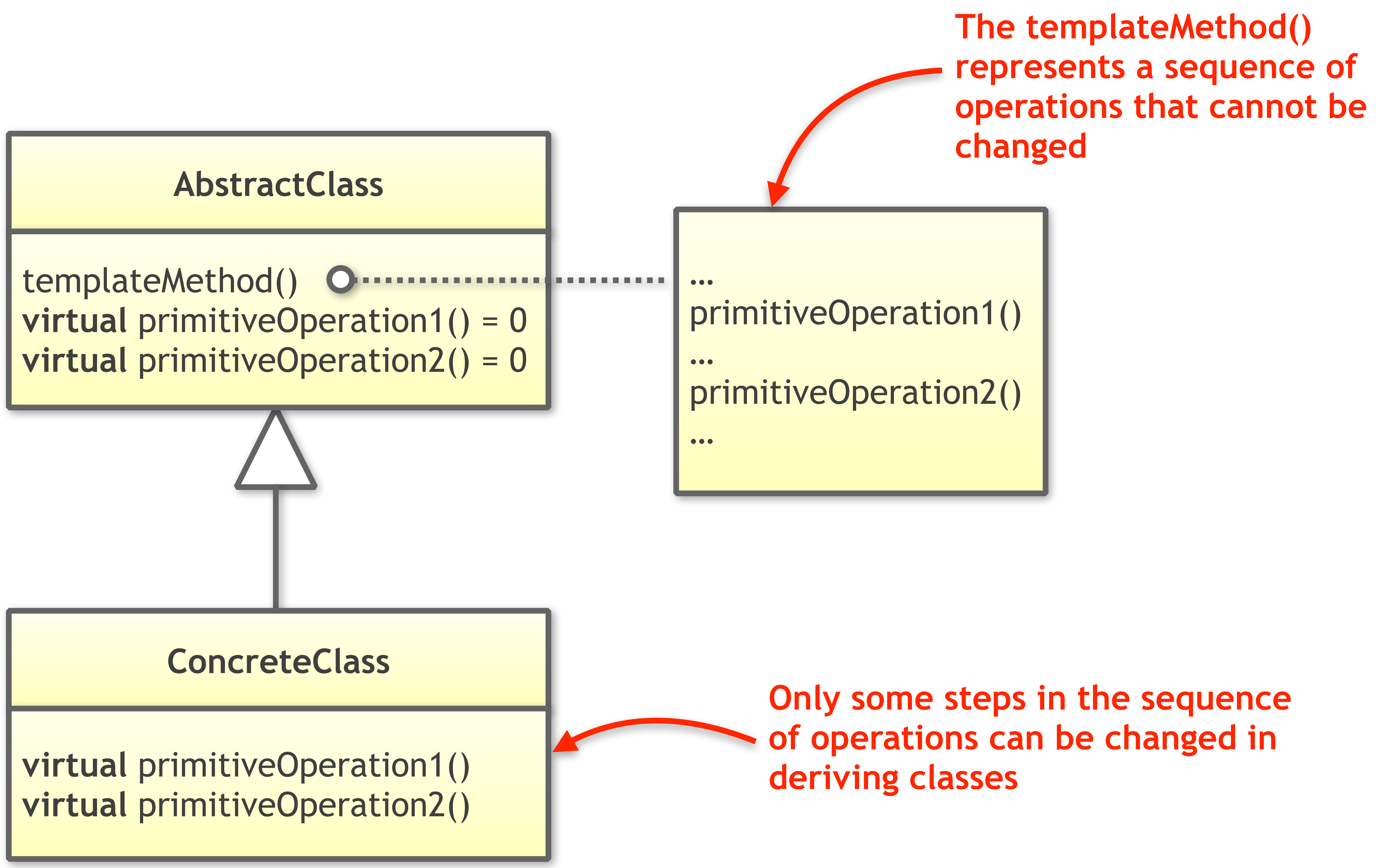
Our Second Toy Problem: Persistence Systems

