andel's
HOTEL BERLIN
Breaking Dependencies:
The Path to
High-Quality Software
Klaus Iglberger, Closing Keynote, Meeting C++ 2022
klaus.iglberger@gmx.de

C++ Trainer/Consultant

Author of the blaze C++ math library

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

Chair of the CppCon Back-to-Basics track

Email: klaus.iglberger@gmx.de

Klaus Iglberger

What is
“high-quality software”?

High-quality software is easy to change, easy to extend, and easy to test.

(and of course also fast)

The truth in our industry:

Software must be adaptable to frequent changes

The truth in our industry:

Software must be adaptable to frequent changes

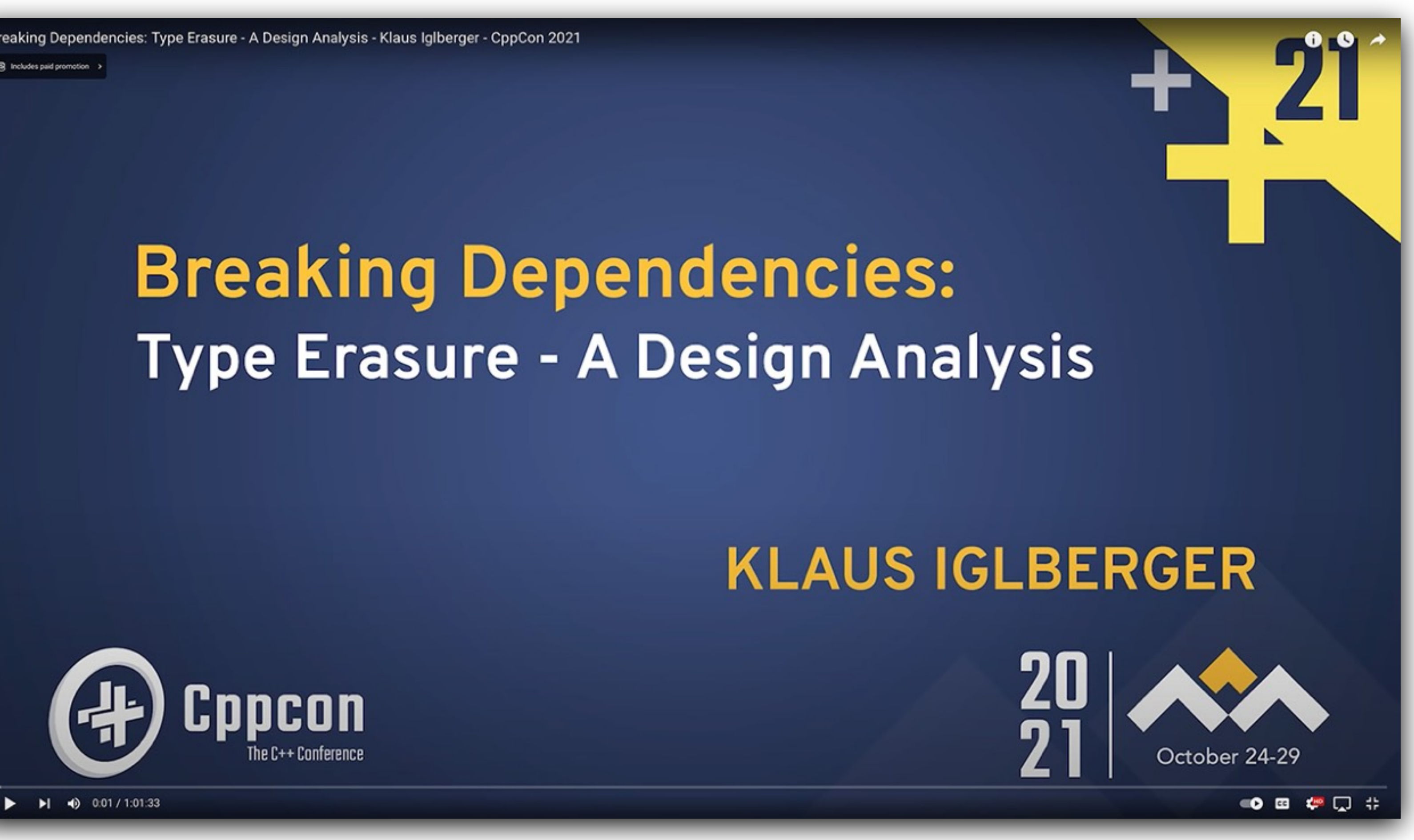
What is the core problem of adaptable software and software development in general?

Dependencies

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

(Kent Beck, TDD by Example)

Breaking Dependencies: The SOLID Principles - Klaus Iglberger - CppCon 2020
Cppcon
The C++ Conference
Breaking Dependencies:
The SOLID Principles
Klaus Iglberger
20
20
September 13-18


Breaking Dependencies: Type Erasure - A Design Analysis - Klaus Iglberger - CppCon 2021
21
Breaking Dependencies:
Type Erasure - A Design Analysis
KLAUS IGLBERGER
Cppcon
The C++ Conference
20
21
October 24-29

Design Patterns: Facts and Misconceptions - Klaus Iglberger - CppCon 2021
21
Design Patterns:
Facts and Misconceptions
KLAUS IGLBERGER
Cppcon
The C++ Conference
20
21
October 24-29

22
Breaking Dependencies
The Visitor Design Pattern
KLAUS IGLBERGER
Cppcon
The C++ Conference
20
22
September 12th-16th

22
Breaking Dependencies:
Type Erasure - The Implementation Details
KLAUS IGLBERGER
Cppcon
The C++ Conference
20
22
September 12th-16th

The common theme:

Breaking Dependencies

The common theme:

Software Design

My Definition of Software Design

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

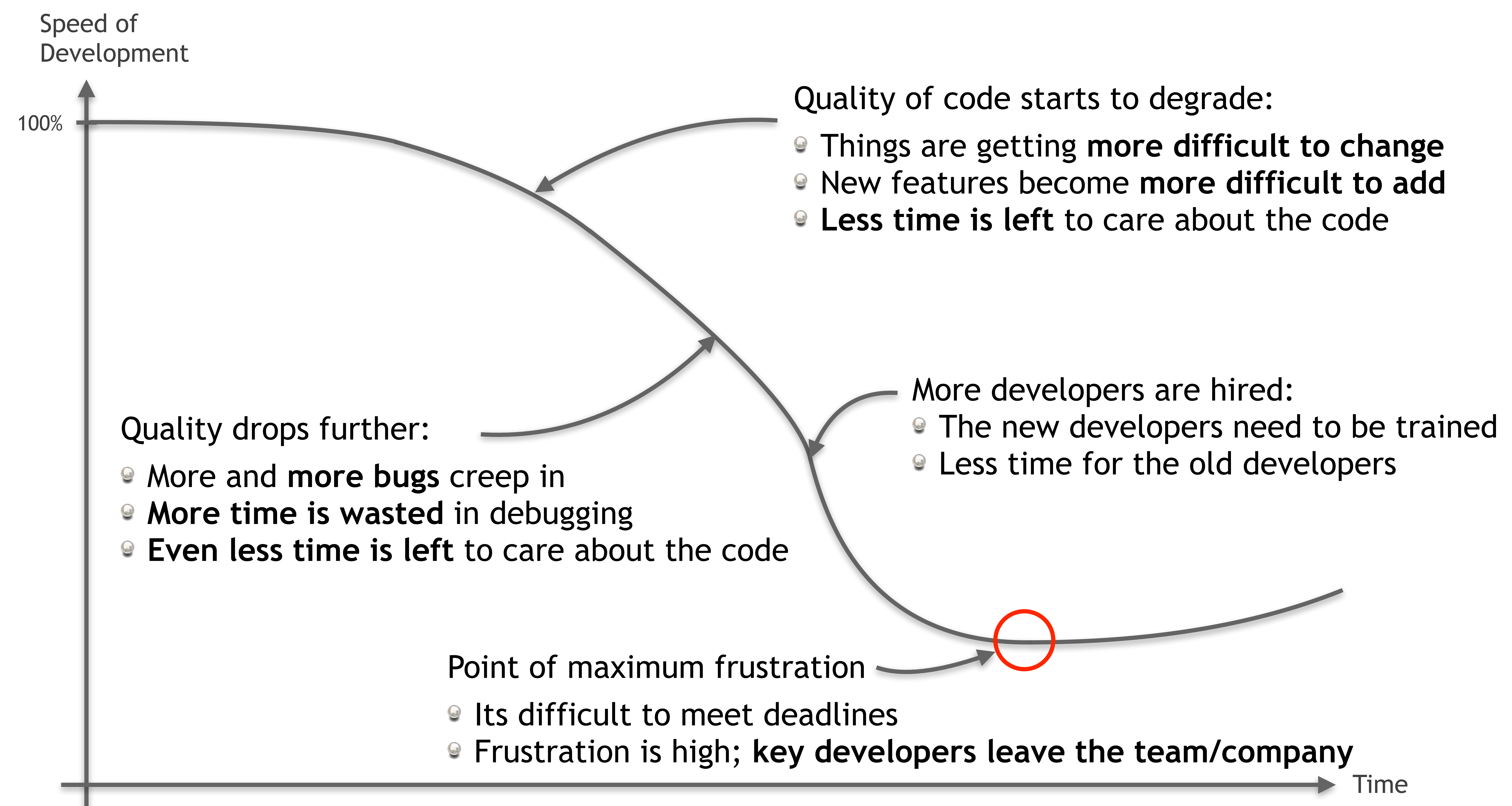
(Klaus Iglberger, C++ Software Design)

My Definition of Software Design

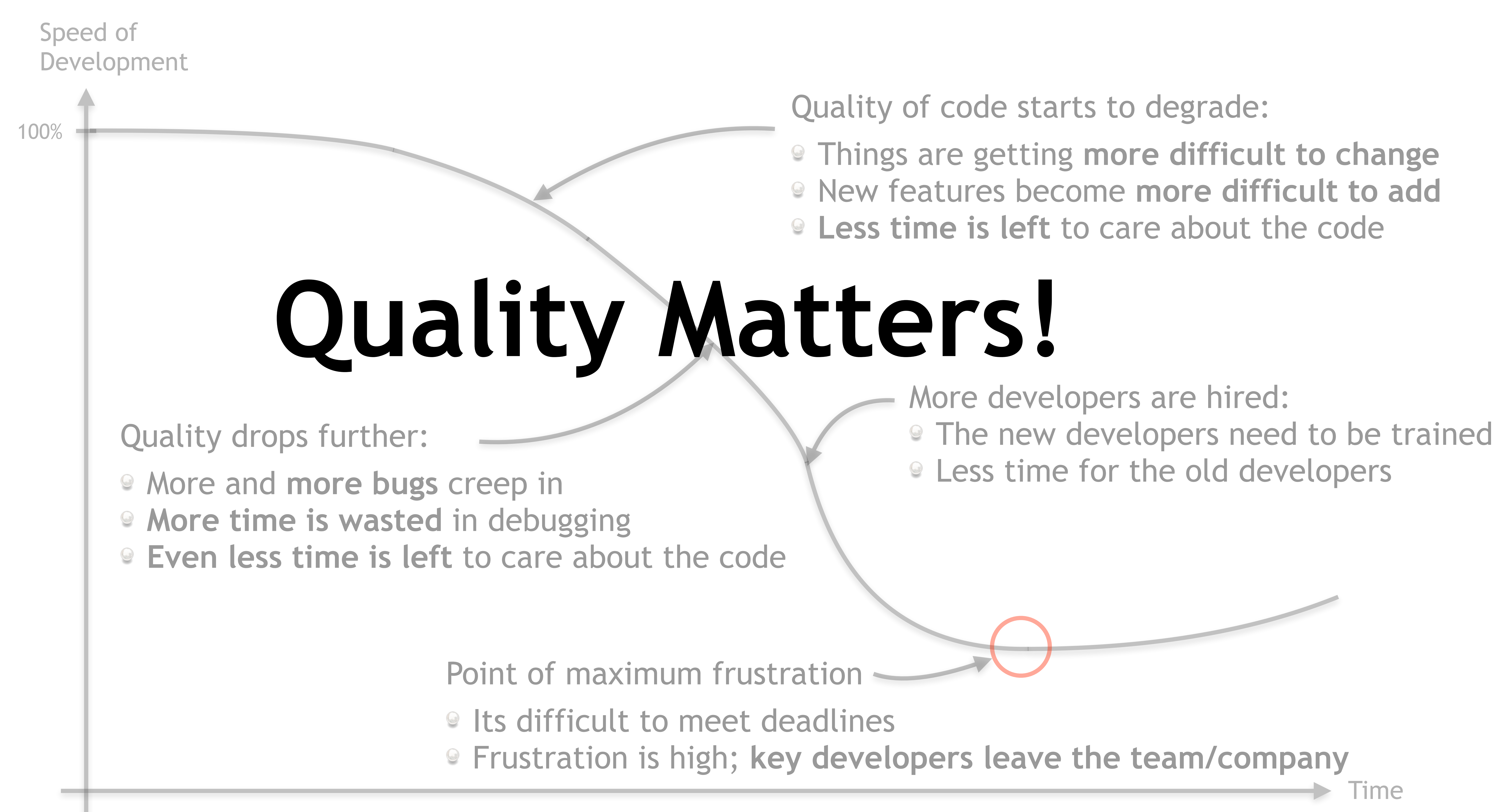
Software Design is the art of managing **dependencies** and **abstractions**.

Ok, but what’s the big deal?

Speed of Development in a Typical Software Project



Speed of Development in a Typical Software Project



Speed of Development in a Typical Software Project

Quality Matters!

- The implementation details matter …
- … but the structure/organization of code (the design) matters even more!

The Role of Architecture/Design

"The design plays a much more central role in the success of a project than any feature could ever do.

