Design Patterns: Facts and Misconceptions

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

Episode #80 - the SOLID principles
Copy link
Phil Nash
Tony
Klaus Iglberger
Watch on YouTube

Let’s talk about

Software Design and **Design Patterns**

Disclaimer

All content is based on personal, subjective impressions and opinions.

You may have another opinion, and that is fine!

There is no definitive answer in software design.

It depends.

But that is the reason why it is fun.

`std::make_unique()` …

- … improves exception safety;
- … fulfills the Single-Responsibility Principle (SRP);
- … is a factory function.

`std::make_unique()` …

- … improves exception safety;
- … fulfills the Single-Responsibility Principle (SRP);
- … is a factory function.

```cpp
void process( std::unique_ptr<Widget> a, std::unique_ptr<Widget> b );

int main()
{
   process( std::unique_ptr<Widget>( new Widget( 1, /*...*/) )
          , std::unique_ptr<Widget>( new Widget( 2, /*...*/) ) );

   // ...

   return EXIT_SUCCESS;
}
```

Possible order of operations :

```
void process( std::unique_ptr<Widget> a, std::unique_ptr<Widget> b );

int main()
{
   process( std::unique_ptr<Widget>( new Widget( 1, /*...*/) )
          , std::unique_ptr<Widget>( new Widget( 2, /*...*/) ) );

   // ...

   return EXIT_SUCCESS;
}
```

Possible order of operations:

- `new Widget( 1, /*...*/)`

```
void process( std::unique_ptr<Widget> a, std::unique_ptr<Widget> b );

int main()
{
   process( std::unique_ptr<Widget>( new Widget( 1, /*...*/) )
          , std::unique_ptr<Widget>( new Widget( 2, /*...*/) ) );

   // ...

   return EXIT_SUCCESS;
}
```

Possible order of operations :

- `new Widget( 1, /*...*/)`
- `std::unique_ptr<Widget>( /*...*/ )`

```
void process( std::unique_ptr<Widget> a, std::unique_ptr<Widget> b );

int main()
{
   process( std::unique_ptr<Widget>( new Widget( 1, /*...*/) )
          , std::unique_ptr<Widget>( new Widget( 2, /*...*/) ) );

   // ...

   return EXIT_SUCCESS;
}
```

Possible order of operations:

- `new Widget( 1, /*...*/)`
- `std::unique_ptr<Widget>( /*...*/ )`
- `new Widget( 2, /*...*/)`

```
void process( std::unique_ptr<Widget> a, std::unique_ptr<Widget> b );

int main()
{
   process( std::unique_ptr<Widget>( new Widget( 1, /*...*/) )
          , std::unique_ptr<Widget>( new Widget( 2, /*...*/) ) );

   // ...

   return EXIT_SUCCESS;
}
```

Possible order of operations:

- `new Widget( 1, /*...*/)`
- `std::unique_ptr<Widget>( /*...*/ )`
- `new Widget( 2, /*...*/)`
- `std::unique_ptr<Widget>( /*...*/ )`

```
void process( std::unique_ptr<Widget> a, std::unique_ptr<Widget> b );

int main()
{
   process( std::unique_ptr<Widget>( new Widget( 1, /*...*/) )
          , std::unique_ptr<Widget>( new Widget( 2, /*...*/) ) );

   // ...

   return EXIT_SUCCESS;
}
```

Possible order of operations :

- `new Widget( 1, /*...*/)`
- `new Widget( 2, /*...*/)   // Resource leak in case of exception`
- `std::unique_ptr<Widget>( /*...*/ )`
- `std::unique_ptr<Widget>( /*...*/ )`

```cpp
void process( std::unique_ptr<Widget> a, std::unique_ptr<Widget> b );

int main()
{
   process( std::unique_ptr<Widget>( new Widget( 1, /*...*/) )
          , std::unique_ptr<Widget>( new Widget( 2, /*...*/) ) );

   // ...

   return EXIT_SUCCESS;
}
```

Possible order of operations:

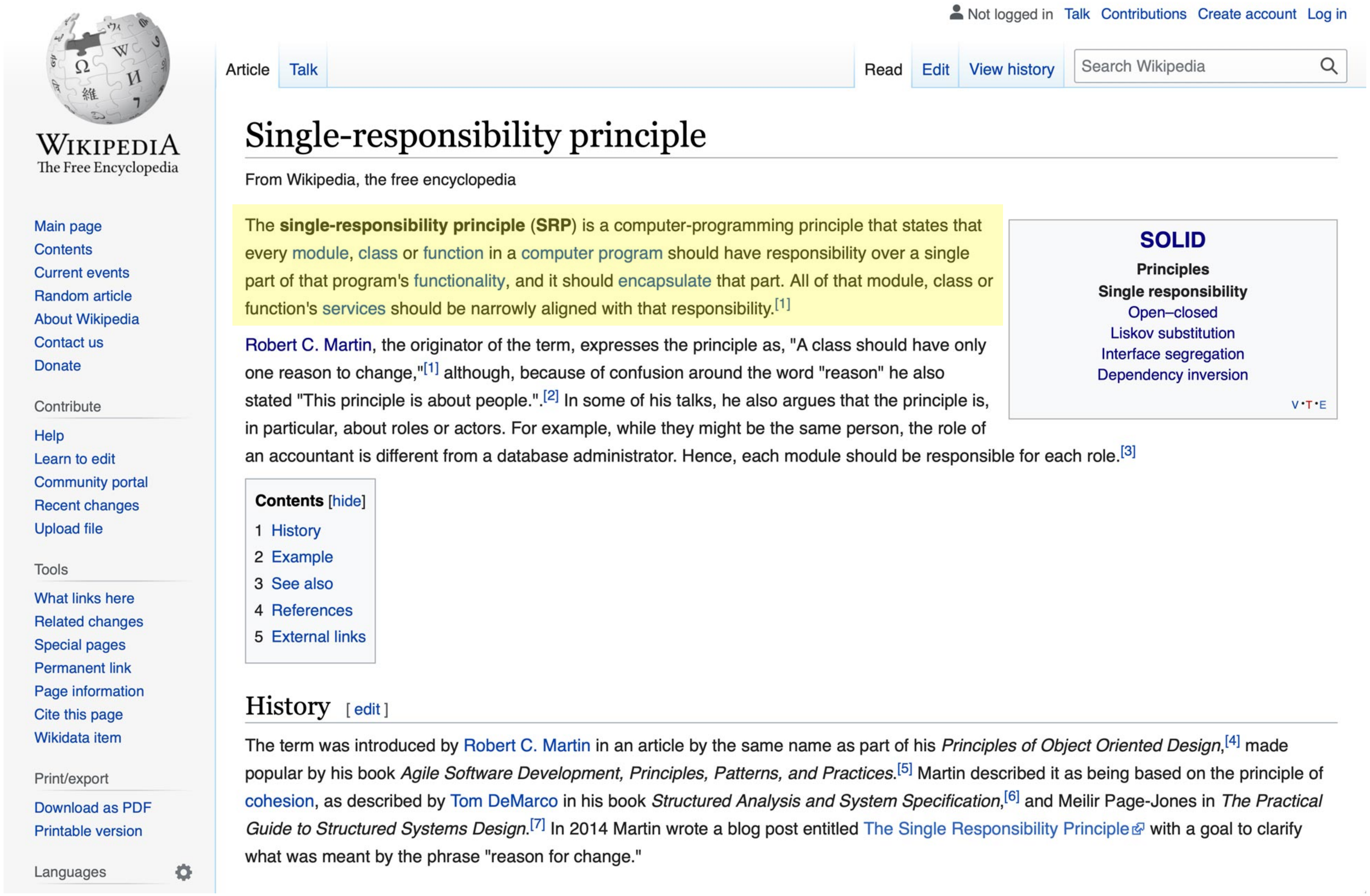
- `new Widget( 1, /*...*/)`
- `std::unique_ptr<Widget>( /*...*/ )`
- `new Widget( 2, /*...*/)    // No longer an issue since C++17`
- `std::unique_ptr<Widget>( /*...*/ )`