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

   virtual ~PersistenceInterface();

   virtual bool write( const Blob& blob ) = 0;
   virtual bool write( const Blob& blob, WriteCallback callback ) = 0;
   virtual bool read ( Blob& blob, uint timeout ) = 0;
   virtual bool read ( Blob& blob, ReadCallback callback, uint timeout ) = 0;
   // ...
};
```

The virtual functions may pose a problem in the future …

- … because they **represent the interface to callers**;
- … because they **represent the interface for deriving classes**;
- … **don’t separate concerns**;
- … potentially **introduces a lot of duplication**;
- … **make changes harder** (and sometimes impossible).

The Template Method Design Pattern


AbstractClass
templateMethod()
virtual primitiveOperation1() = 0
virtual primitiveOperation2() = 0
...
primitiveOperation1()
...
primitiveOperation2()
...
ConcreteClass
virtual primitiveOperation1()
virtual primitiveOperation2()

The templateMethod() represents a sequence of operations that cannot be changed

Only some steps in the sequence of operations can be changed in deriving classes

The Template Method-Based Solution

```
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;
   // ...
};
```

No virtual function in the public interface (except for the destructor).

In C++ we call this the Non-Virtual Interface Idiom (NVI)

The Template Method-Based Solution

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

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

    const auto start = std::chrono::high_resolution_clock()::now();
    const bool success = doWrite( blob );
    const uint32_t time = std::chrono::high_resolution_clock::now() - start;

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

    return success;
}
```

The Template Method-Based Solution

```
class PersistenceInterface
{
public:
   PersistenceInterface();
   virtual ~PersistenceInterface();
   bool write( const Blob& blob );
   // ...

private:

   virtual bool doWrite( const Blob& blob ) = 0;
   // ...
};
```

The Template Method-Based Solution

```
class PersistenceInterface
{
public:
   PersistenceInterface();
   virtual ~PersistenceInterface();
   bool write( const Blob& blob );
   // ...

private:
   virtual bool prepareWrite() = 0;
   virtual bool doWrite( const Blob& blob ) = 0;
   // ...
};
```

By means of the **Non-Virtual Interface Idiom (NVI)** we have ...

- ... separated concerns and **simplified change** (SRP);
- ... enabled internal changes with **no impact on callers;**
- ... reduced duplication (DRY).

A Template Method-Based Solution — Guidelines

Guideline: Design classes for easy change.

Guideline: Design classes for easy extensions.

Design for Change and Extension

Design for Change and Extension

Guideline: Classes should be …

- … concise and focused on one purpose (SRP)
- … developed with extensibility in mind (OCP)
- … split into smaller pieces to favor reuse (DRY)

Design for Testability

Back to Basics: Class Design (Part 1)

- The Challenge of Class Design
- Design Guidelines
  - Design for Readability
  - Design for Change and Extension
  - Design for Testability
- Implementation Guidelines
  - Resource Management

Back to Basics: Class Design (Part 2)

- Implementation Guidelines
  - Data Member Initialization
  - Implicit Conversions
  - Order of Data Members
  - Const Correctness
  - Encapsulating Design Decisions
  - Qualified/Modified Member Data
  - Visibility vs. Accessibility