Good software is not primarily about the proper use of any feature; rather, it is about solid architecture and design.

Good software design can tolerate some bad implementation decisions, but bad software design cannot be saved by the heroic use of features (old or new) alone."

(Klaus Iglberger, C++ Software Design)

The Reality of Architecture/Design

"But the architecture this [...] code hung from was often an afterthought. They were so focused on features that organization went overlooked."

(Robert Nystrom, Game Programming Patterns)

You’ve ignored software structure and organization?
No problem, you can always …

rewrite the code!

The Real Problem

They [made] the **single worst strategic mistake** that any software company can make: They decided to rewrite the code from scratch.

(Joel Spolsky, joelonsoftware.com*)*

Software design matters, from the very beginning, and it is a continuous effort!

This is a messy child’s room

This is an angry child not interested in cleaning up its room!

Software design matters, from the very beginning, and it is a continuous effort!

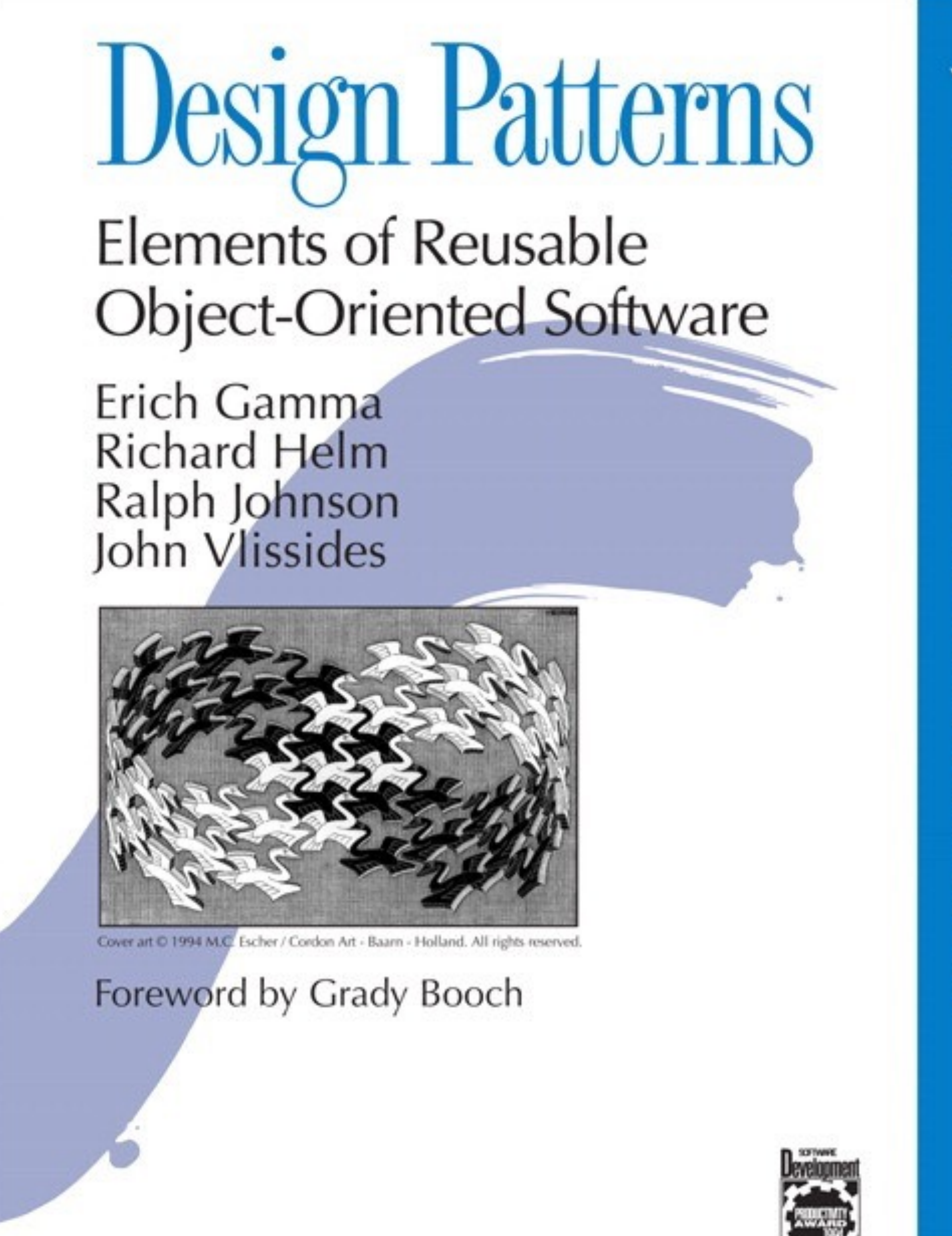
Keep your code easy to change, easy to extend and easy to test.

So how can we keep the software easy to change, extend and test?

The Solution:

Design Patterns

The Solution: Design Patterns



The **Gang-of-Four** (**GoF**) book from 1994: Origin of 23 of the most commonly used design patterns.

This book collects design patterns and …

- … gives them a **name, …**
- … describes their **intent** and …
- … shows how they help to manage **dependencies.**

The Reality of Design Patterns

“Design patterns are everywhere.”

(Klaus Iglberger, C++ Software Design)

Unfortunately we don’t talk enough about software design and design patterns 😢

Meeting C++ 2022 in Numbers

Meeting C++
2022

Total number of talks:	~47	
Number of talks on C++ Features/Standards:	~16	(34%)
Number of talks on software design:	~4	(8%)

Why do we not talk about software design?

Reason 1:

We already know **everything** about design patterns.

Reason 2:

Design patterns are for OOP,
but OOP is not in favor anymore!

Reason 3:

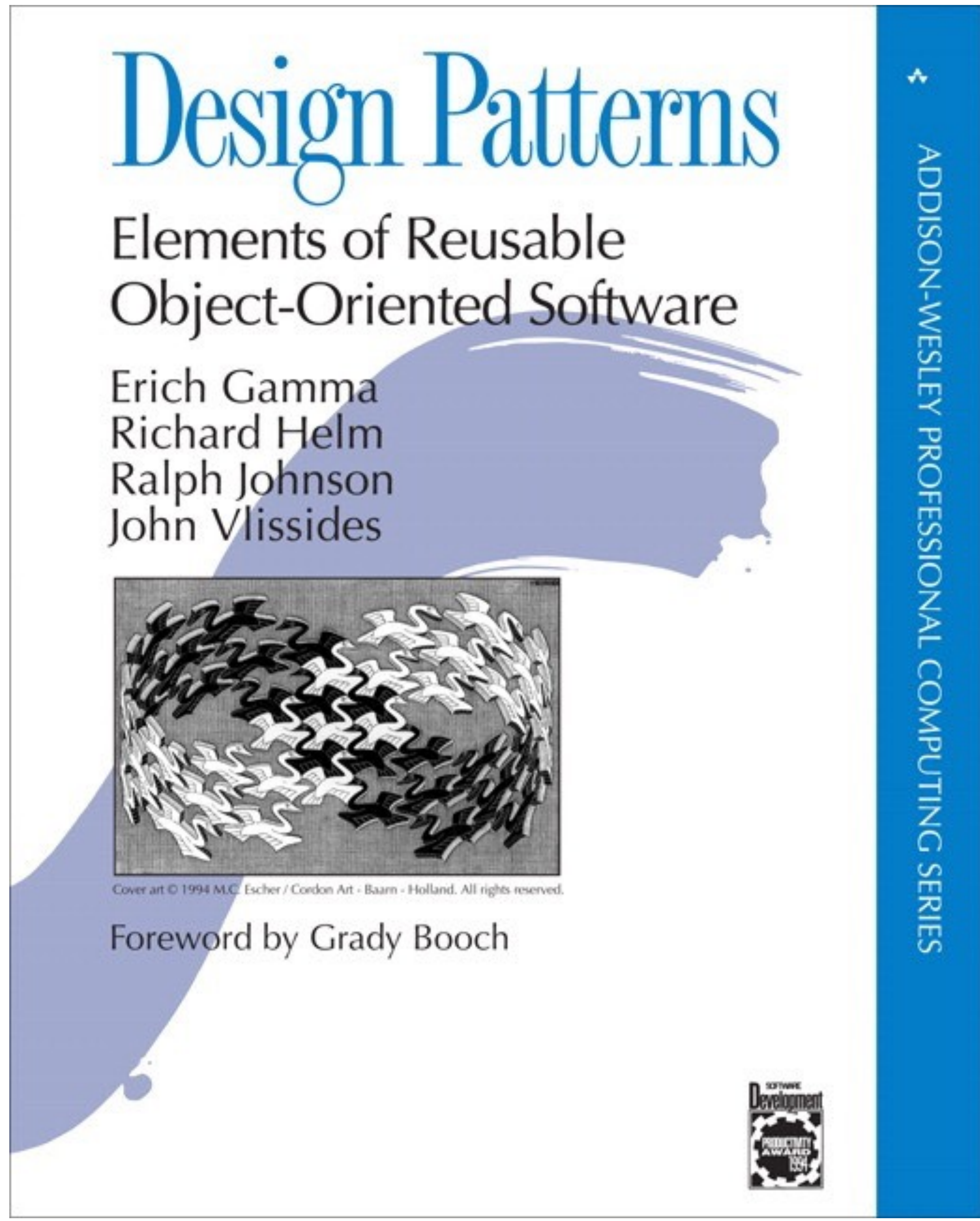
Design patterns are simple!

Reason 1:

We already know **everything** about design patterns.

(After all, the first have been introduced in 1994!)

The Classic Factory Method Design Pattern



The Classic Factory Method Design Pattern

“Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.”

(The Gang of Four, Design Patterns - Elements of Reusable Object-Oriented Software)

The Classic Factory Method Design Pattern

"Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses."

(The Gang of Four, Design Patterns - Elements of Reusable Object-Oriented Software)

Factory Method is often used as synonym for “creating something”.

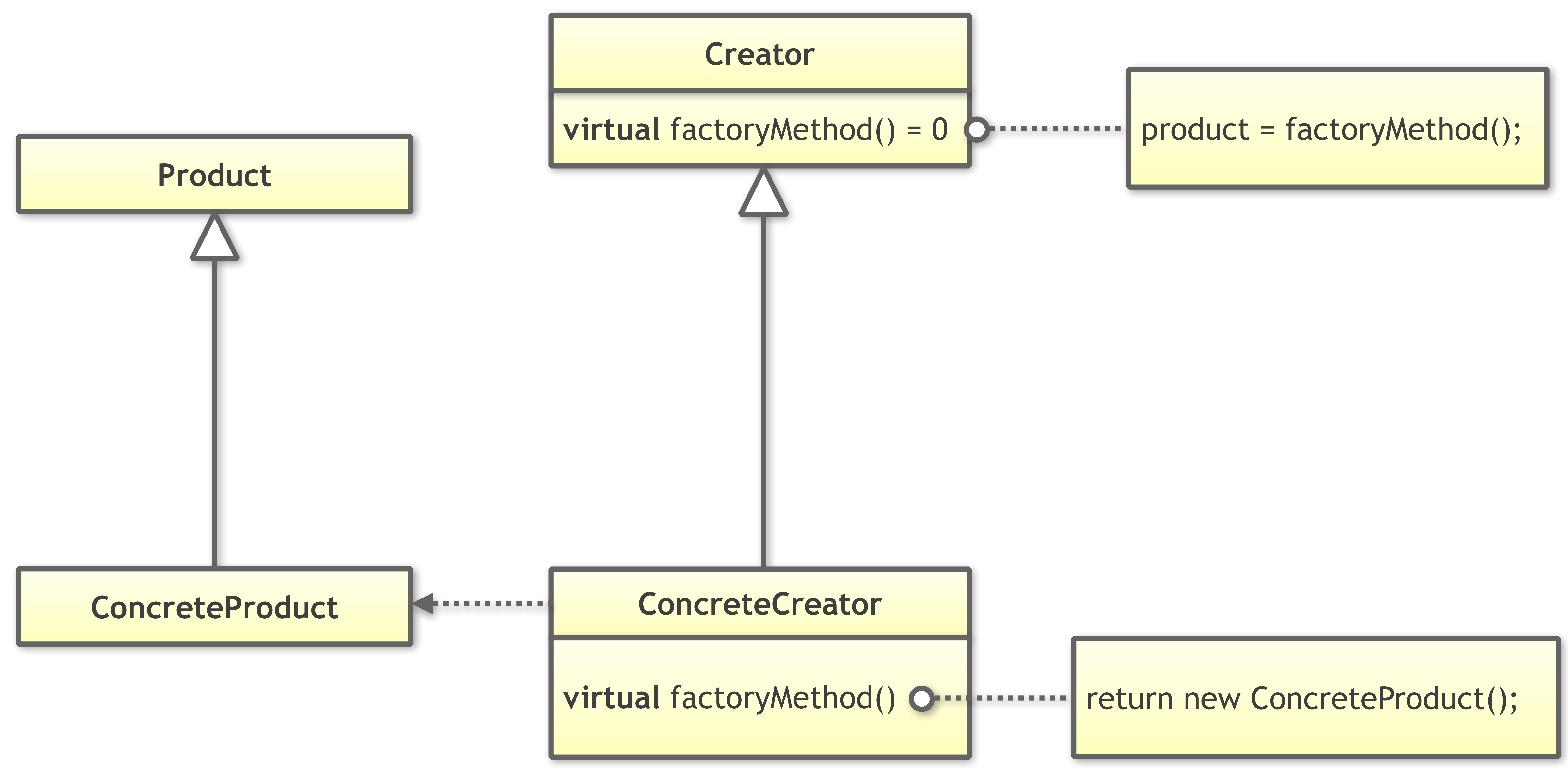
Consequently, `std::make_unique()` is often referred to as a Factory Method.