`std::make_unique()` …

- ~~… improves exception safety;~~
- … fulfills the Single-Responsibility Principle (SRP);
- … is a factory function.

`std::make_unique()` …

- ~~… improves exception safety;~~
- … fulfills the Single-Responsibility Principle (SRP);
- … is a factory function.



Single-responsibility principle

From Wikipedia, the free encyclopedia

The **single-responsibility principle** (**SRP**) is a computer-programming principle that states that every module, class or function in a computer program should have responsibility over a single part of that program's functionality, and it should encapsulate that part. All of that module, class or function's services should be narrowly aligned with that responsibility.[1]

SOLID
Principles
Single responsibility
Open–closed
Liskov substitution
Interface segregation
Dependency inversion
V·T·E

Robert C. Martin, the originator of the term, expresses the principle as, "A class should have only one reason to change,"[1] although, because of confusion around the word "reason" he also stated "This principle is about people.".[2] In some of his talks, he also argues that the principle is, in particular, about roles or actors. For example, while they might be the same person, the role of an accountant is different from a database administrator. Hence, each module should be responsible for each role.[3]

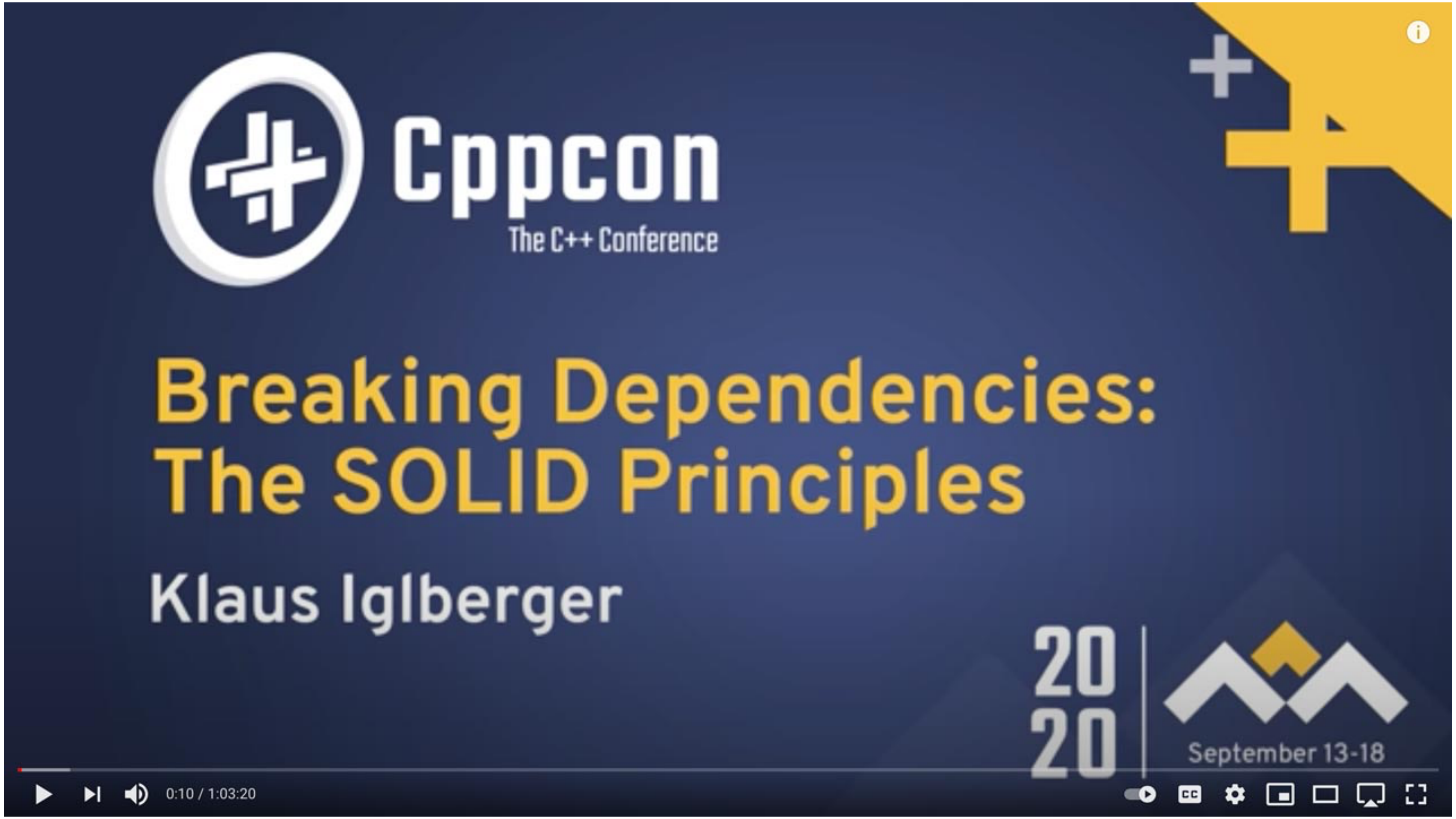
Contents [hide]

History [edit]

The term was introduced by Robert C. Martin in an article by the same name as part of his *Principles of Object Oriented Design*,[4] made popular by his book *Agile Software Development, Principles, Patterns, and Practices*.[5] Martin described it as being based on the principle of cohesion, as described by Tom DeMarco in his book *Structured Analysis and System Specification*,[6] and Meilir Page-Jones in *The Practical Guide to Structured Systems Design*.[7] In 2014 Martin wrote a blog post entitled The Single Responsibility Principle with a goal to clarify what was meant by the phrase "reason for change."

“Everything should do just one thing.”

(Common Knowledge?)