Design for Testability

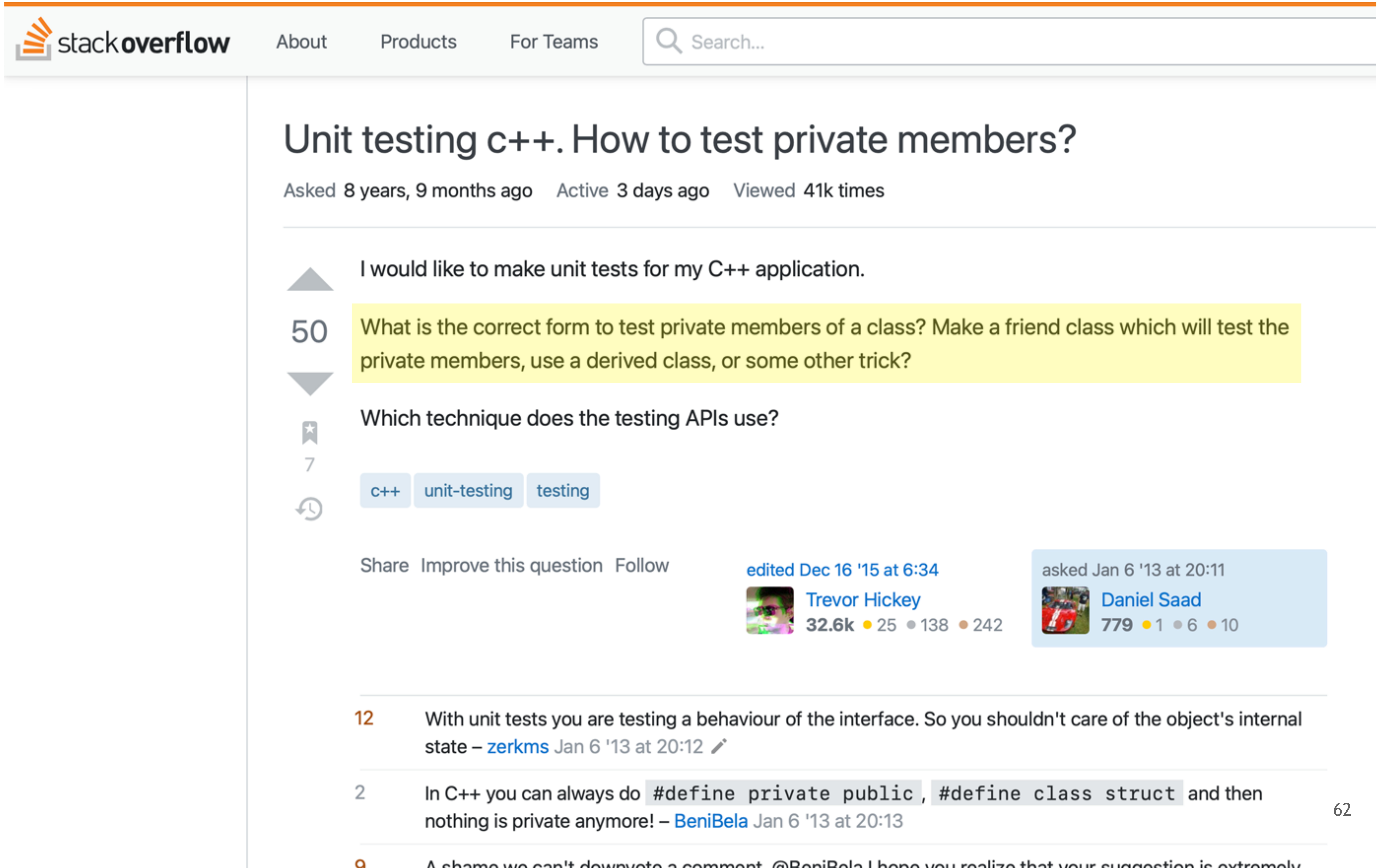
```
template< typename Type, size_t Capacity >
class FixedVector
{
 public:
   // ...

 private:
   // ...

   void destroy( Type* first, Type* last )
   {
      for( ; first != last; ++first ) {
         first->~Type();
      }
   }

   size_t size_;
   std::byte raw_[Capacity*sizeof(Type)];
};
```

Note the choice of names!

... but it is private!

You want to test this function
(and not just as part of some public function) ...

Design for Testability



Unit testing c++. How to test private members?

Asked 8 years, 9 months ago Active 3 days ago Viewed 41k times

50

I would like to make unit tests for my C++ application.

What is the correct form to test private members of a class? Make a friend class which will test the private members, use a derived class, or some other trick?

Which technique does the testing APIs use?