The Classic Factory Method Design Pattern

"Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses."

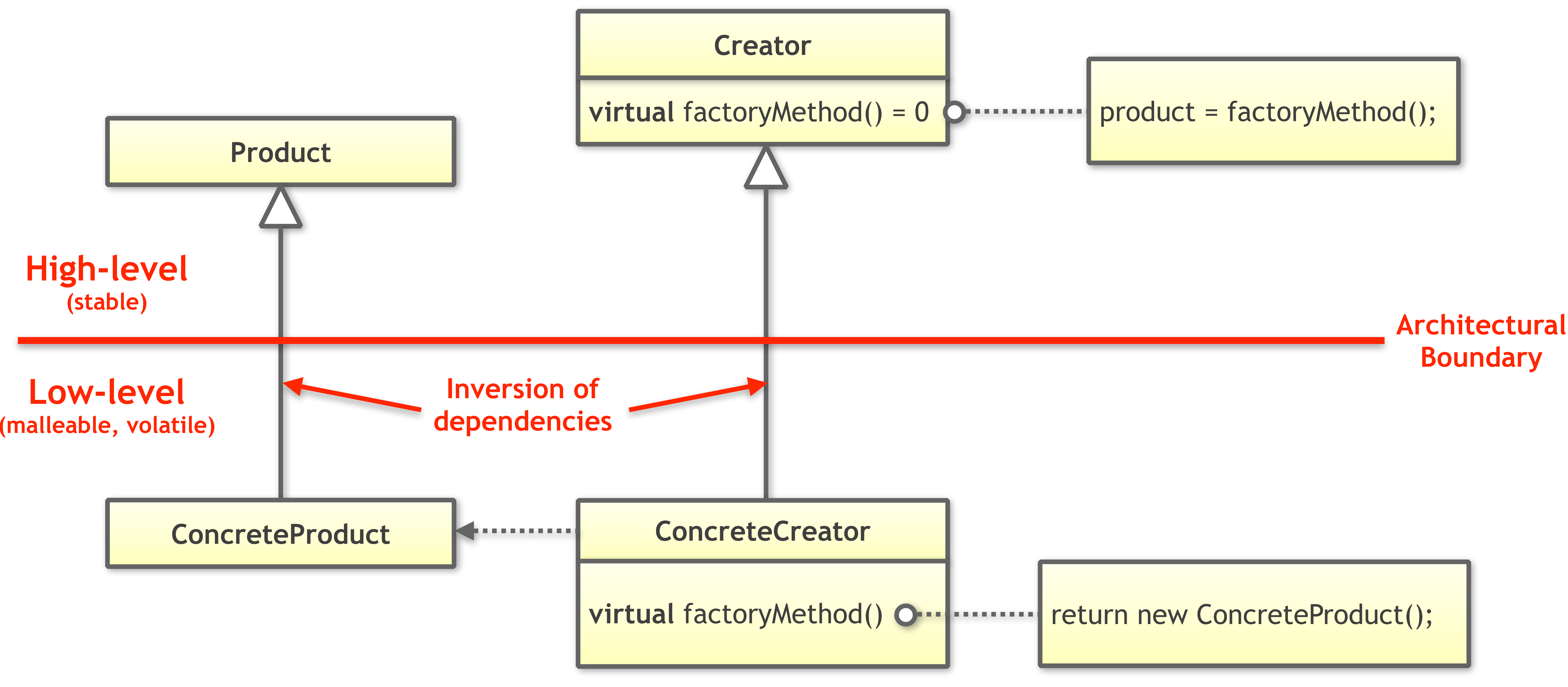
(The Gang of Four, Design Patterns - Elements of Reusable Object-Oriented Software)

The Classic Factory Method Design Pattern



The Classic Factory Method Design Pattern

This is an architecture!



Is std::make_unique() a Factory Method?

`std::make_unique()` does not allow customization.

It does not manage the relationship between entities.

It does not invert dependencies.

It does not help to break dependencies.

And it is not a “method”, but only a function.

From the Index of the Current Draft of the C++ Standard (November 11th 2022)

No entry for "method"

member of the current instantiation, *see* current instantiation, member of the
member pointer to, *see* pointer to member
member subobject, **[intro.object]**
member-declaration, [class.mem.general], [gram.class]
member-declarator, [class.mem.general], [gram.class]
member-declarator-list, [class.mem.general], [gram.class]
member-specification, [class.mem.general], [gram.class]
memory location, **[intro.memory]**
memory management, *see* `new`, *see* `delete`
memory model, [intro.memory]
message
    diagnostic, **[defns.diagnostic]**, [intro.compliance.general]
model
    concept, **[res.on.requirements]**
modifiable, **[basic.lval]**
modification order, **[intro.races]**
module, **[module.unit]**
    exported, **[module.import]**
    global, **[module.unit]**
    named, **[module.unit]**
    reserved name of, [module.unit]

"A virtual function is sometimes called a method."

(Bjarne Stroustrup, The C++ Programming Language, 4th Edition)

"A virtual function is sometimes called a method."

(Bjarne Stroustrup, The C++ Programming Language, 4th Edition)

Is std::make_unique() a Factory Method?

`std::make_unique()` is **not** an example for the **Factory Method** design pattern …

… but it is an example for a **factory function.**

The Power of Terminology

Please use the term **factory function** as a synonym for “creating something”.

Please use the term **Factory Method** as synonym for a customizable **factory function**.

Reason 1:

We already know **everything** about design patterns.

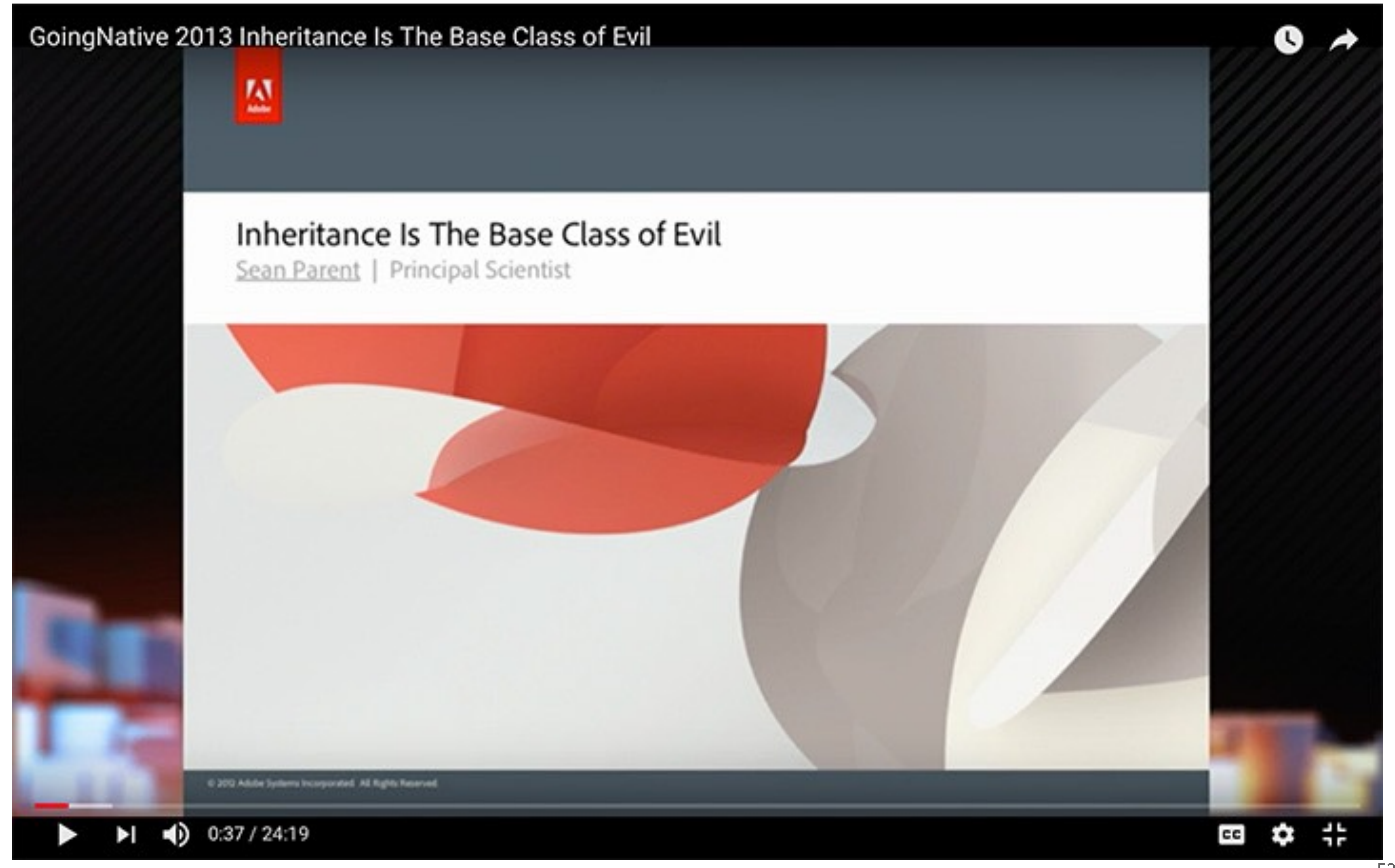
Realization 1:

We already know a lot about design patterns. But we definitely need to talk more about them.

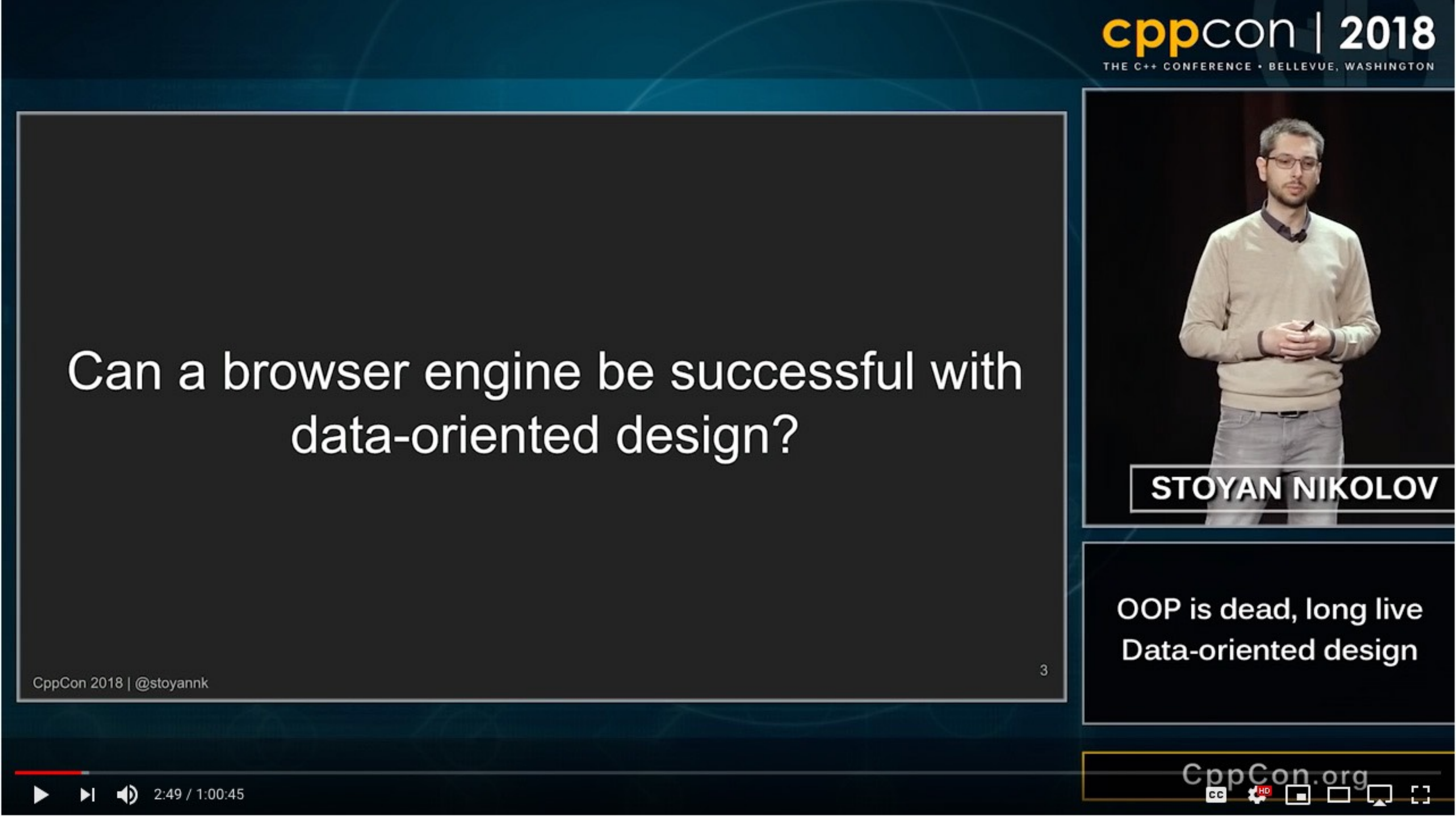
Reason 2:

Design patterns are for OOP, but OOP is not in favor anymore!