Cppcon
The C++ Conference

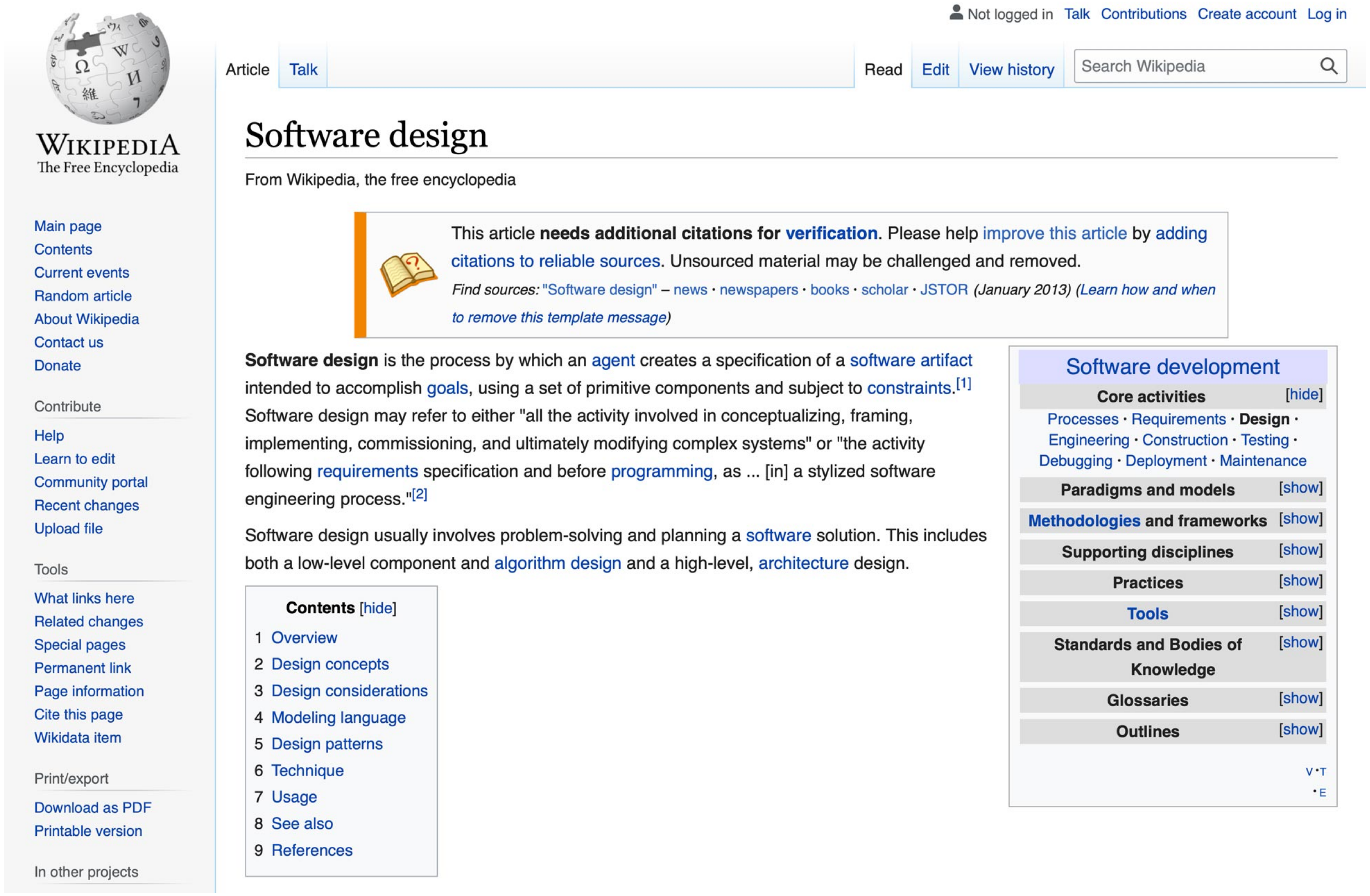
Breaking Dependencies:
The SOLID Principles

Klaus Iglberger

2020 September 13-18

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

(Klaus Iglberger)



Not logged in Talk Contributions Create account Log in

Article Talk Read Edit View history Search Wikipedia

WIKIPEDIA
The Free Encyclopedia

Main page
Contents
Current events
Random article
About Wikipedia
Contact us
Donate

Contribute

Help
Learn to edit
Community portal
Recent changes
Upload file

Tools

What links here
Related changes
Special pages
Permanent link
Page information
Cite this page
Wikidata item

Print/export

Download as PDF
Printable version

In other projects

Software design

From Wikipedia, the free encyclopedia

This article **needs additional citations for verification**. Please help improve this article by adding citations to reliable sources. Unsourced material may be challenged and removed.
Find sources: "Software design" – news · newspapers · books · scholar · JSTOR *(January 2013) (Learn how and when to remove this template message)*

Software design is the process by which an agent creates a specification of a software artifact intended to accomplish goals, using a set of primitive components and subject to constraints.[1] Software design may refer to either "all the activity involved in conceptualizing, framing, implementing, commissioning, and ultimately modifying complex systems" or "the activity following requirements specification and before programming, as ... [in] a stylized software engineering process."[2]

Software design usually involves problem-solving and planning a software solution. This includes both a low-level component and algorithm design and a high-level, architecture design.

Contents [hide]

Software development

Core activities [hide]
Processes · Requirements · **Design** · Engineering · Construction · Testing · Debugging · Deployment · Maintenance

Paradigms and models [show]
Methodologies and frameworks [show]
Supporting disciplines [show]
Practices [show]
Tools [show]
Standards and Bodies of Knowledge [show]
Glossaries [show]
Outlines [show]

V · T · E

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

…

The goal of software architecture is to minimize the human resources required to build and maintain the required system.”

(Robert C. Martin, Clean Architecture)

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

(Klaus Iglberger)

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

```
namespace std {

template< typename T, typename... Args >
std::unique_ptr<T> std::make_unique( Args&&... args )
{
   return std::unique_ptr<T>( new T( std::forward<Args>(args)... ) );
}

} // namespace std
```

`std::make_unique()` …

- … does not resolve any dependency;
- … does not provide any (semantic) abstraction (no customization);
- … has nothing to do with software design;
- … has nothing to do with design principles.