7

c++ unit-testing testing

Share Improve this question Follow

edited Dec 16 '15 at 6:34
Trevor Hickey
32.6k ● 25 ● 138 ● 242

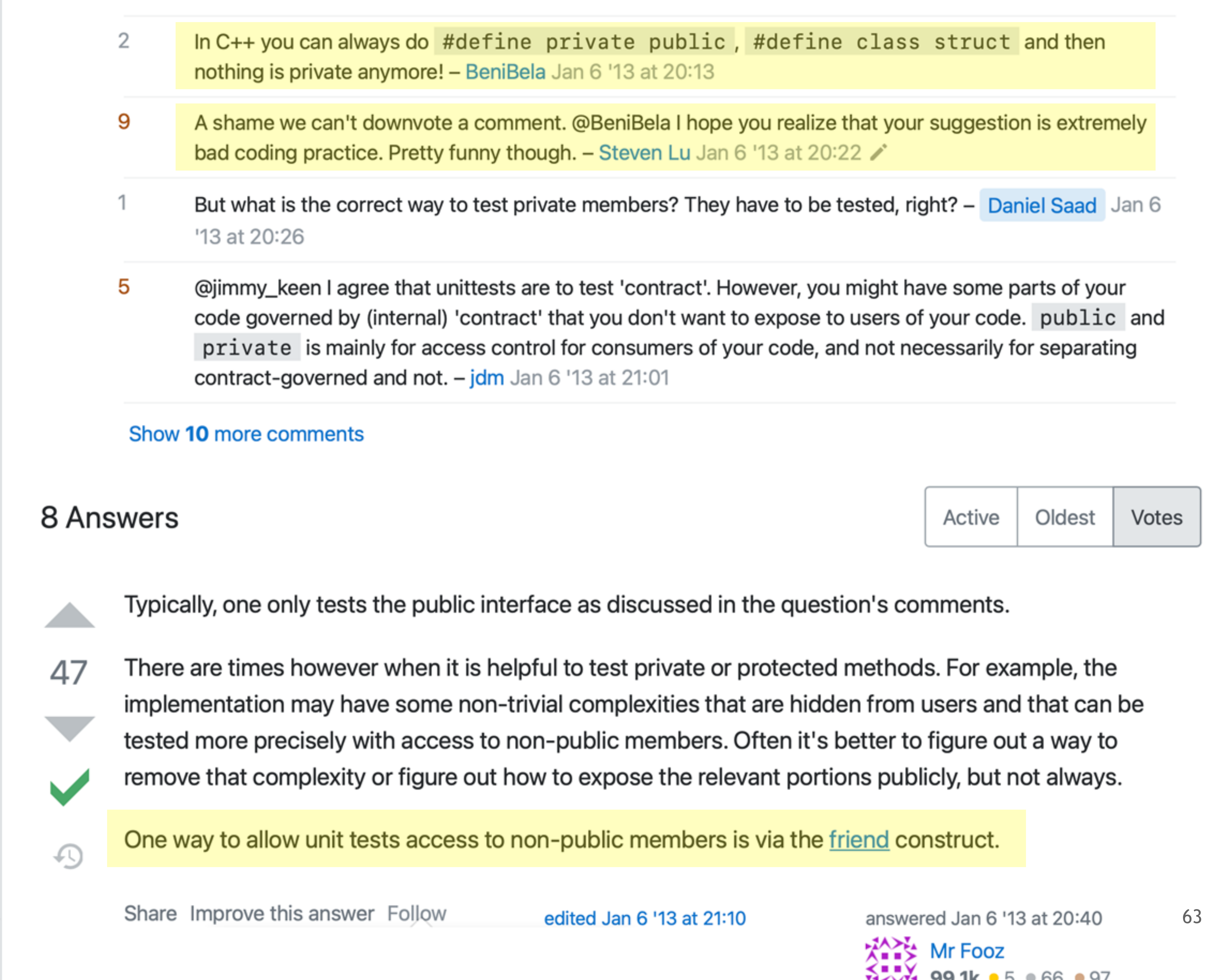
asked Jan 6 '13 at 20:11
Daniel Saad
779 ● 1 ● 6 ● 10

12 With unit tests you are testing a behaviour of the interface. So you shouldn't care of the object's internal state – zerkms Jan 6 '13 at 20:12

2 In C++ you can always do `#define private public`, `#define class struct` and then nothing is private anymore! – BeniBela Jan 6 '13 at 20:13

9 A shame we can't downvote a comment. @BeniBela I hope you realize that your suggestion is extremely

Design for Testability



2 In C++ you can always do `#define private public`, `#define class struct` and then nothing is private anymore! – BeniBela Jan 6 '13 at 20:13

9 A shame we can't downvote a comment. @BeniBela I hope you realize that your suggestion is extremely bad coding practice. Pretty funny though. – Steven Lu Jan 6 '13 at 20:22

1 But what is the correct way to test private members? They have to be tested, right? – Daniel Saad Jan 6 '13 at 20:26

5 @jimmy_keen I agree that unittests are to test 'contract'. However, you might have some parts of your code governed by (internal) 'contract' that you don't want to expose to users of your code. `public` and `private` is mainly for access control for consumers of your code, and not necessarily for separating contract-governed and not. – jdm Jan 6 '13 at 21:01

Show **10** more comments

8 Answers

47

Typically, one only tests the public interface as discussed in the question's comments.