The Current Attitude towards OOP



The Current Attitude towards OOP



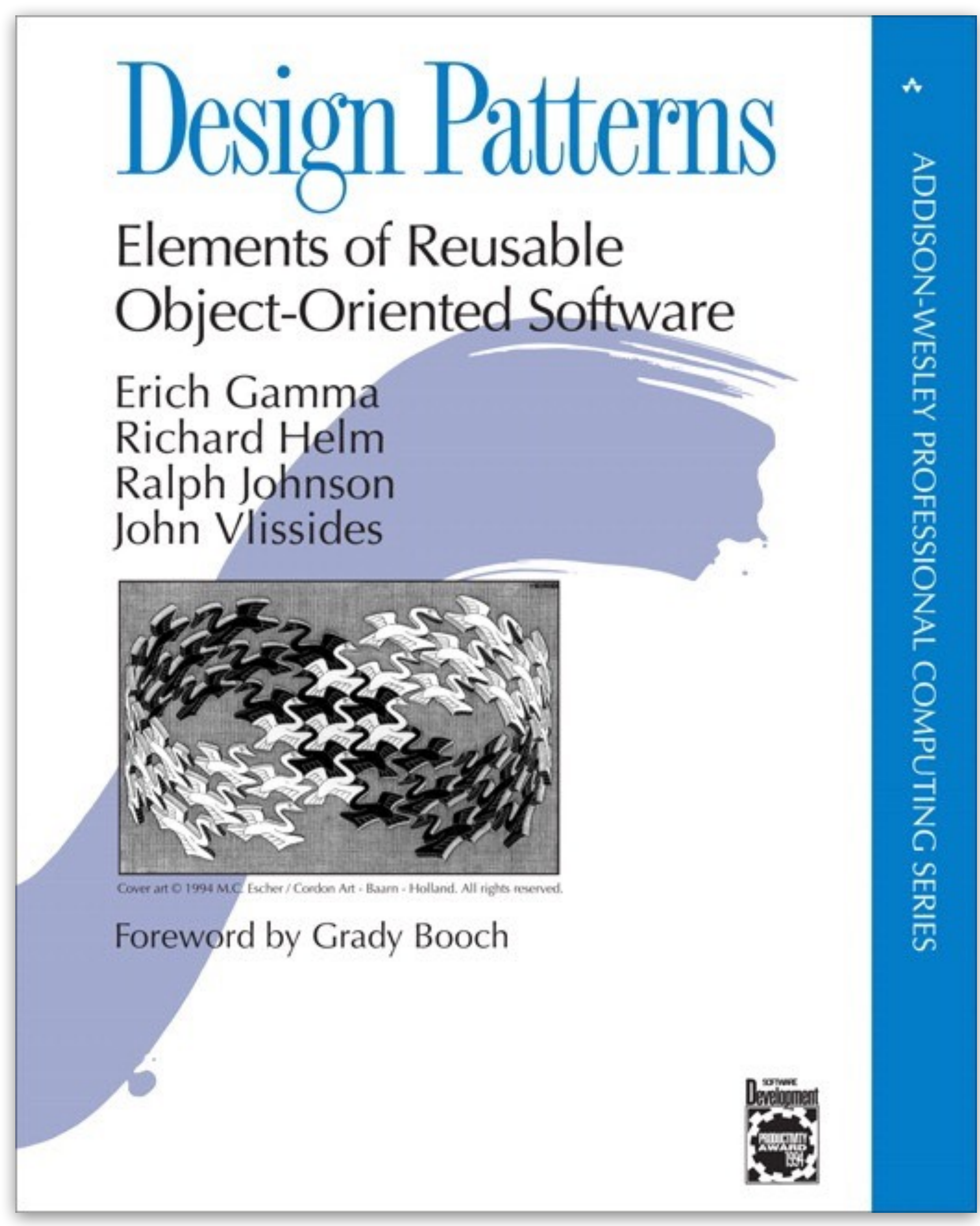
The Common Attitude Toward Design Patterns

The Reality of Design Patterns

"Design patterns are everywhere."

(Klaus Iglberger, C++ Software Design)

The Classic Strategy Design Pattern

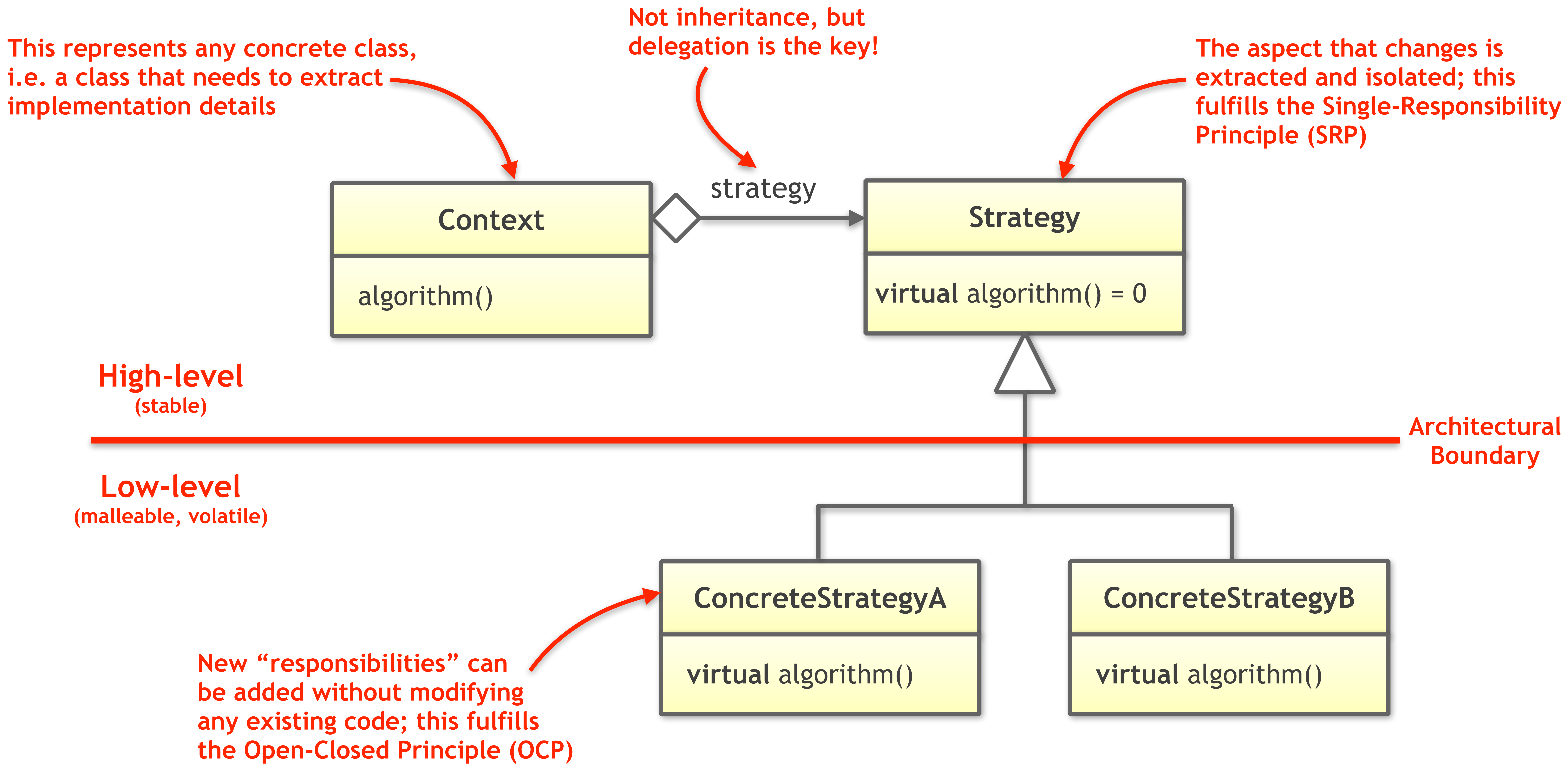


The Classic Strategy Design Pattern

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

(The Gang of Four, Design Patterns - Elements of Reusable Object-Oriented Software)

The Classic Strategy Design Pattern



The classic OO Strategy is often considered
to be THE ONE implementation…

… nothing could be farther from the truth!

The C++ Standard Library itself uses hundreds of Strategies …

Examples from the Standard Library

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

Examples from the Standard Library

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

← Strategy

Examples from the Standard Library

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

← **Strategy**

Examples from the Standard Library

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

Strategy

Examples from the Standard Library

```
std::vector<int> numbers{ 1, 2, 3, 4, 5, 6, 7 };

std::accumulate( begin(numbers), end(numbers), int{0}
               , std::plus<>{} );
```

Strategy

The Reality of Design Patterns

In particular Strategy
is everywhere!

"Design patterns are everywhere."

(Klaus Iglberger, C++ Software Design)

Reason 2:

Design patterns are for OOP, but OOP is not in favor anymore!

Realization 2:

Design patterns are not only for OOP, but come in many different forms. We really should talk more about them!

Reason 3:

Design patterns are simple!

The Toy Problem: Drawing Shapes

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

   virtual void draw( Screen&, /*...*/ ) const = 0;
   virtual void serialize( ByteStream&, /*...*/ ) const = 0;
   // ...
};


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

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

   void draw( Screen&, /*...*/ ) const override;
   void serialize( ByteStream&, /*...*/ ) const override;

   // ...

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

The Toy Problem: Drawing Shapes

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

   virtual void draw( Screen&, /*...*/ ) const = 0;
   virtual void serialize( ByteStream&, /*...*/ ) const = 0;
   // ...
};


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

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

   void draw( Screen&, /*...*/ ) const override;
   void serialize( ByteStream&, /*...*/ ) const override;

   // ...

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

The Toy Problem: Drawing Shapes

```
   virtual void draw( Screen&, /*...*/ ) const = 0;
   virtual void serialize( ByteStream&, /*...*/ ) const = 0;
   // ...
};


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

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

   void draw( Screen&, /*...*/ ) const override;
   void serialize( ByteStream&, /*...*/ ) const override;

   // ...

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


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

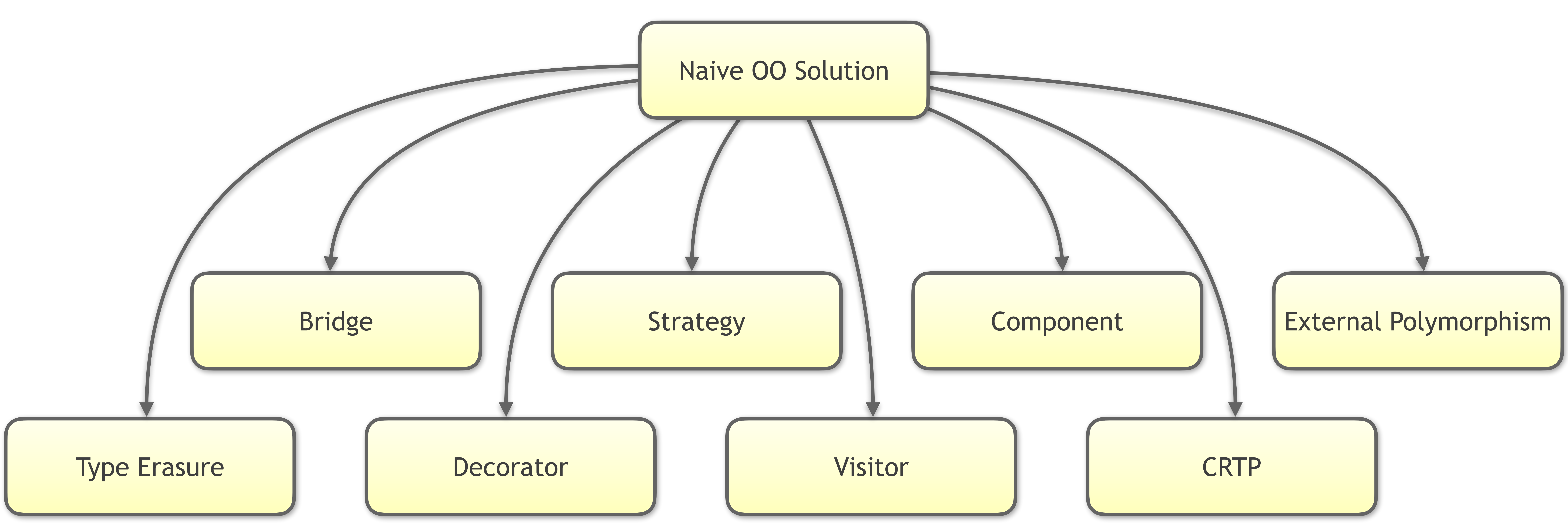
A Naive Object-Oriented Solution