`std::make_unique()` …

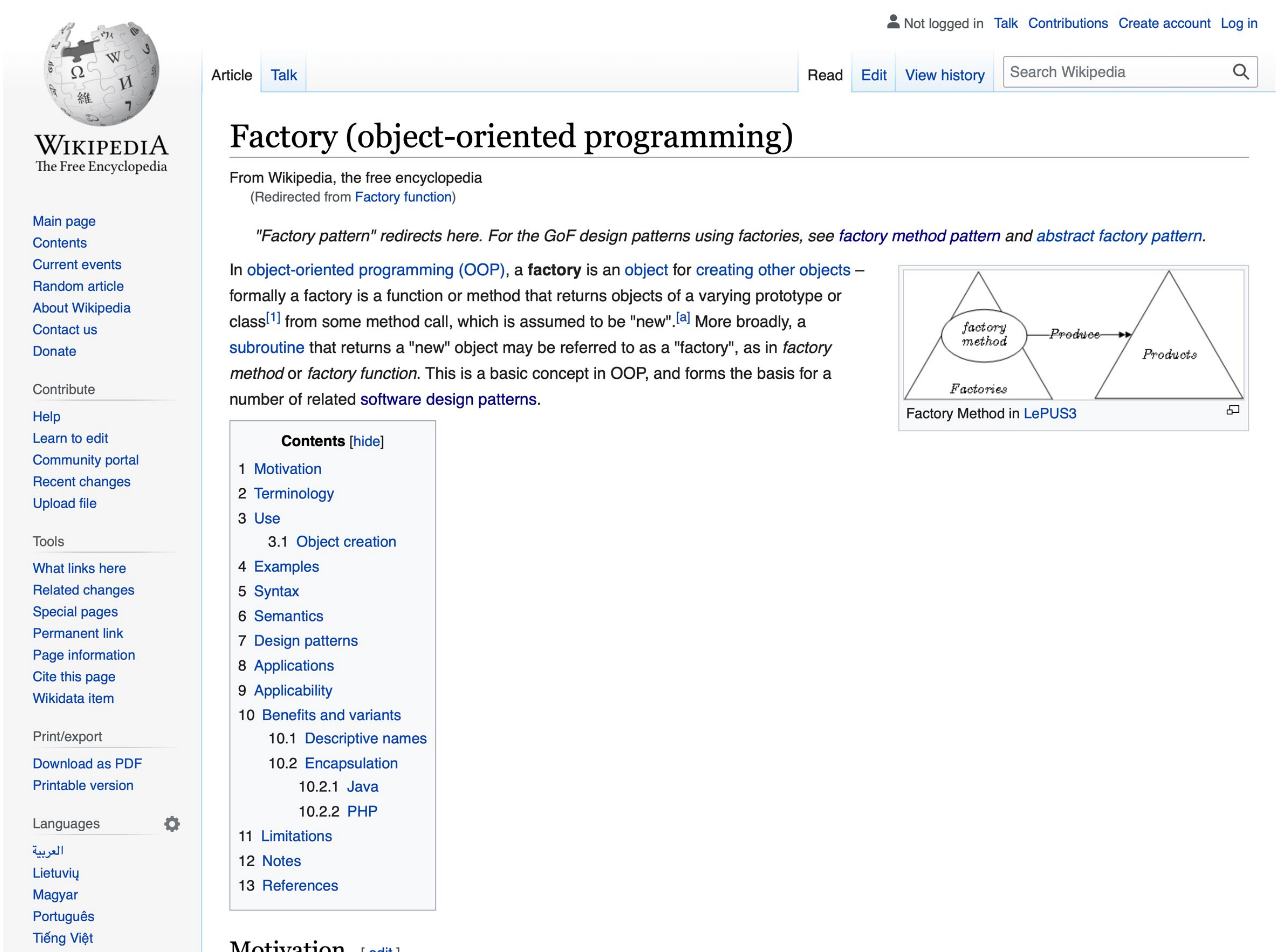
- ~~… improves exception safety;~~
- ~~… fulfills the Single-Responsibility Principle (SRP);~~
- … is a factory function.

`std::make_unique()` …

- ~~… improves exception safety;~~
- ~~… fulfills the Single-Responsibility Principle (SRP);~~
- … is a factory function.

Core Guideline C.50: Use a factory function if you need “virtual behavior” during initialization.

```
std::unique_ptr<MyThing> createMyThing()
{
   auto tmp{ std::make_unique<MyThing>() };
   tmp->init();  // Virtual function call
   return tmp;
}
```



Not logged in Talk Contributions Create account Log in

Article Talk Read Edit View history Search Wikipedia

Main page
Contents
Current events
Random article
About Wikipedia
Contact us
Donate

Contribute
Help
Learn to edit
Community portal
Recent changes
Upload file

Tools
What links here
Related changes
Special pages
Permanent link
Page information
Cite this page
Wikidata item

Print/export
Download as PDF
Printable version

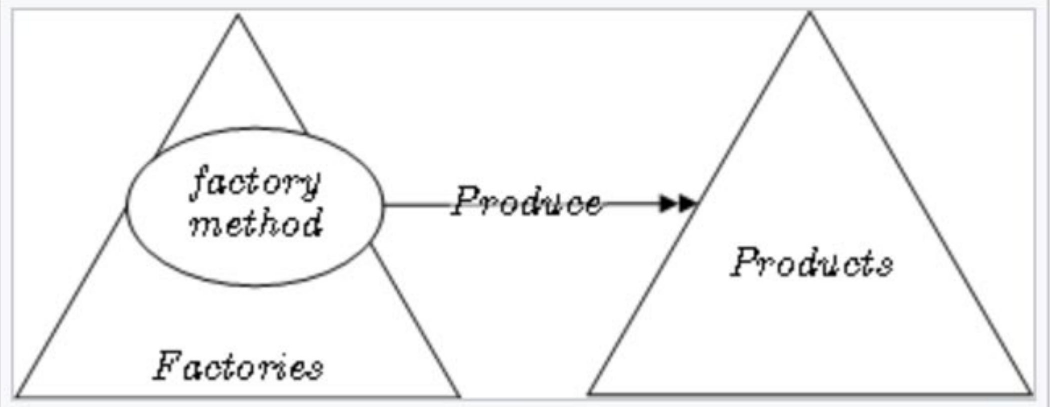
Languages
العربية
Lietuvių
Magyar
Português
Tiếng Việt

Factory (object-oriented programming)

From Wikipedia, the free encyclopedia
(Redirected from Factory function)

"Factory pattern" redirects here. For the GoF design patterns using factories, see factory method pattern and abstract factory pattern.

In object-oriented programming (OOP), a **factory** is an object for creating other objects – formally a factory is a function or method that returns objects of a varying prototype or class[1] from some method call, which is assumed to be "new".[a] More broadly, a subroutine that returns a "new" object may be referred to as a "factory", as in *factory method* or *factory function*. This is a basic concept in OOP, and forms the basis for a number of related software design patterns.



Factory Method in LePUS3

Contents [hide]

Motivation [edit]

Download as PDF
Printable version

Languages

العربية
Lietuvių
Magyar
Português
Tiếng Việt
中文

Edit links

Motivation [edit]

In class-based programming, a factory is an abstraction of a constructor of a class, while in prototype-based programming a factory is an abstraction of a prototype object. A constructor is concrete in that it creates objects as instances of a single class, and by a specified process (class instantiation), while a factory can create objects by instantiating various classes, or by using other allocation schemes such as an object pool. A prototype object is concrete in that it is used to create objects by being cloned, while a factory can create objects by cloning various prototypes, or by other allocation schemes.