There are times however when it is helpful to test private or protected methods. For example, the implementation may have some non-trivial complexities that are hidden from users and that can be tested more precisely with access to non-public members. Often it's better to figure out a way to remove that complexity or figure out how to expose the relevant portions publicly, but not always.

One way to allow unit tests access to non-public members is via the friend construct.

edited Jan 6 '13 at 21:10

answered Jan 6 '13 at 20:40

Mr Fooz

99.1k ● 5 ● 66 ● 97

Design for Testability

The choices to test private members:

- `#define private public` 😱
- Make the test a `friend` 😒
- Make the member `public` 🤨
- Derive the test class from the tested class 🥴
- Separate concerns 😍
  - Move the member into a private namespace …
  - … or into another class (as a separate service).

Design for Testability

This is the design favored by the C++ standard library:

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


template< class ForwardIt >
constexpr void destroy( ForwardIt first, ForwardIt last );
```

Design for Testability

Guideline: Resist the urge to put everything into **one** class.

Guideline: Design classes to be testable.

Implementation Guidelines

Back to Basics: Class Design (Part 1)

- The Challenge of Class Design
- Design Guidelines
  - Design for Readability
  - Design for Change and Extension
  - Design for Testability
- Implementation Guidelines
  - Resource Management

Back to Basics: Class Design (Part 2)

- Implementation Guidelines
  - Data Member Initialization
  - Implicit Conversions
  - Order of Data Members
  - Const Correctness
  - Encapsulating Design Decisions
  - Qualified/Modified Member Data
  - Visibility vs. Accessibility

Resource Management

Back to Basics: Class Design (Part 1)

- The Challenge of Class Design
- Design Guidelines
  - Design for Readability
  - Design for Change and Extension
  - Design for Testability
- Implementation Guidelines
  - Resource Management

Back to Basics: Class Design (Part 2)

- Implementation Guidelines
  - Data Member Initialization
  - Implicit Conversions
  - Order of Data Members
  - Const Correctness
  - Encapsulating Design Decisions
  - Qualified/Modified Member Data
  - Visibility vs. Accessibility

Resource Management

```
class Widget
{
 public:
   // ...

   Widget( );                                    // Default constructor

   Widget( Widget const& other );                // Copy constructor

   Widget& operator=( Widget const& other );     // Copy assignment operator

   Widget( Widget&& other ) noexcept;            // Move constructor

   Widget& operator=( Widget&& other ) noexcept; // Move assignment operator

   ~Widget();                                    // Destructor
   // ...

 private:



};
```

Resource Management

```
class Widget
{
 public:
   // ...
   Widget( );                                    // Default constructor

   Widget( Widget const& other );                // Copy constructor

   Widget& operator=( Widget const& other );     // Copy assignment operator

   Widget( Widget&& other ) noexcept;            // Move constructor

   Widget& operator=( Widget&& other ) noexcept; // Move assignment operator

   ~Widget();                                    // Destructor
   // ...

 private:
   int i;               // - i as a representative of a fundamental type
   std::string s;       // - s as a representative of a class (user-defined) type

};
```

Resource Management

```
class Widget
{
 public:
   // ...
```

Core Guideline C.20: If you can avoid defining default operations, do

The Rule of 0

```
   // ...

 private:
   int i;
   std::string s;

};
```

Resource Management

```
class Widget
{
 public:
   // ...
```

Core Guideline C.32: If a class has a raw pointer (T*) or reference (T&), consider whether it might be owning

Core Guideline C.33: If a class has an owning pointer member, define a destructor

```
   ~Widget() { delete pr; }

   // ...

 private:
   int i;
   std::string s;
   Resource* pr;     // - pr as representative of a possible resource
};
```

Resource Management

```
class Widget
{
 public:
   // ...