```
class Shape
{
 public:
   virtual void draw( Screen&, /*...*/ ) const = 0;
   virtual void serialize( ByteStream&, /*...*/ ) const = 0;
   // ...
};

class Circle : public Shape
{
 public:
   void draw( Screen&, /*...*/ ) const override;
   void serialize( ByteStream&, /*...*/ ) const override;
   // ...
};
```

A naive implementation of a `Shape` abstraction causes a lot of dependencies:

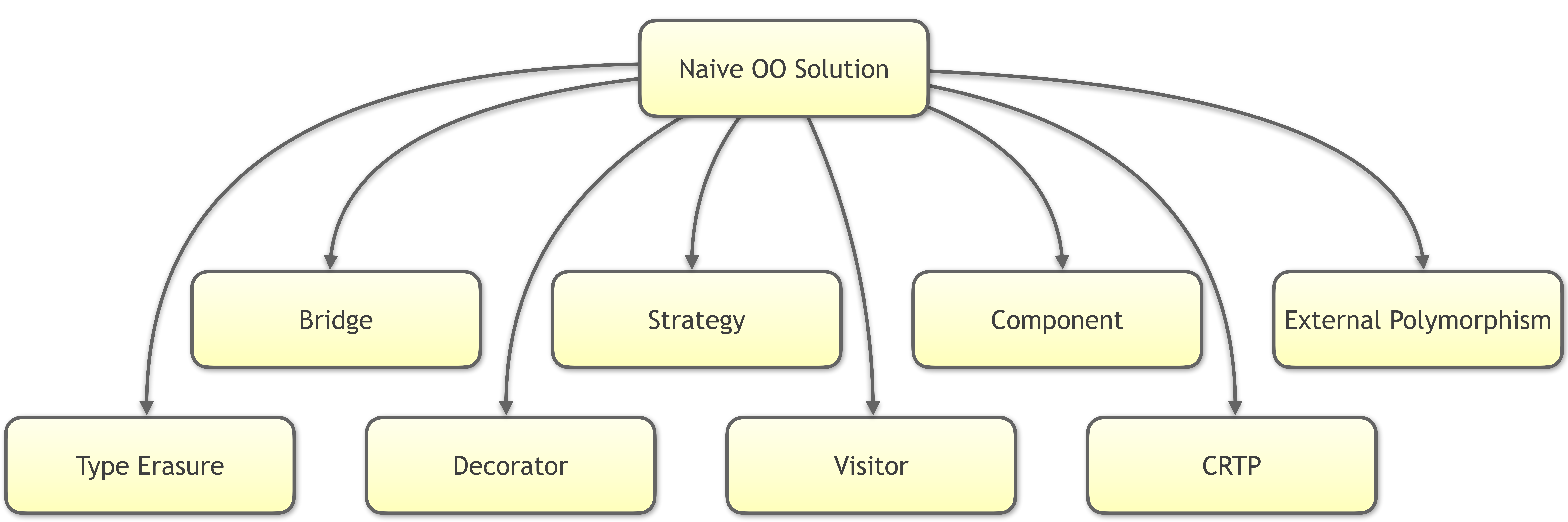
- coupling to the **type of arguments** (`Screen`, `ByteStream`, …);
- coupling to a **graphics library** in the derived classes;
- coupling to a **serialization library** in the derived classes;
- coupling between **orthogonal aspects** (draw vs. serialization);
- …

The Many Choices of Design Patterns



There are many solutions, …

The Many Choices of Design Patterns



… many design patterns to choose from …

… and so many important questions:

“Which design pattern should I use?”

“What are the consequences of my choice?”

“What are the advantages?”

“What are the disadvantages?”

Luckily there is an answer …

“The answer is,
it depends.”

Episode #80 - the SOLID principles
Phil Nash

Copy link
Tony

Watch on YouTube

Klaus Iglberger

Reason 3:

Design patterns are simple!

Realization 3:

Design patterns are **not** simple!

Design patterns are difficult!

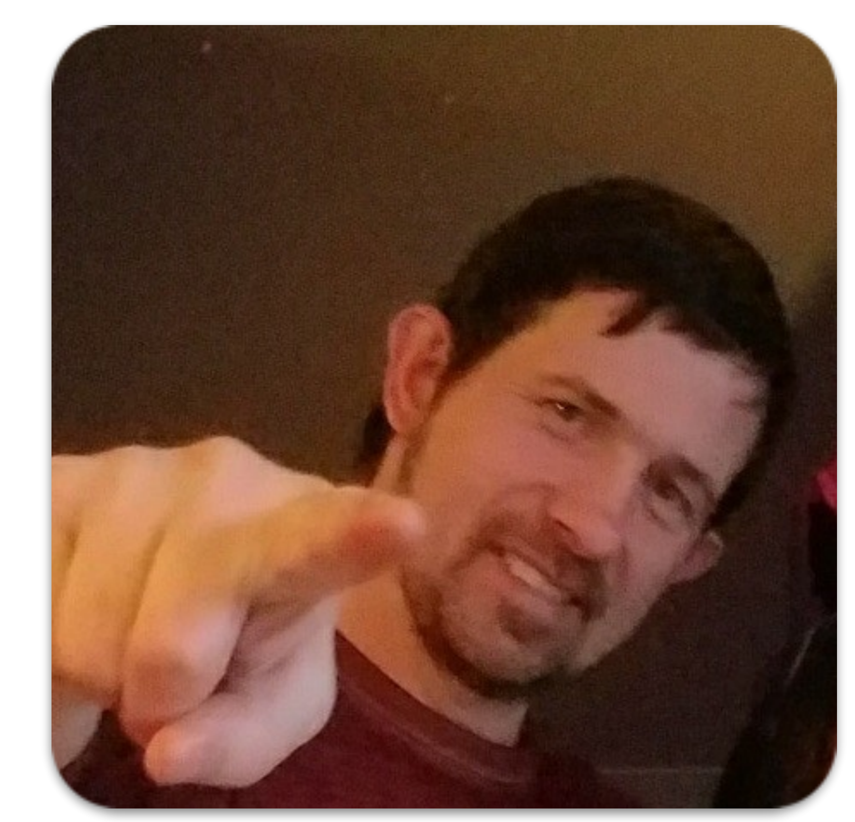
Software design is difficult, …

… very difficult …

… probably the most difficult part of developing software.

So we need to **talk** about dependencies and software design.

Of course some people **talk** about software design:

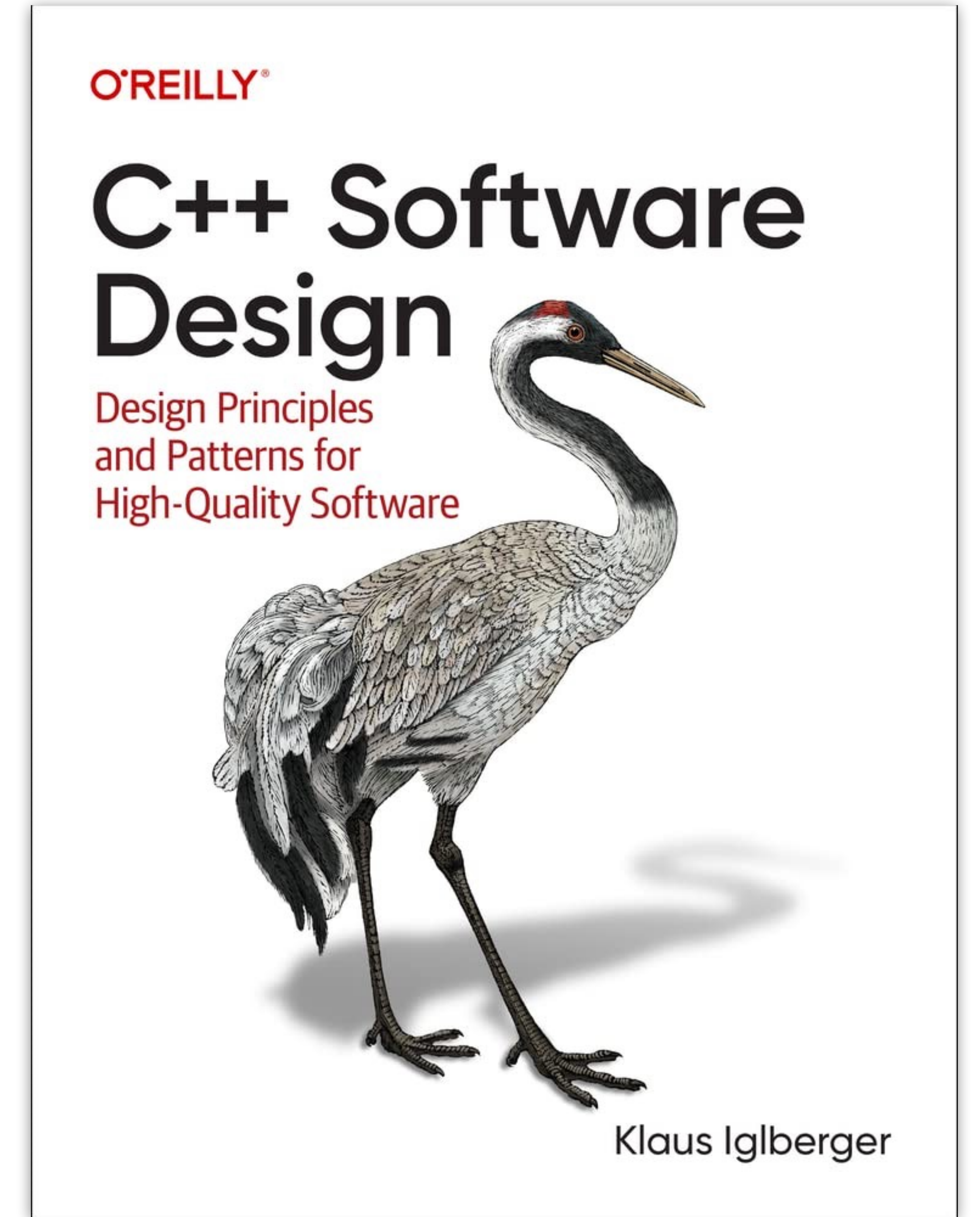


Peter Muldoon

Charley Bay

Tony van Eerd

Mike Shah



www.oreilly.com

Best Sellers

Our most popular products based on sales. Updated hourly.

Best Sellers in C++

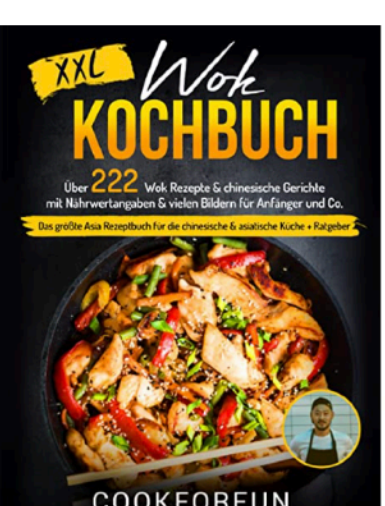
#1

C++ Programmieren: für Einsteiger: Der leich…
› Michael Bonacina
724
Paperback
€14.99

#2

Programmieren für Absolute Anfänger: Der…
› Alan Grid
85
Paperback
€13.76

#3



Wok Kochbuch XXL: Über 222 Wok Rezepte …
› Cookforfun
134
Paperback
€10.99

#4

C++ Programmieren: für Einsteiger: Der leich…
› Michael Bonacina
724
Kindle Edition
€9.99

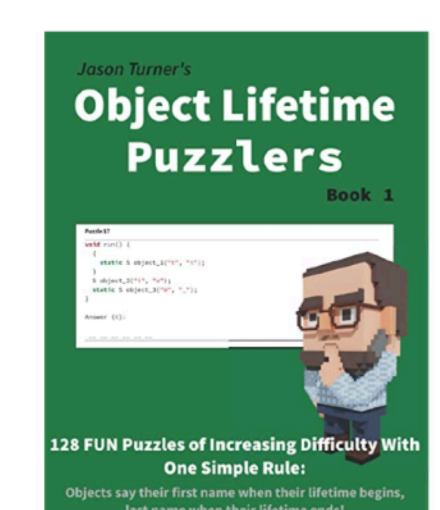
#5

XXL Diabetes Kochbuch und Ratgeber: mit se…
› Heinrich Stegmaier
46
Paperback
€15.99

#6

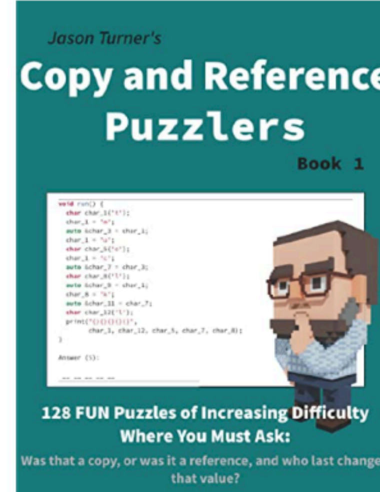
Grübeln stoppen: Wie du mit diesen erprobte…
› Leonie Weinberger
143
Paperback
€13.90

#7

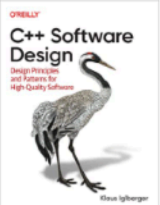


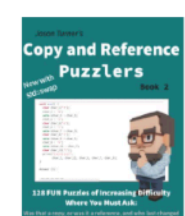
Object Lifetime Puzzlers - Book 1: 128 FUN…
› Jason Turner
23
Paperback
€9.44

#8

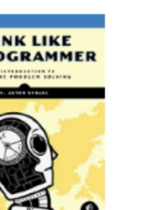


Copy and Reference Puzzlers - Book 1: 128 F…
› Jason Turner
4
Paperback
€9.51

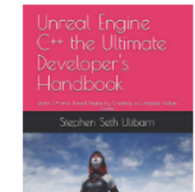
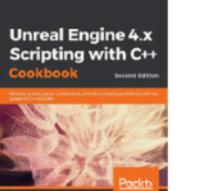




Hot New Releases
in C++



Most Wished For
in C++

Most Gifted
in C++

#9

#10

#11

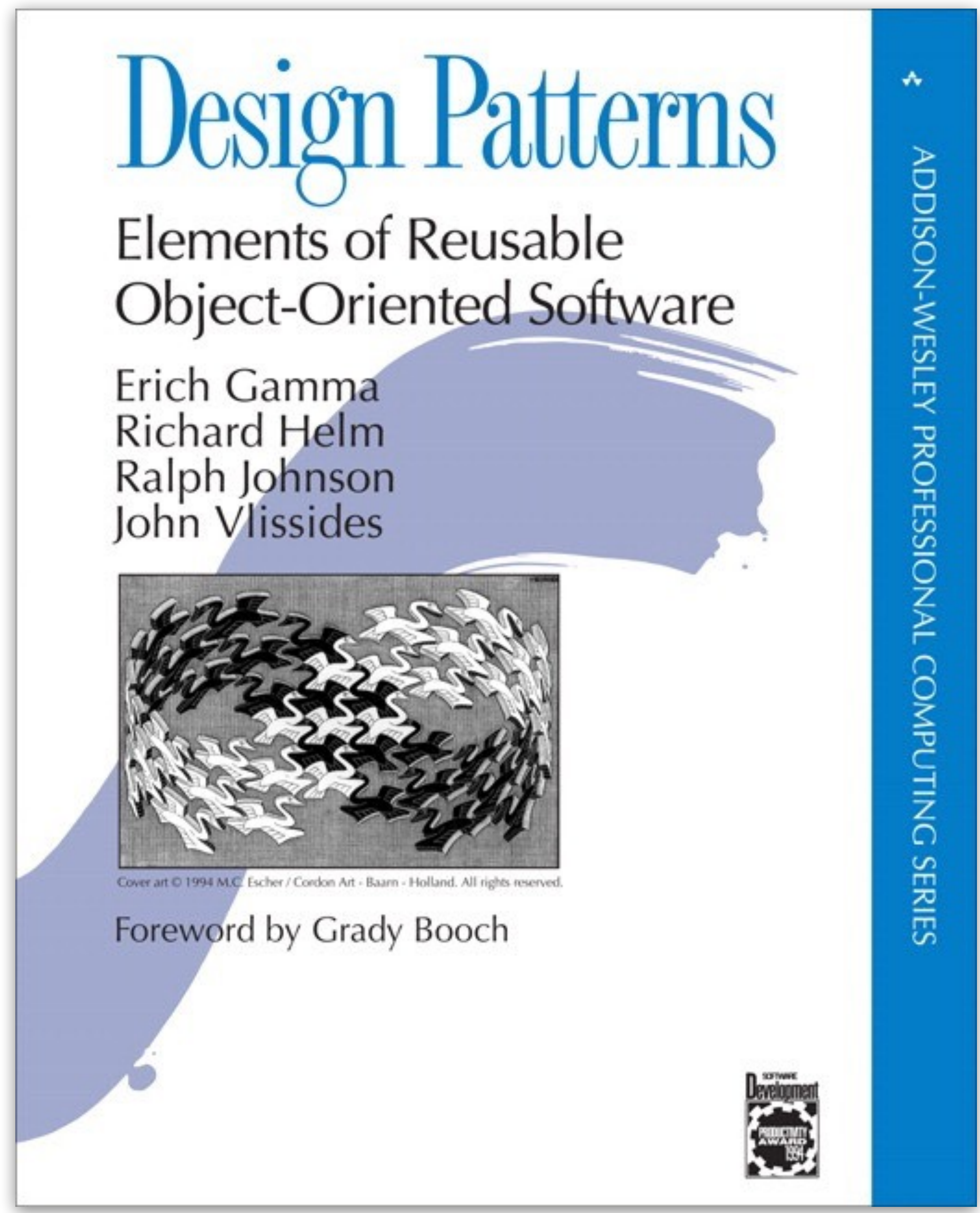
#12

Let’s **talk** about dependencies and software design …

… let’s **talk** about one of the most controversial patterns …

The Singleton Pattern

The Singleton Pattern

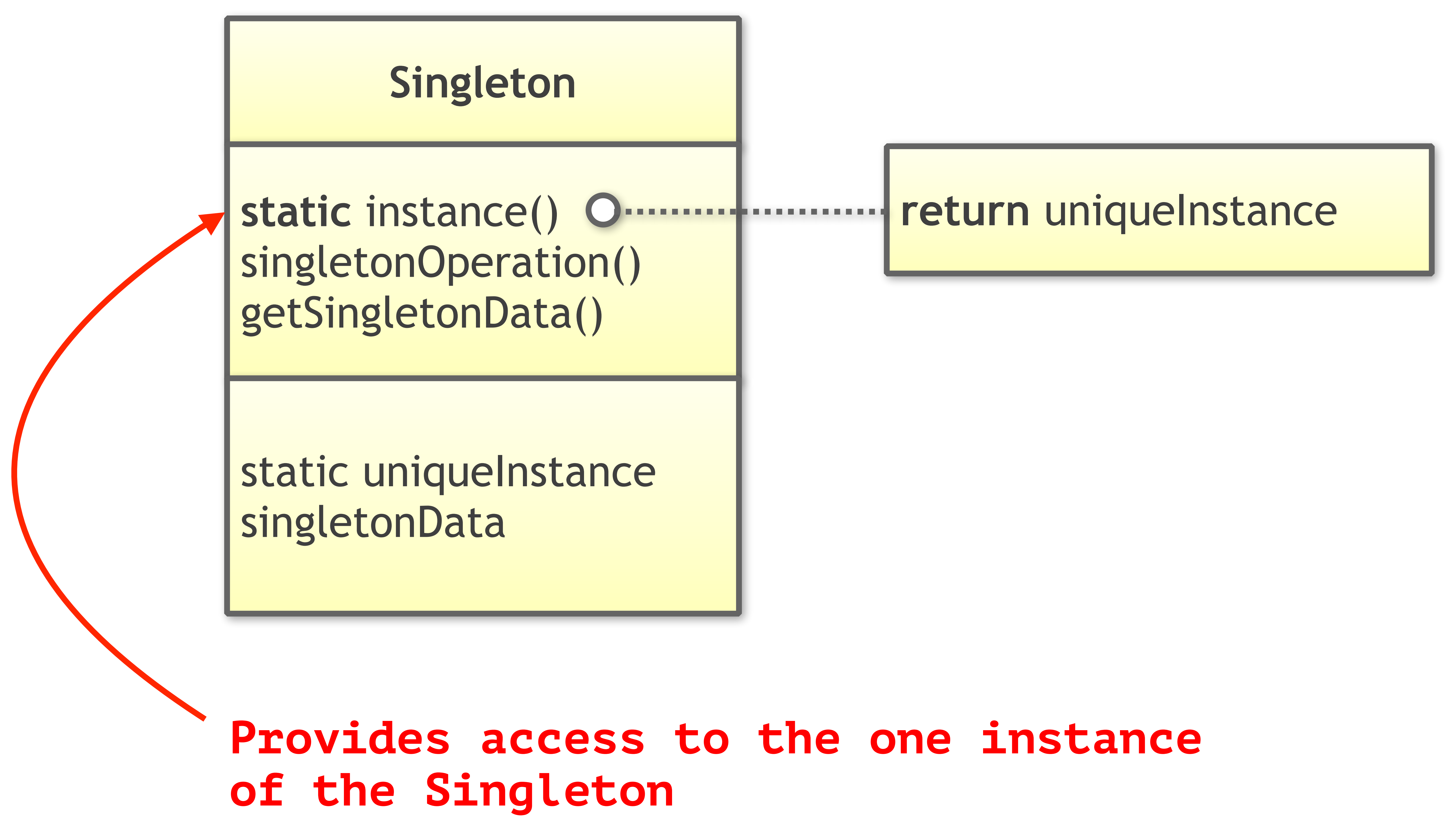


The Singleton Pattern

“Ensure a class has only one instance and provide a global point of access to it.”

(The Gang of Four, Design Patterns - Elements of Reusable Object-Oriented Software)

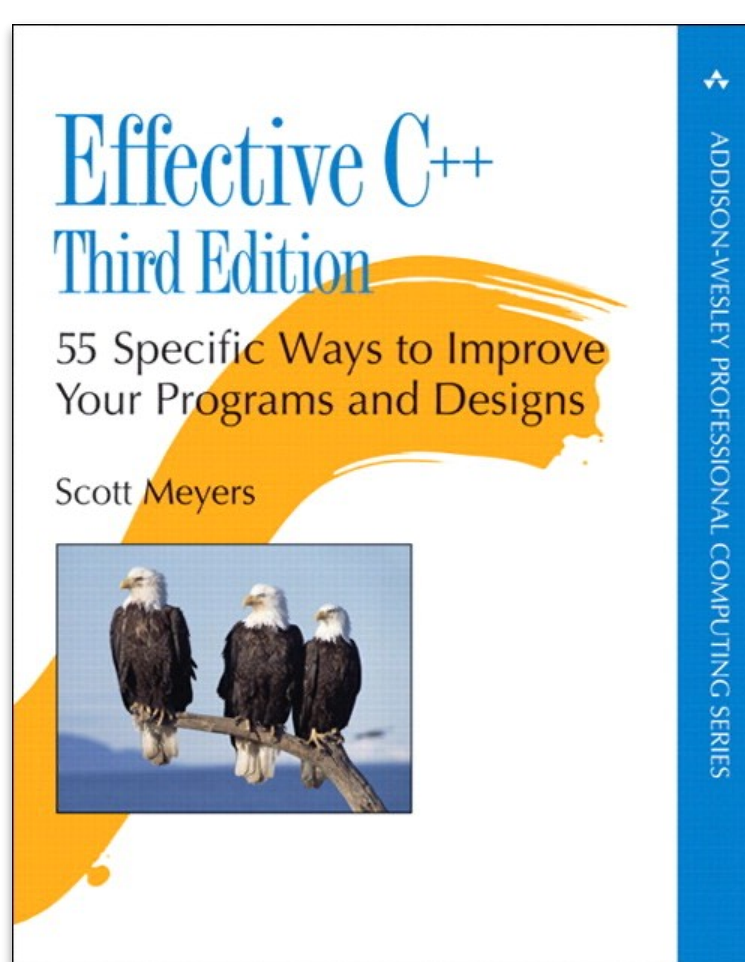
The Singleton Pattern



The Singleton Pattern

A common form of Singleton is the so-called **Meyers' Singleton**:

```
class Database
{
 public:
   static Database& instance()
   {
      static Database db;  // The one, unique instance
      return db;
   }

   bool write( /* some arguments */ );
   bool read( /* some arguments */ );

 private:
   Database() {}
   Database( Database const& ) = default;
};
```



The Problems of Singletons

Most developers are not particularly fond of Singletons …

The Problems of Singletons

"Anytime you make something accessible to every part of your program, you're asking for trouble."

(Robert Nystrom, Game Programming Patterns)

So Singleton is usually considered an anti-pattern. Why?

The Problems of Singletons

Singletons represent

Global State!

The Problems of Singletons

”The singleton pattern is one of the mechanisms people use to make global variables. In general, global variables are a bad idea for a couple of reasons.”

(Michael Feathers, Working Effectively with Legacy Code)

The Problems of Singletons

Global variables are a bad idea because …

- … they represent **mutable state**;
- … **read and write** operations are difficult to control (especially in multi-threaded environments);
- … they are **hard to reason** about;
- … they may be subject to the **Static Initialization Order Fiasko** (SIOF).

So the common advice is:

Don’t use Singletons!

The Problems of Singletons

”Globals are bad, m’kay?”

(Guy Davidson, Beautiful C++)

The Problems of Singletons

However, there are some intrinsically global aspects:

- memory
- time
- the system-wide configuration
- the audio or display device in computer games
- the logger
- …

The Problems of Singletons

“There are times when manually passing around an object is gratuitous or actively makes code harder to read. Some systems, like logging or memory management, shouldn’t be part of a module’s public API.“

(Robert Nystrom, Game Programming Patterns)

So it seems that sometimes
we need Singletons …

But there is a **bigger** problem.

Singletons create dependencies!

- Dependencies on concrete implementation details
- Invisible dependencies
- Artificial dependencies
- Bad dependencies

The Problems of Singletons

Singletons make it difficult to

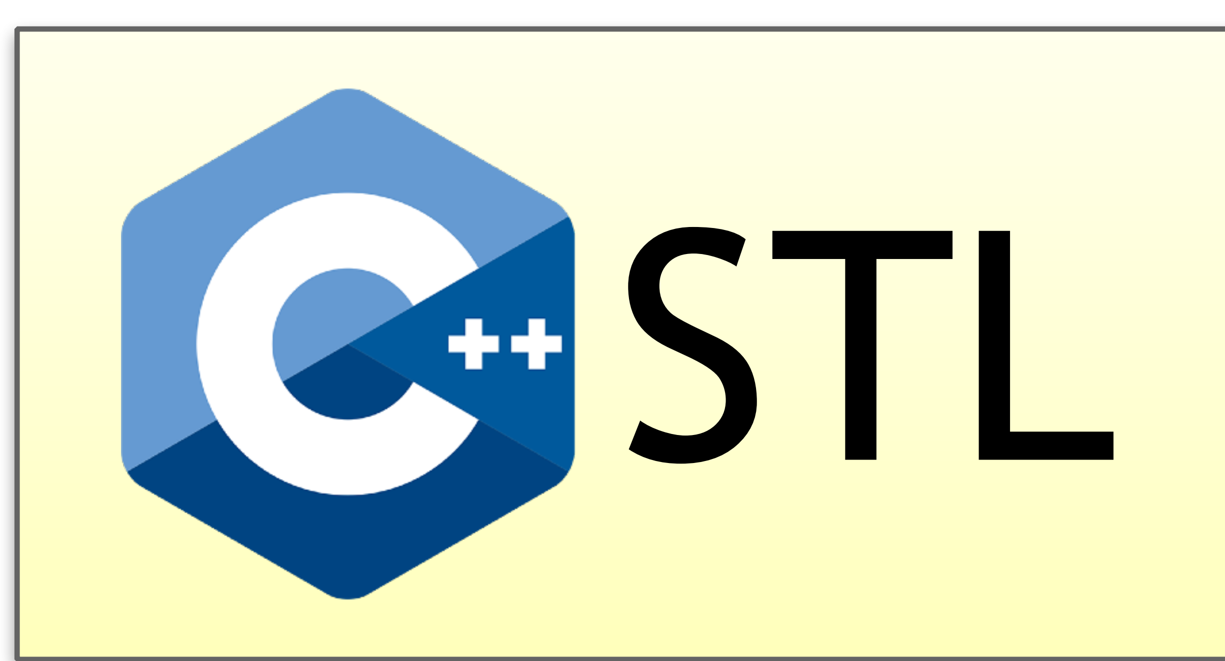
change, extend and test!

A Database Example

Let's return to the `Database` example:

```
class Database
{
 public:
   static Database& instance()
   {
      static Database db;  // The one, unique instance
      return db;
   }

   bool write( /* some arguments */ );
   bool read( /* some arguments */ );

 private:
   Database() {}
   Database( Database const& ) = default;
};
```

A Database Example: the Desired State …




```cpp
class Database
{
 public:
   static Database& instance();
   // ...
   bool write( /* some arguments */ );
   bool read( /* some arguments */ );
 private:
   Database() {}
   Database( Database const& ) = default;
};
```

High-level

(stable, low dependencies)

Architectural Boundary

```cpp
class Widget
{
 public:
   void doSomething( /*some arguments*/ )
   {
      // ...
      Database::instance().read( /*some arguments*/ );
      // ...
   }
};
```

Architectural Boundary

```cpp
class Gadget
{
 public:
   void doSomething( /*some arguments*/ )
   {
      // ...
      Database::instance().write( /*some arguments*/ );
      // ...
   }
};
```

Low-level

(volatile, malleable, high dependencies)

A Database Example: … and the Real State

```cpp
class Widget
{
 public:
   void doSomething( /*some arguments*/ )
   {
      // ...
      Database::instance().read( /*some arguments*/ );
      // ...
   }
};
```

High-level
(stable, low dependencies)

Architectural Boundary

This is NOT an architecture!