Factories may be invoked in various ways, most often a method call (a *factory method*), sometimes by being called as a function if the factory is a function object (a *factory function*). In some languages factories are generalizations of constructors, meaning constructors are themselves factories and these are invoked in the same way. In other languages factories and constructors are invoked differently, for example using the keyword `new` to invoke constructors but an ordinary method call to invoke factories; in these languages factories are an abstraction of constructors but not strictly a generalization, as constructors are not themselves factories.

Terminology [edit]

Terminology differs as to whether the concept of a factory is itself a design pattern – in the seminal book *Design Patterns* there is no "factory pattern", but instead two patterns (factory method pattern and abstract factory pattern) that use factories. Some sources refer to the concept as the **factory pattern**,[2][3] while others consider the concept itself a programming idiom,[4] reserving the term "factory pattern" or "factory patterns" to more complicated patterns that use factories, most often the factory method pattern; in this context, the concept of a factory itself may be referred to as a **simple factory.**[4] In other contexts, particularly the Python language, "factory" itself is used, as in this article.[5] More broadly, "factory" may be applied not just to an object that returns objects from some method call, but to a *subroutine* that returns objects, as in a *factory function* (even if functions are not objects) or *factory method*.[6] Because in many languages factories are invoked by calling a method, the general concept of a factory is often confused with the specific factory method pattern design pattern.

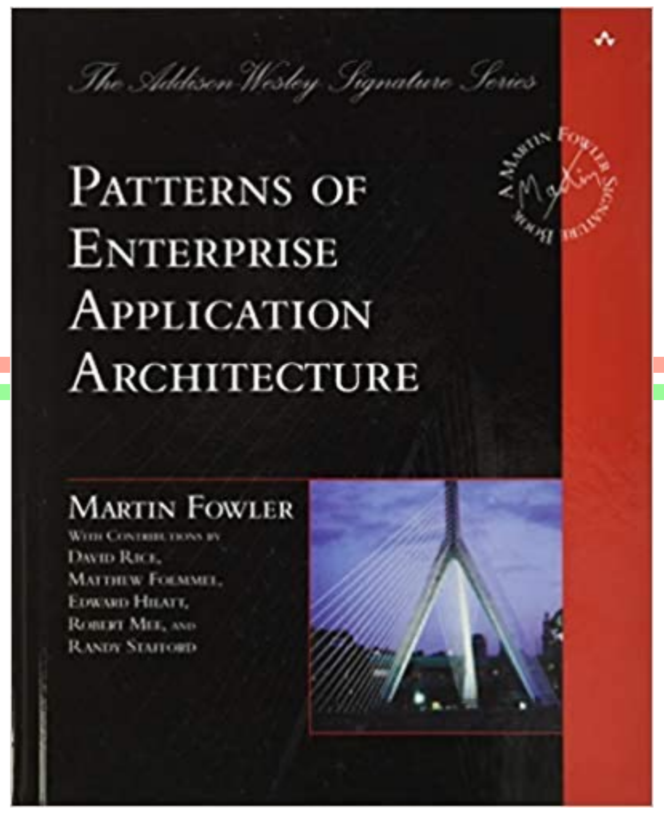
Use [edit]

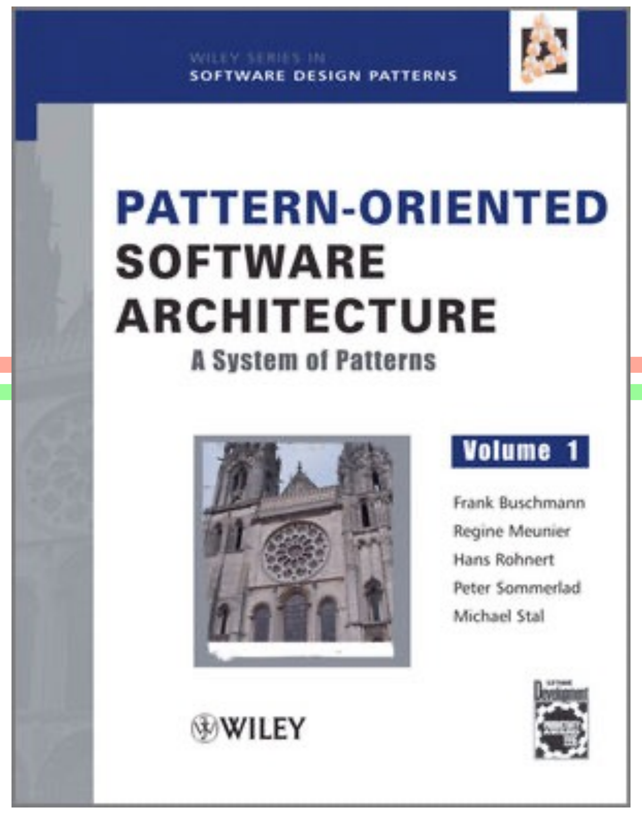
OOP provides polymorphism on object *use* by method dispatch, formally subtype polymorphism via single dispatch determined by the type of the object on which the method is called. However, this does not work for constructors, as constructors *create* an object of some type, rather than *use* an existing object. More concretely, when a constructor is called, there is no object yet on which to dispatch.[b]

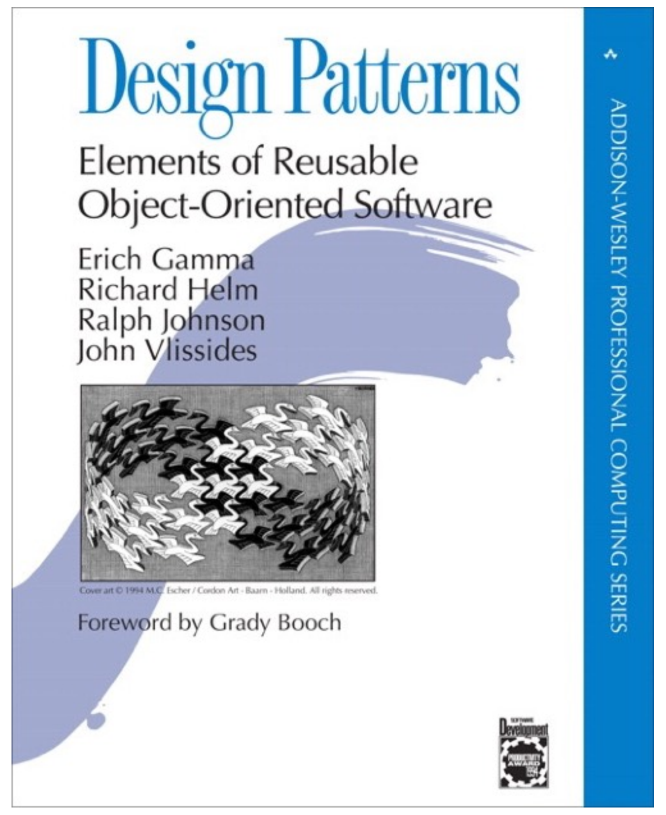
Using factories instead of constructors or prototypes allows one to use polymorphism for object creation, not only object use. Specifically, using factories provides encapsulation, and means the code is not tied to specific classes or objects, and thus the class hierarchy or