 private:
   int i;
   std::string s;
   Resource* pr;      // - pr as representative of a possible resource
};
```

Core Guideline C.32: If a class has a raw pointer (T*) or reference (T&), consider whether it might be owning

Core Guideline C.33: If a class has an owning pointer member, define a destructor

Core Guideline R.3: A raw pointer (a T*) is non-owning

Resource Management

```
class Widget
{
 public:
   // ...
```

Core Guideline C.32: If a class has a raw pointer (T*) or reference (T&), consider whether it might be owning

Core Guideline C.33: If a class has an owning pointer member, define a destructor

Core Guideline R.3: A raw pointer (a T*) is non-owning

```
 private:
   int i;
   std::string s;
   std::unique_ptr<Resource> pr;
};
```

Resource Management

```
class Widget
{
 public:
   // ...
```

Core Guideline R.1: Manage resources automatically using resource handles and RAII (Resource Acquisition Is Initialization)

```
   // ...

 private:
   int i;
   std::string s;
   std::unique_ptr<Resource> pr;
};
```

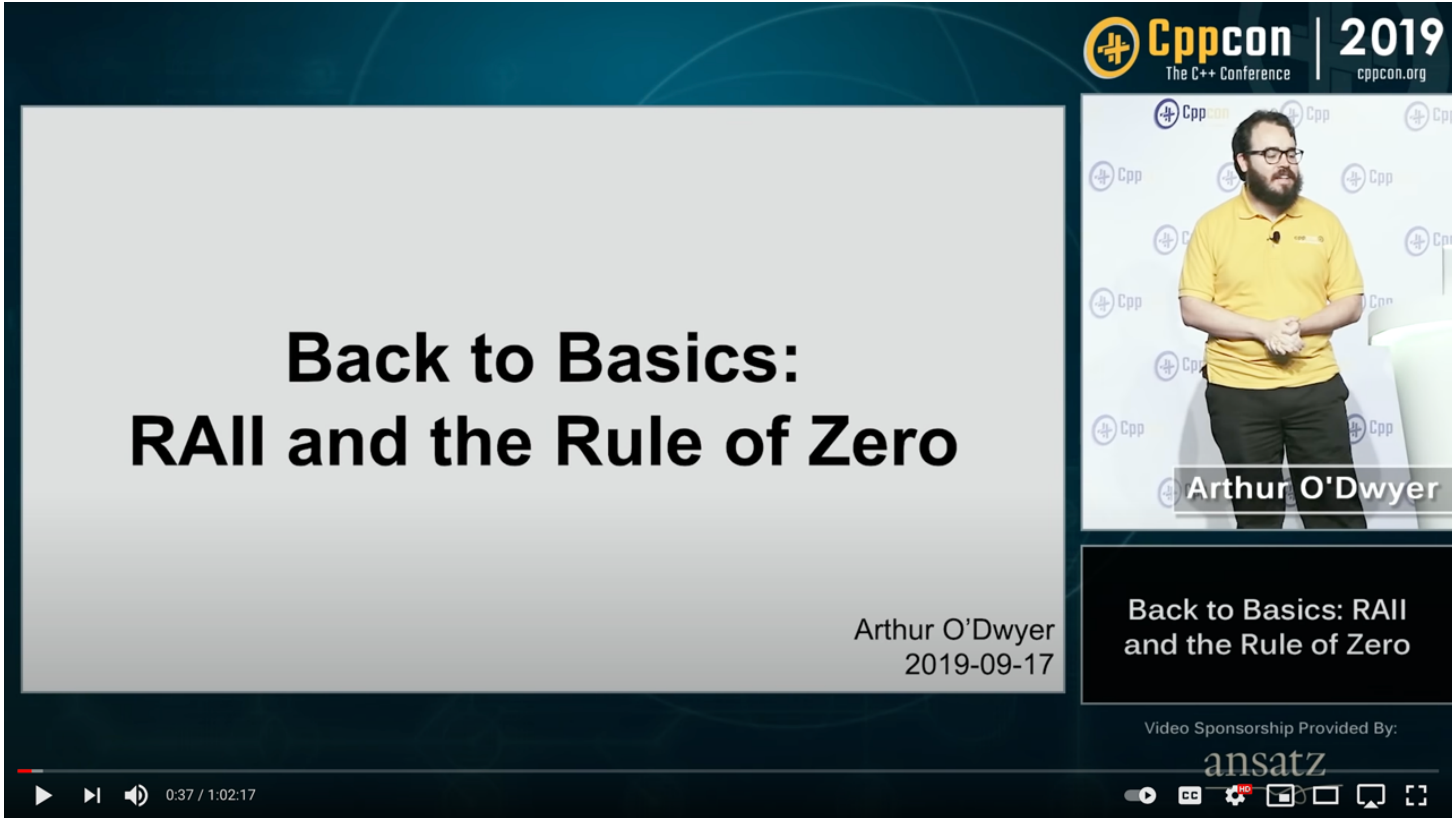
Resource Management

C++’s most important idiom:

RAII

(Resource Acquisition Is Initialization)

Resource Management



Back to Basics:
RAII and the Rule of Zero

Arthur O'Dwyer
2019-09-17

Back to Basics:

Smart Pointers and RAII

INBAL LEVI

21
Cppcon
The C++ Conference
20
21
October 24-29

Thursday, October 28th, 3:15pm MDT

Resource Management