```cpp
class Gadget
{
 public:
   void doSomething( /*some arguments*/ )
   {
      // ...
      Database::instance().write( /*some arguments*/ );
      // ...
   }
};
```

Architectural Boundary

```cpp
class Database
{
 public:
   static Database& instance();
   // ...
   bool write( /* some arguments */ );
   bool read( /* some arguments */ );
 private:
   Database() {}
   Database( Database const& ) = default;
};
```

Low-level
(volatile, malleable, high dependencies)

Because of these dependencies, Singletons are always bad, right?

Singletons in the Standard Library

"…, it might be very surprising to learn that there are a couple of "Singleton"-like instances in the Standard Library. Seriously! And, honestly, they work fantastically!

(Klaus Iglberger, C++ Software Design)

An Example from the Standard Library

```
// ...
#include <memory_resource>

int main()
{
   std::array<std::byte,1000> raw;  // Note: not initialized!









   return EXIT_SUCCESS;
}
```

An Example from the Standard Library

```
// ...
#include <memory_resource>


int main()
{
   std::array<std::byte,1000> raw;  // Note: not initialized!

   std::pmr::monotonic_buffer_resource buffer{ raw.data(), raw.size()
                                             , std::pmr::null_memory_resource() };









   return EXIT_SUCCESS;
}
```

An Example from the Standard Library

```
// ...
#include <memory_resource>

int main()
{
   std::array<std::byte,1000> raw;  // Note: not initialized!

   std::pmr::monotonic_buffer_resource buffer{ raw.data(), raw.size()
                                             , std::pmr::null_memory_resource() };

   std::pmr::vector<std::pmr::string> strings{ &buffer };

   return EXIT_SUCCESS;
}
```

An Example from the Standard Library

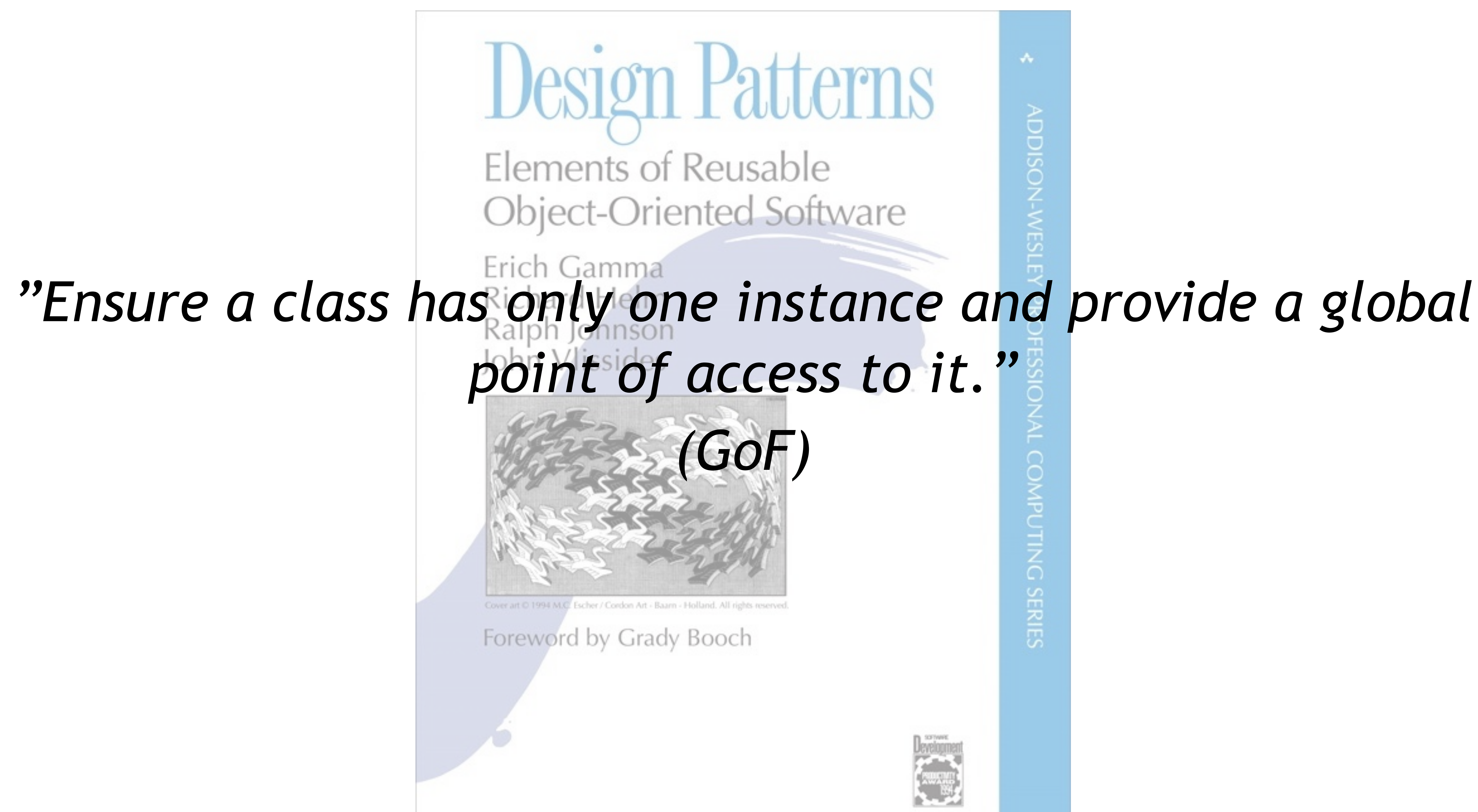
```cpp
// ...
#include <memory_resource>

int main()
{
   std::array<std::byte,1000> raw;  // Note: not initialized!

   std::pmr::monotonic_buffer_resource buffer{ raw.data(), raw.size()
                                             , std::pmr::null_memory_resource() };

   std::pmr::vector<std::pmr::string> strings{ &buffer };

   strings.emplace_back( "String longer than what SSO can handle" );
   strings.emplace_back( "Another long string that goes beyond SSO" );
   strings.emplace_back( "A third long string that cannot be handled by SSO" );

   for( const auto& s : strings ) {
      std::cout << std::quoted(s) << '\n';
   }

   return EXIT_SUCCESS;
}
```

An Example from the Standard Library

```
// ...
#include <memory_resource>

int main()
{
   std::array<std::byte,1000> raw;  // Note: not initialized!

   std::pmr::monotonic_buffer_resource buffer{ raw.data(), raw.size()
                                              , std::pmr::null_memory_resource() };

   std::pmr::vector<std::pmr::string> strings{ &buffer };

   strings.emplace_back( "String longer than what SSO can handle" );
   strings.emplace_back( "Another long string that goes beyond SSO" );
   strings.emplace_back( "A third long string that cannot be handled by SSO" );

   for( const auto& s : strings ) {
      std::cout << std::quoted(s) << '\n';
   }

   return EXIT_SUCCESS;
}
```

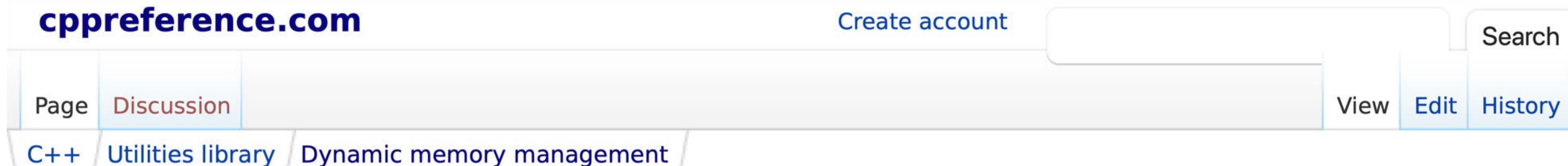
This acts as a Singleton! (annotation pointing to `std::pmr::null_memory_resource()`)

The Singleton Pattern



”Ensure a class has only one instance and provide a global point of access to it.”

(GoF)



std::pmr::null_memory_resource

Defined in header `<memory_resource>`

```
std::pmr::memory_resource* null_memory_resource() noexcept;
```
(since C++17)

Returns a pointer to a `memory_resource` that doesn't perform any allocation.

Return value

Returns a pointer p to a static storage duration object of a type derived from `std::pmr::memory_resource`, with the following properties:

- its `allocate()` function always throws `std::bad_alloc`;
- its `deallocate()` function has no effect;
- for any `memory_resource` r, `p->is_equal(r)` returns `&r == p`.

The same value is returned every time this function is called.

Global point of access, exactly one instance: fulfills the definition of the Singleton pattern!

Example

The program demos the main usage of `null_memory_resouce`: ensure that a memory pool which requires memory allocated on the stack will NOT allocate memory on the heap if it needs more memory.

Run this code

An Example from the Standard Library

Still, this represents a specific allocator, and thus a dependency

```
// ...
#include <memory_resource>

int main()
{
   std::array<std::byte,1000> raw;  // Note: not initialized!

   std::pmr::monotonic_buffer_resource buffer{ raw.data(), raw.size()
                                              , std::pmr::null_memory_resource() };

   std::pmr::vector<std::pmr::string> strings{ &buffer };

   strings.emplace_back( "String longer than what SSO can handle" );
   strings.emplace_back( "Another long string that goes beyond SSO" );
   strings.emplace_back( "A third long string that cannot be handled by SSO" );

   for( const auto& s : strings ) {
      std::cout << std::quoted(s) << '\n';
   }

   return EXIT_SUCCESS;
}
```

An Example from the Standard Library

```
// ...
#include <memory_resource>

int main()
{
   std::array<std::byte,1000> raw;  // Note: not initialized!


   std::pmr::monotonic_buffer_resource buffer{ raw.data(), raw.size()
                                              , std::pmr::get_default_resource() };

   std::pmr::vector<std::pmr::string> strings{ &buffer };

   strings.emplace_back( "String longer than what SSO can handle" );
   strings.emplace_back( "Another long string that goes beyond SSO" );
   strings.emplace_back( "A third long string that cannot be handled by SSO" );

   for( const auto& s : strings ) {
      std::cout << std::quoted(s) << '\n';
   }

   return EXIT_SUCCESS;
}
```

Provides access to the system-wide default allocator …

An Example from the Standard Library

… which can be set via
std::pmr::set_default_resource()

```
// ...
#include <memory_resource>

int main()
{
   std::array<std::byte,1000> raw;  // Note: not initialized!

   std::pmr::set_default_resource( std::pmr::null_memory_resource() );

   std::pmr::monotonic_buffer_resource buffer{ raw.data(), raw.size()
                                             , std::pmr::get_default_resource() };

   std::pmr::vector<std::pmr::string> strings{ &buffer };

   strings.emplace_back( "String longer than what SSO can handle" );
   strings.emplace_back( "Another long string that goes beyond SSO" );
   strings.emplace_back( "A third long string that cannot be handled by SSO" );

   for( const auto& s : strings ) {
      std::cout << std::quoted(s) << '\n';
   }

   return EXIT_SUCCESS;
}
```

An Example from the Standard Library

This gives us the ability to customize the system-wide allocator …

```
// ...
#include <memory_resource>

int main()
{
   std::array<std::byte,1000> raw;  // Note: not initialized!

   std::pmr::set_default_resource( std::pmr::null_memory_resource() );

   std::pmr::monotonic_buffer_resource buffer{ raw.data(), raw.size()
                                             , std::pmr::get_default_resource() };

   std::pmr::vector<std::pmr::string> strings{ &buffer };

   strings.emplace_back( "String longer than what SSO can handle" );
   strings.emplace_back( "Another long string that goes beyond SSO" );
   strings.emplace_back( "A third long string that cannot be handled by SSO" );

   for( const auto& s : strings ) {
      std::cout << std::quoted(s) << '\n';
   }

   return EXIT_SUCCESS;
}
```

An Example from the Standard Library

… in other words, we can inject the dependency on the allocator …

```
// ...
#include <memory_resource>

int main()
{
   std::array<std::byte,1000> raw;  // Note: not initialized!

   std::pmr::set_default_resource( std::pmr::null_memory_resource() );

   std::pmr::monotonic_buffer_resource buffer{ raw.data(), raw.size()
                                              , std::pmr::get_default_resource() };

   std::pmr::vector<std::pmr::string> strings{ &buffer };

   strings.emplace_back( "String longer than what SSO can handle" );
   strings.emplace_back( "Another long string that goes beyond SSO" );
   strings.emplace_back( "A third long string that cannot be handled by SSO" );

   for( const auto& s : strings ) {
      std::cout << std::quoted(s) << '\n';
   }

   return EXIT_SUCCESS;
}
```

An Example from the Standard Library

```
// ...
#include <memory_resource>

int main()
{
   std::array<std::byte,1000> raw;  // Note: not initialized!