Architecture

- How are big entities depending on each other?
- Design decisions that are hard to change
- Architectural patterns
- Examples:
  - Client-Server Architecture
  - Micro-Services
  - MVC, …


The Addison-Wesley Signature Series
Patterns of Enterprise Application Architecture
Martin Fowler




Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
Addison-Wesley Professional Computing Series

Design

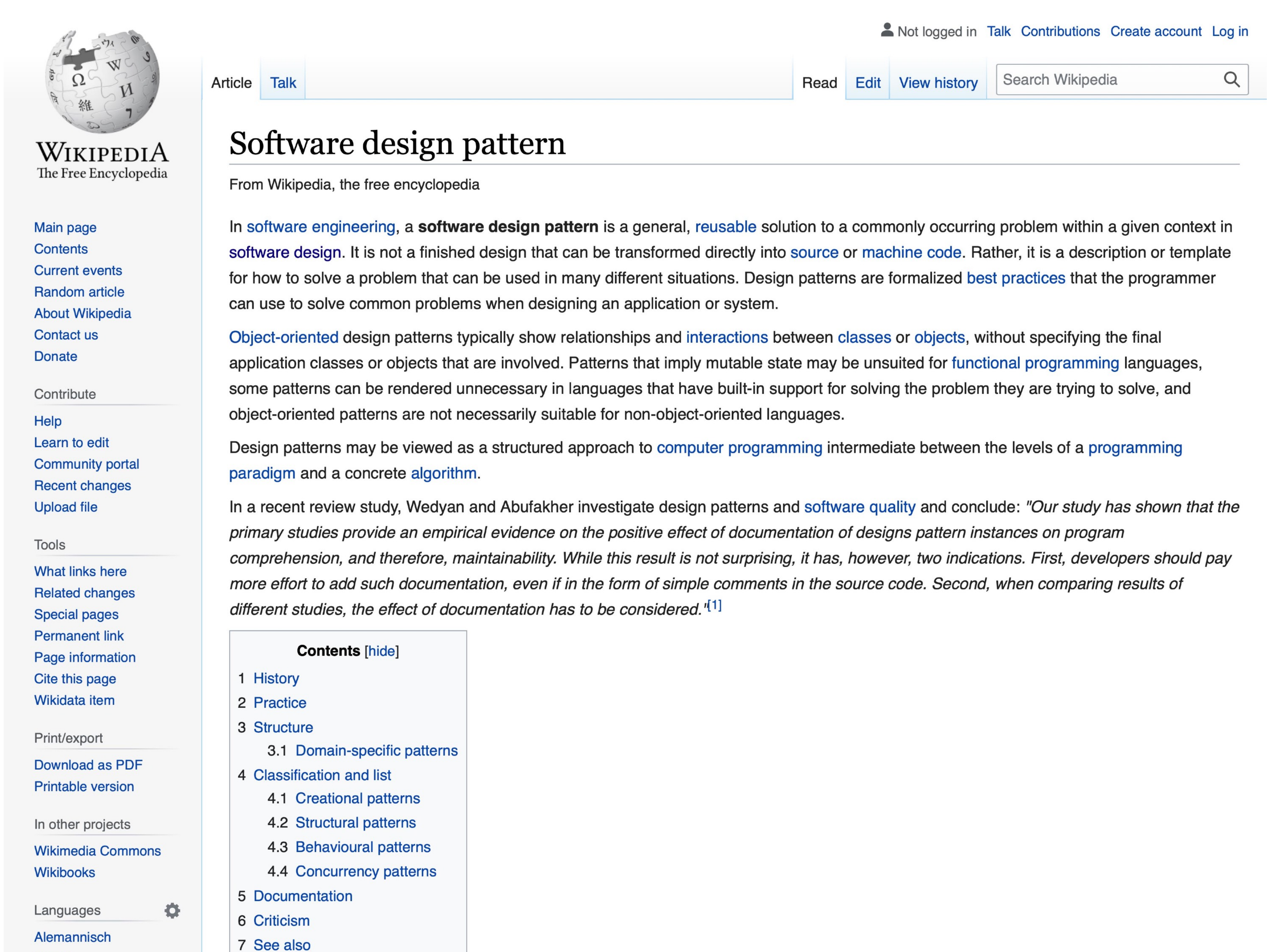
- How are small entities depending on each other?
- Design decisions that are easier to change
- Design patterns
- Examples:
  - GoF Patterns: Visitor, Strategy, Observer, …
  - External Polymorphism
  - …

Idioms

- NVI Idiom (Template Method Design Pattern)
- Pimpl Idiom (Bridge Design Pattern)
- Temporary-Swap Idiom
- RAII Idiom
- `enable_if`
- Factory Function

Implementation Details

- How is a design implemented?
- Which features are used?
- Implementation patterns
- Examples:
  - `new`, `malloc`, …
  - class vs. struct, lambda, …
  - …