```
class Widget
{
 public:
   // ...
```

Core Guideline R.1: Manage resources automatically using resource handles and RAII (Resource Acquisition Is Initialization)

Guideline: Strive for the **Rule of 0**: Classes that don’t require an explicit destructor, explicit copy operations and explicit move operations are much (!) easier to handle.

```
   // ...

 private:
   int i;
   std::string s;
   std::unique_ptr<Resource> pr;
};
```

std::unique_ptr cannot be copied!

Resource Management

```
class Widget
{
 public:
   // ...






   Widget( Widget const& other );
   Widget& operator=( Widget const& other );
   // Widget( Widget&& other ) noexcept;              // not declared
   // Widget& operator=( Widget&& other ) noexcept;   // not declared

   // ...

 private:
   int i;
   std::string s;
   std::unique_ptr<Resource> pr;
};
```

Resource Management

```
class Widget
{
 public:
   // ...

   Widget( Widget const& other );
   Widget& operator=( Widget const& other );
   Widget( Widget&& other ) noexcept = default;
   Widget& operator=( Widget&& other ) noexcept = default;
   ~Widget() = default;
   // ...

 private:
   int i;
   std::string s;
   std::unique_ptr<Resource> pr;
};
```

Core Guideline C.21: If you define or =`delete` any default operation, define or =`delete` them all

The Rule of 5

Resource Management

```
class Widget
{
 public:
   // ...

   Widget( Widget const& other );
   Widget& operator=( Widget const& other );
   Widget( Widget&& other ) noexcept = default;
   Widget& operator=( Widget&& other ) noexcept = default;
   ~Widget() = default;
   // ...

 private:
   int i;
   std::string s;
   std::shared_ptr<Resource> pr;     // fundamentally changes the semantics!
};
```

Core Guideline C.21: If you define or =delete any default operation, define or =delete them all

The Rule of 5

Resource Management