   std::pmr::set_default_resource( std::pmr::null_memory_resource() );

   std::pmr::monotonic_buffer_resource buffer{ raw.data(), raw.size()
                                              , std::pmr::get_default_resource() };

   std::pmr::vector<std::pmr::string> strings{ &buffer };

   strings.emplace_back( "String longer than what SSO can handle" );
   strings.emplace_back( "Another long string that goes beyond SSO" );
   strings.emplace_back( "A third long string that cannot be handled by SSO" );

   for( const auto& s : strings ) {
      std::cout << std::quoted(s) << '\n';
   }

   return EXIT_SUCCESS;
}
```

Yes, this is an example for the Strategy design pattern!

Repairing the Database Example

```
class Widget
{
 public:
   void doSomething( /*some arguments*/ )
   {
      // ...
      Database::instance().read( /*some arguments*/ );
      // ...
   }
};
```

High-level

(stable, low dependencies)

Architectural Boundary

What is missing is an abstraction …

```
class Gadget
{
 public:
   void doSomething( /*some arguments*/ )
   {
      // ...
      Database::instance().write( /*some arguments*/ );
      // ...
   }
};
```

Architectural Boundary

```
class Database
{
 public:
   static Database& instance();
   // ...
   bool write( /* some arguments */ );
   bool read( /* some arguments */ );
 private:
   Database() {}
   Database( Database const& ) = default;
};
```

Low-level

(volatile, malleable, high dependencies)

Repairing the Database Example

```
//---- <PersistenceInterface.h> ----------------

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

   bool read( /*some arguments*/ ) const
   {
      return do_read( /*...*/ );
   }
   bool write( /*some arguments*/ )
   {
      return do_write( /*...*/ );
   }

   // ... More database specific functionality

 private:
   virtual bool do_read( /*some arguments*/ ) const = 0;
   virtual bool do_write( /*some arguments*/ ) = 0;
};
```

Repairing the Database Example

```
//---- <PersistenceInterface.h> ----------------

class PersistenceInterface
{
   // ...
 private:
   virtual bool do_read( /*some arguments*/ ) const = 0;
   virtual bool do_write( /*some arguments*/ ) = 0;
};


//---- <Database.h> ----------------

class Database : public PersistenceInterface
{
 public:
   // ... Potentially access to data members

 private:
   bool do_read( /*some arguments*/ ) const override;
   bool do_write( /*some arguments*/ ) override;
   // ... More database-specific functionality

   // ... Potentially some data members
};
```

Repairing the Database Example

```
//---- <PersistenceInterface.h> ----------------

class PersistenceInterface
{
   // ...
 private:
   virtual bool do_read( /*some arguments*/ ) const = 0;
   virtual bool do_write( /*some arguments*/ ) = 0;
};


PersistenceInterface* get_persistence_interface();
void set_persistence_interface( PersistenceInterface* persistence );
```

Repairing the Database Example

```
//---- <PersistenceInterface.h> ----------------

class PersistenceInterface
{
   // ...
 private:
   virtual bool do_read( /*some arguments*/ ) const = 0;
   virtual bool do_write( /*some arguments*/ ) = 0;
};


PersistenceInterface* get_persistence_interface();
void set_persistence_interface( PersistenceInterface* persistence );

// Declaration of the one 'instance' variable
extern PersistenceInterface* instance;
```

Repairing the Database Example

```
//---- <PersistenceInterface.h> ----------------

class PersistenceInterface
{
   // ...
 private:
   virtual bool do_read( /*some arguments*/ ) const = 0;
   virtual bool do_write( /*some arguments*/ ) = 0;
};


PersistenceInterface* get_persistence_interface();
void set_persistence_interface( PersistenceInterface* persistence );

// Declaration of the one 'instance' variable
extern PersistenceInterface* instance;



//---- <PersistenceInterface.cpp> ----------------

#include <Database.h>

// Definition of the one 'instance' variable
PersistenceInterface* instance = nullptr;

PersistenceInterface* get_persistence_interface()
{
```

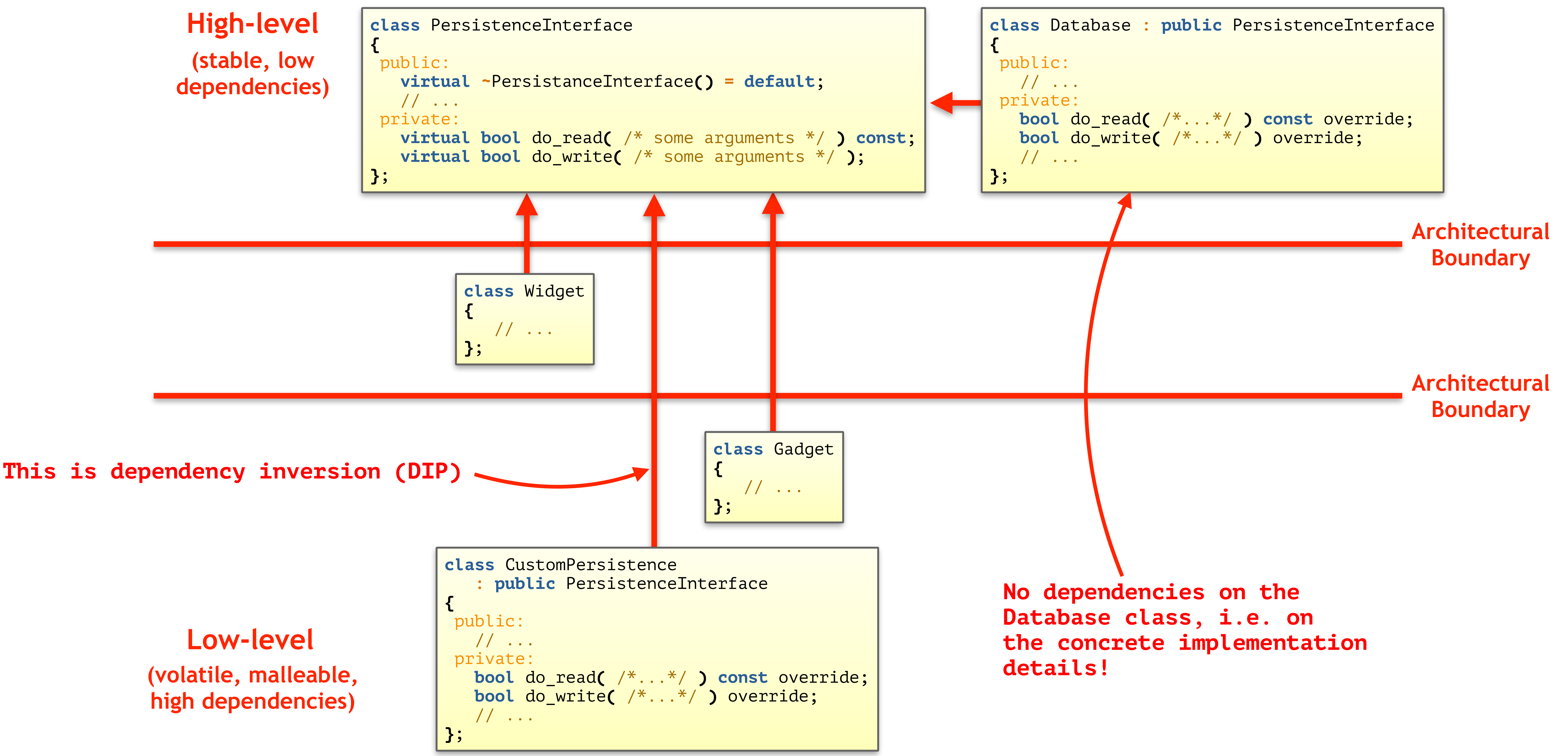
Repairing the Database Example

```
//---- <PersistenceInterface.cpp> ----------------

#include <Database.h>

// Definition of the one 'instance' variable
PersistenceInterface* instance = nullptr;

PersistenceInterface* get_persistence_interface()
{
      if( !instance ) {
         static Database db;
         instance = &db;
      }

   return instance;
}

void set_persistence_interface( PersistenceInterface* persistence )
{
   instance = persistence;
}
```

Repairing the Database Example

```
//---- <PersistenceInterface.cpp> ----------------

#include <Database.h>

// Definition of the one 'instance' variable
PersistenceInterface* instance = nullptr;

PersistenceInterface* get_persistence_interface()
{
   // Local object, initialized by an 'Immediately Invoked Lambda Expression (IILE)'
   static bool init = [](){
      if( !instance ) {
         static Database db;
         instance = &db;
      }
      return true;  // or false, as the actual value does not matter.
   }();  // Note the '()' after the lambda expression. This invokes the lambda.

   return instance;
}

void set_persistence_interface( PersistenceInterface* persistence )
{
   instance = persistence;
}
```

Repairing the Database Example



Repairing the Database Example

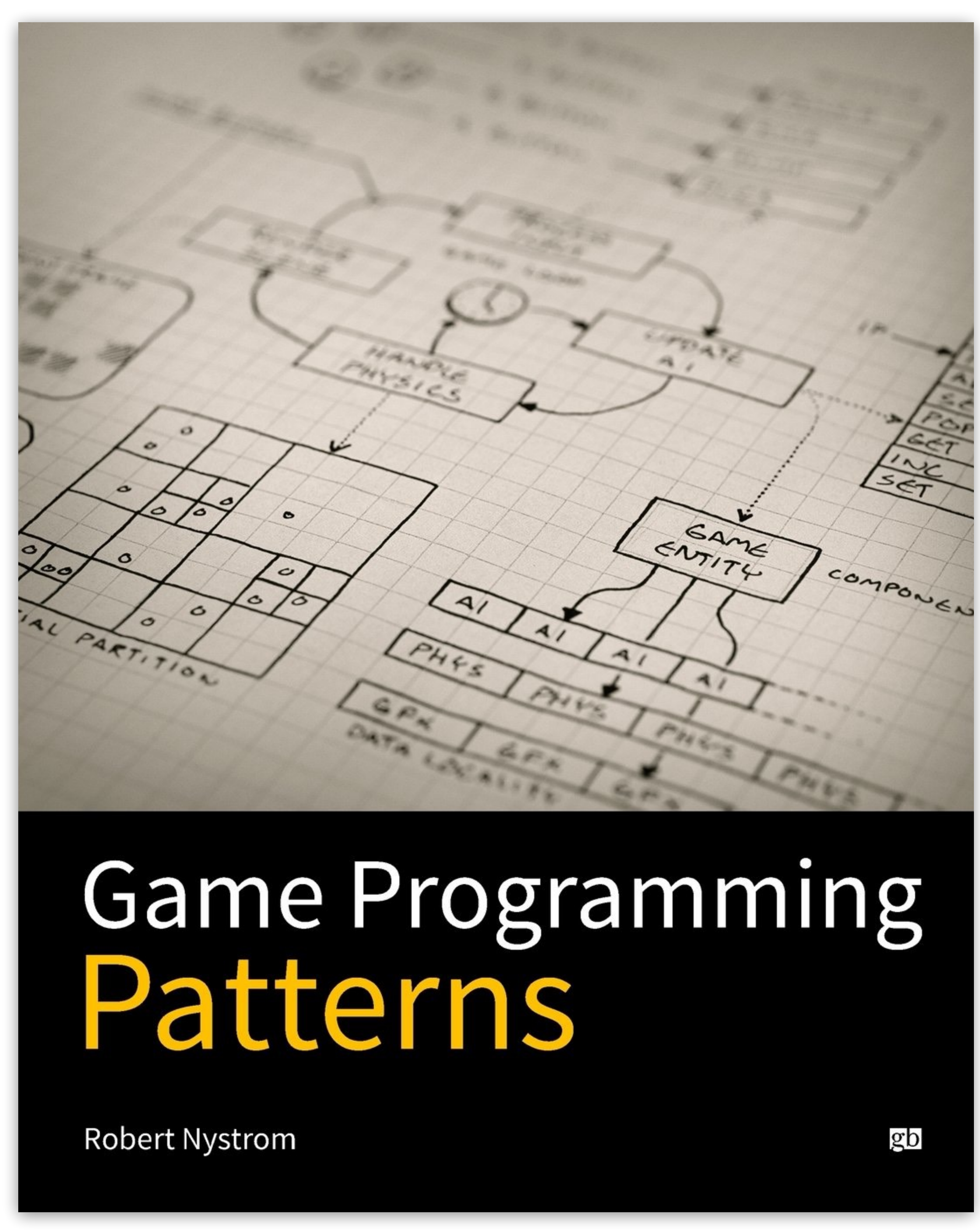
Dependency inversion (for instance via the Strategy design pattern) is the necessary first step to an “acceptable” Singleton.

It is an “acceptable” singleton because …

- … user code no longer depends on one specific implementation;
- … implementation details can be changed and extended;
- … code using the Singleton can be tested;
- … it enables you to cope with the intrinsically global aspects.

It’s still global, though, so take care!

The Service Locator Design Pattern



The Service Locator Design Pattern

"Provide a global point of access to a service without coupling users to the concrete class that implements it."

(Robert Nystrom, Game Programming Patterns)

The Reality of Design Patterns

“Design patterns are everywhere.”

(Klaus Iglberger, C++ Software Design)

Summary

- Software design is not an afterthought, but essential for the success of a project
- Let's stop pretending that C++ is all about features and standards
- Let's start to talk about the really important aspects of software
- Design patterns are everywhere …
  - … so learn about different patterns
  - … and their advantages and disadvantages
- Let's make software easy to change, extend, and test, so let's …

Break Dependencies!

Breaking Dependencies:
The Path to
High-Quality Software
Klaus Iglberger, Closing Keynote, Meeting C++ 2022
klaus.iglberger@gmx.de