Software design pattern

From Wikipedia, the free encyclopedia

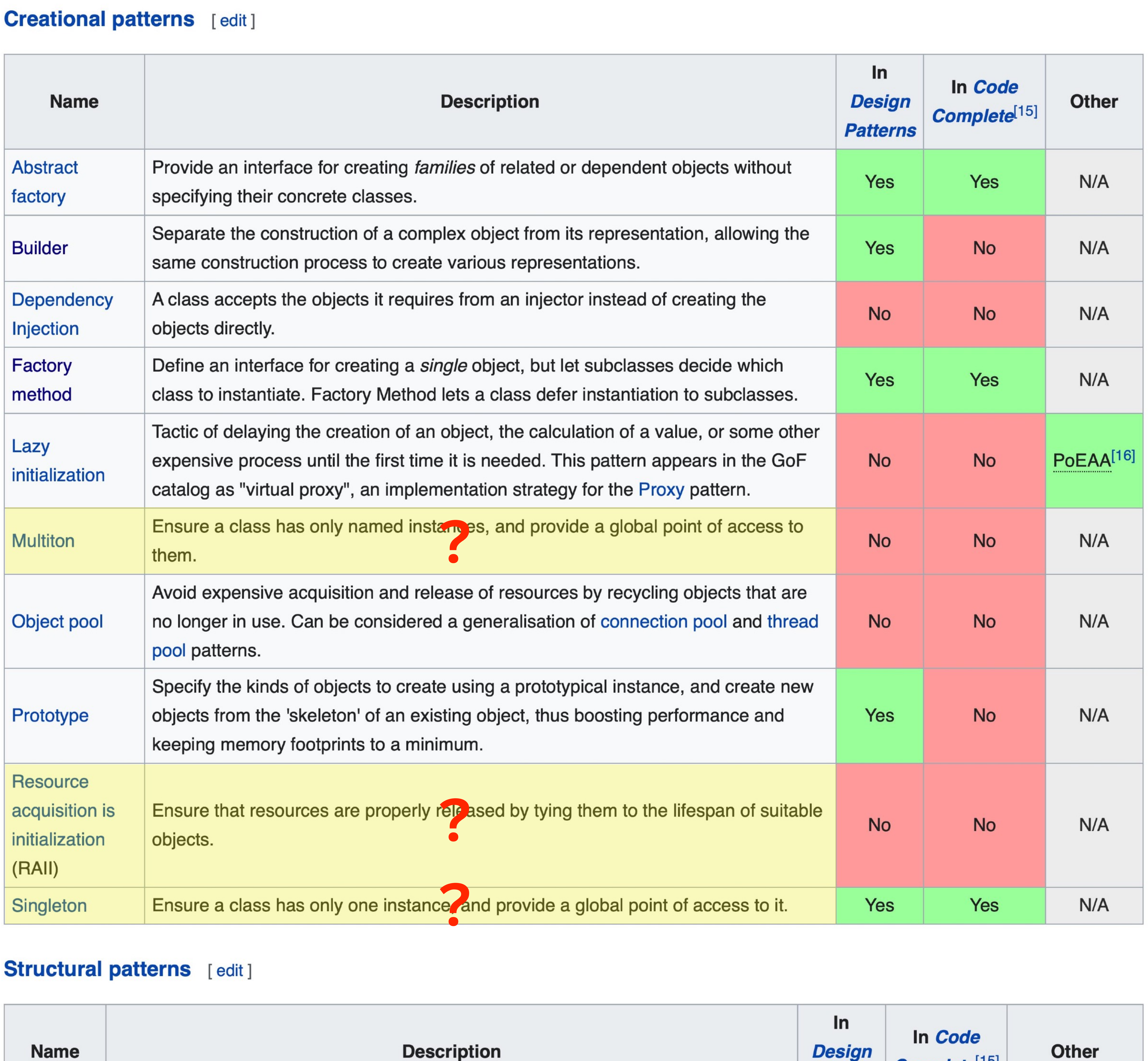
In software engineering, a **software design pattern** is a general, reusable solution to a commonly occurring problem within a given context in software design. It is not a finished design that can be transformed directly into source or machine code. Rather, it is a description or template for how to solve a problem that can be used in many different situations. Design patterns are formalized best practices that the programmer can use to solve common problems when designing an application or system.

Object-oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved. Patterns that imply mutable state may be unsuited for functional programming languages, some patterns can be rendered unnecessary in languages that have built-in support for solving the problem they are trying to solve, and object-oriented patterns are not necessarily suitable for non-object-oriented languages.

Design patterns may be viewed as a structured approach to computer programming intermediate between the levels of a programming paradigm and a concrete algorithm.

In a recent review study, Wedyan and Abufakher investigate design patterns and software quality and conclude: *"Our study has shown that the primary studies provide an empirical evidence on the positive effect of documentation of designs pattern instances on program comprehension, and therefore, maintainability. While this result is not surprising, it has, however, two indications. First, developers should pay more effort to add such documentation, even if in the form of simple comments in the source code. Second, when comparing results of different studies, the effect of documentation has to be considered."*[1]

Contents [hide]



Creational patterns [edit]

Name	Description	In *Design Patterns*	In *Code Complete*[15]	Other
Abstract factory	Provide an interface for creating *families* of related or dependent objects without specifying their concrete classes.	Yes	Yes	N/A
Builder	Separate the construction of a complex object from its representation, allowing the same construction process to create various representations.	Yes	No	N/A
Dependency Injection	A class accepts the objects it requires from an injector instead of creating the objects directly.	No	No	N/A
Factory method	Define an interface for creating a *single* object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.	Yes	Yes	N/A
Lazy initialization	Tactic of delaying the creation of an object, the calculation of a value, or some other expensive process until the first time it is needed. This pattern appears in the GoF catalog as "virtual proxy", an implementation strategy for the Proxy pattern.	No	No	PoEAA[16]
Multiton	Ensure a class has only named instances, and provide a global point of access to them.	No	No	N/A
Object pool	Avoid expensive acquisition and release of resources by recycling objects that are no longer in use. Can be considered a generalisation of connection pool and thread pool patterns.	No	No	N/A
Prototype	Specify the kinds of objects to create using a prototypical instance, and create new objects from the 'skeleton' of an existing object, thus boosting performance and keeping memory footprints to a minimum.	Yes	No	N/A
Resource acquisition is initialization (RAII)	Ensure that resources are properly released by tying them to the lifespan of suitable objects.	No	No	N/A
Singleton	Ensure a class has only one instance, and provide a global point of access to it.	Yes	Yes	N/A

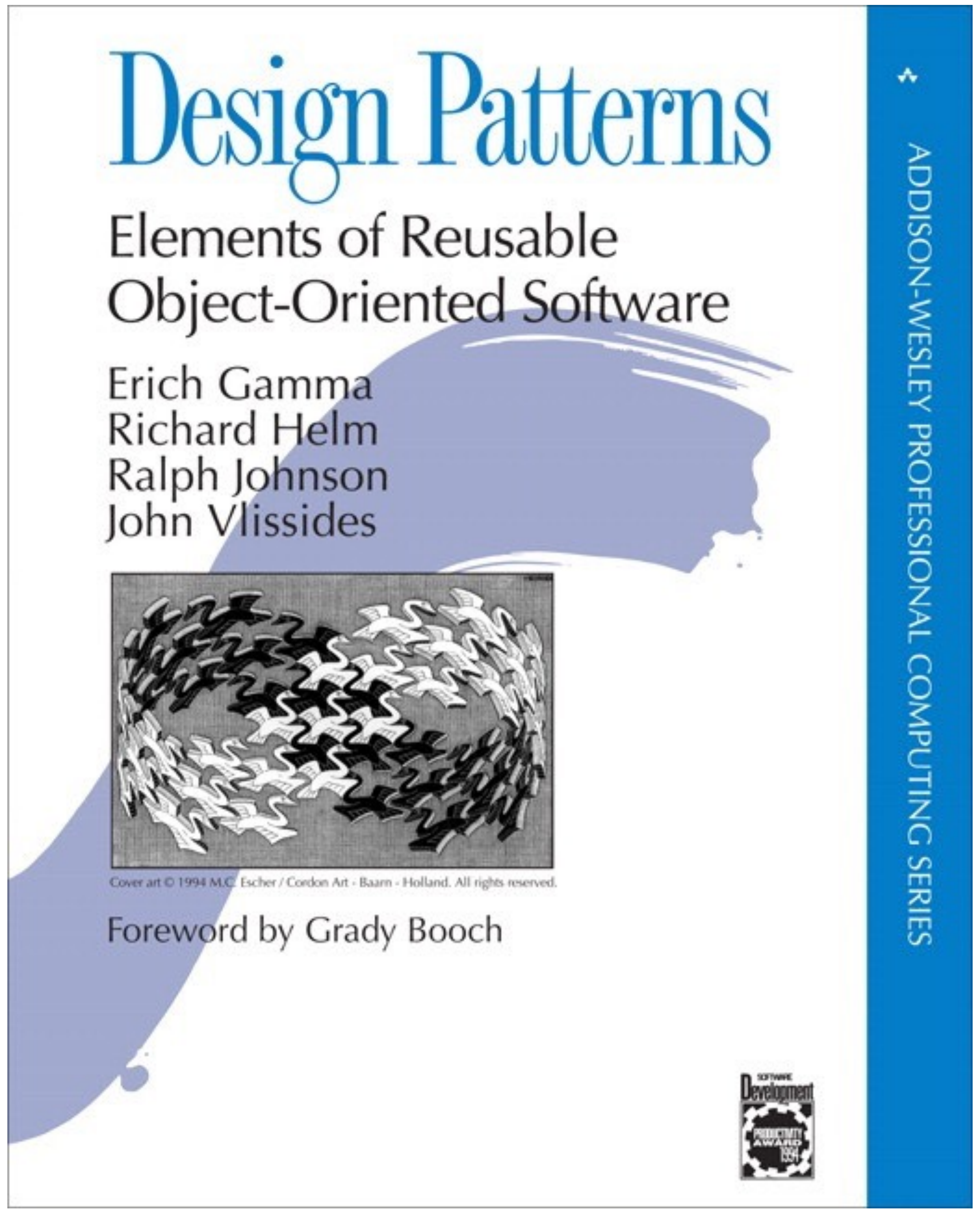
Structural patterns [edit]