```
class Widget
{
 public:
   // ...
```

Core Guideline C.21: If you define or `=delete` any default operation, define or `=delete` them all

The Rule of 5

Core Guideline C.20: If you can avoid defining default operations, do

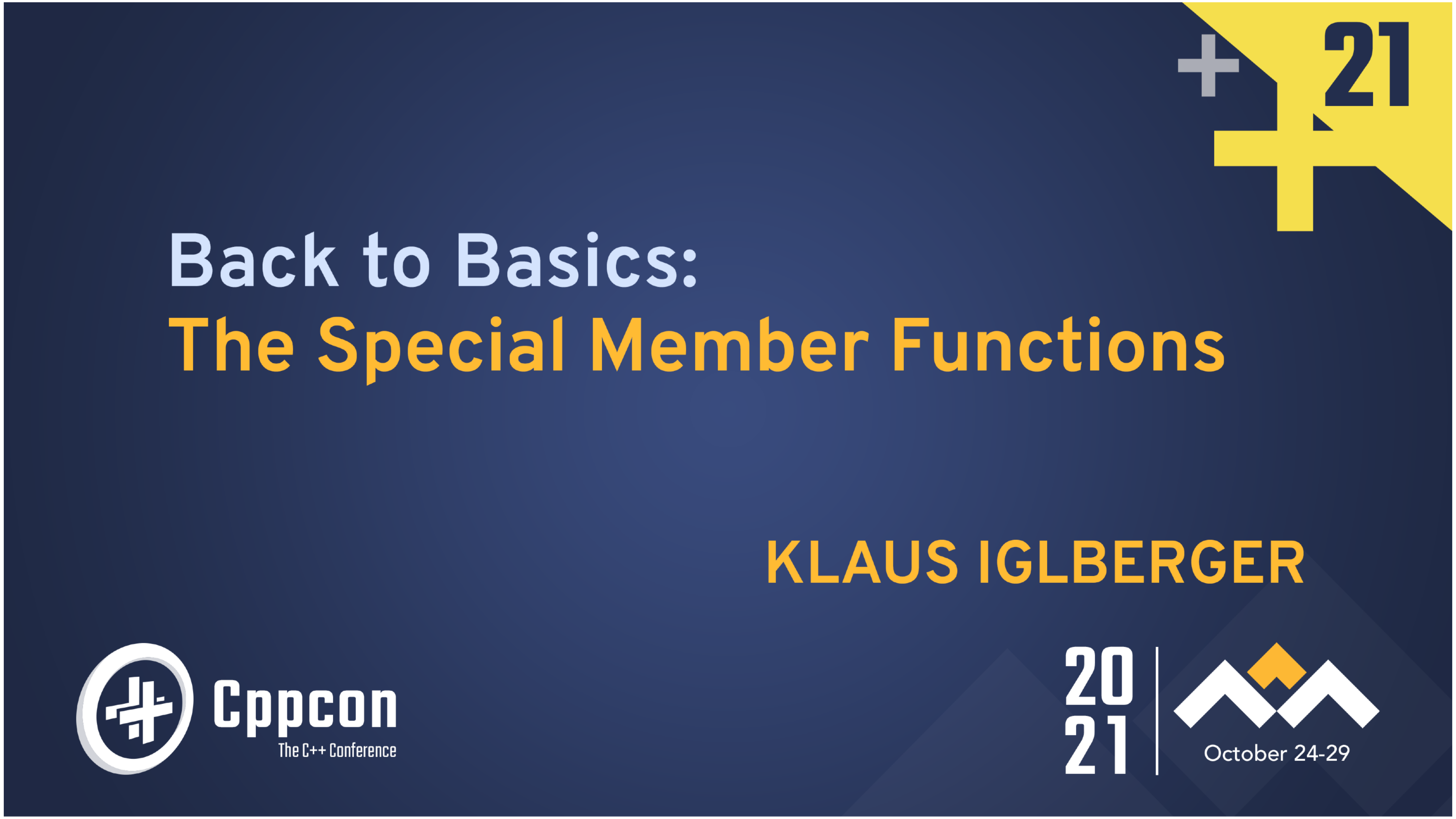
The Rule of 0

```
   // ...

 private:
   int i;
   std::string s;
   std::shared_ptr<Resource> pr;     // fundamentally changes the semantics!
};
```

Resource Management

Guideline: Strive for the **Rule of 0**, but if it cannot be achieved (e.g. because the class implements RAII itself), follow the **Rule of 5**.

Guideline: Design classes for easy change.



Back to Basics:

The Special Member Functions

KLAUS IGLBERGER

21
Cppcon
The C++ Conference
2021
October 24-29

Wednesday, October 27th, 7:45am MDT

Content

Back to Basics: Class Design (Part 1)

- The Challenge of Class Design
- Design Guidelines
  - Design for Readability
  - Design for Change and Extension
  - Design for Testability
- Implementation Guidelines
  - Resource Management

Back to Basics: Class Design (Part 2)

- Implementation Guidelines
  - Data Member Initialization
  - Implicit Conversions
  - Order of Data Members
  - Const Correctness
  - Encapsulating Design Decisions
  - Qualified/Modified Member Data
  - Visibility vs. Accessibility

Back to Basics:
Designing Classes (part 1 of 2)

KLAUS IGLBERGER

Cppcon
The C++ Conference

2021
October 24-29