Name	Description	In *Design*	In *Code*	Other

Behavioural patterns [edit]

Name	Description	In *Design Patterns*	In *Code Complete*[15]	Other
Blackboard	Artificial intelligence pattern for combining disparate sources of data (see blackboard system)	No	No	N/A
Chain of responsibility	Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.	Yes	No	N/A
Command	Encapsulate a request as an object, thereby allowing for the parameterization of clients with different requests, and the queuing or logging of requests. It also allows for the support of undoable operations.	Yes	No	N/A
Interpreter	Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.	Yes	No	N/A
Iterator	Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.	Yes	Yes	N/A
Mediator	Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it allows their interaction to vary independently.	Yes	No	N/A
Memento	Without violating encapsulation, capture and externalize an object's internal state allowing the object to be restored to this state later.	Yes	No	N/A
Null object	Avoid null references by providing a default object.	No	No	N/A
Observer or Publish/subscribe	Define a one-to-many dependency between objects where a state change in one object results in all its dependents being notified and updated automatically.	Yes	Yes	N/A
Servant	Define common functionality for a group of classes. The servant pattern is also frequently called helper class or utility class implementation for a given set of classes. The helper classes generally have no objects hence they have all static methods that act upon different kinds of class objects.	No	No	N/A
Specification	Recombinable business logic in a Boolean fashion.	No	No	N/A
State	Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.	Yes	No	N/A
	Define a family of algorithms, encapsulate each one, and make them interchangeable.			

The Classic Factory Method Design Pattern



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

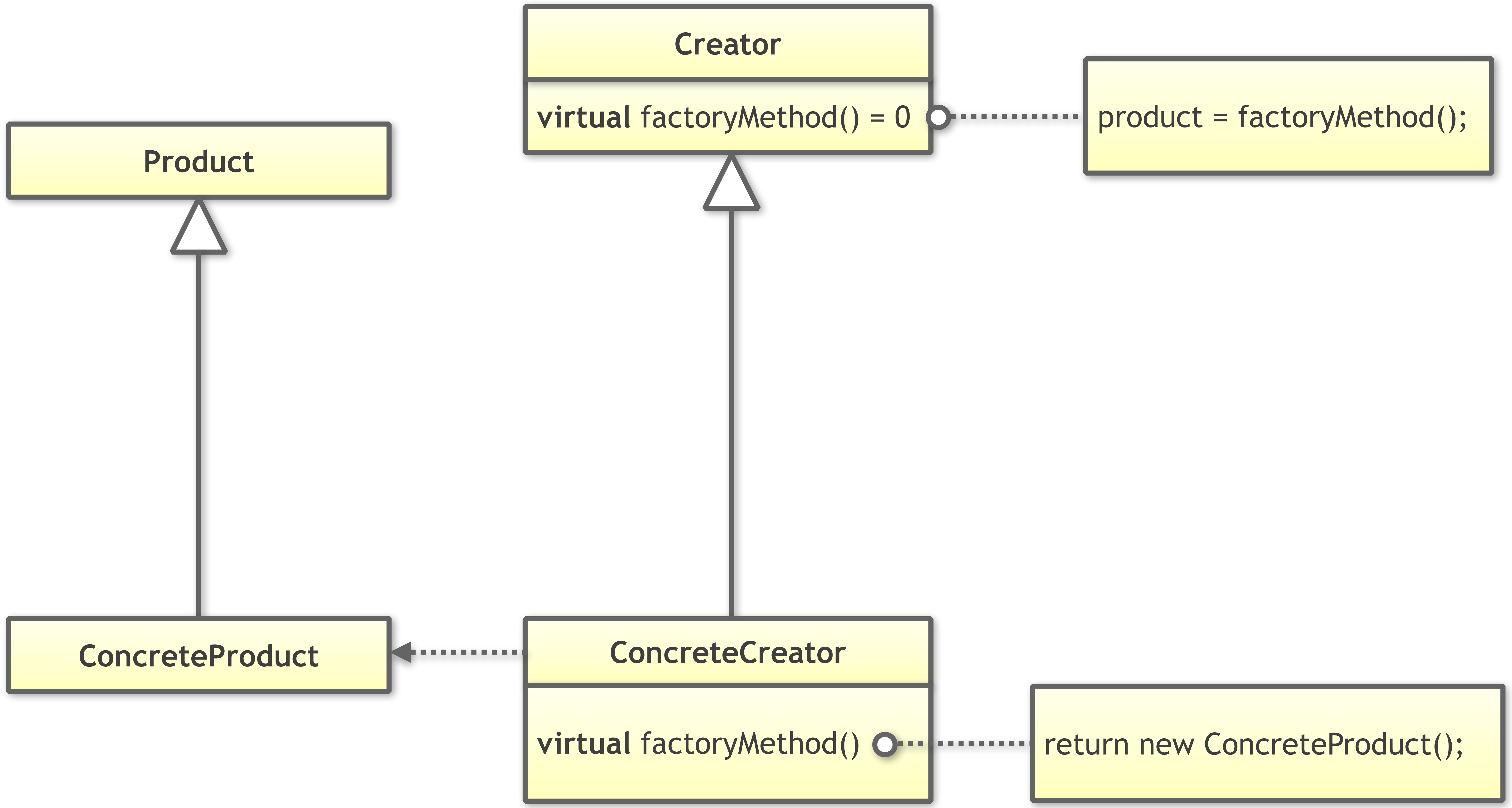
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

The Classic Factory Method Design Pattern

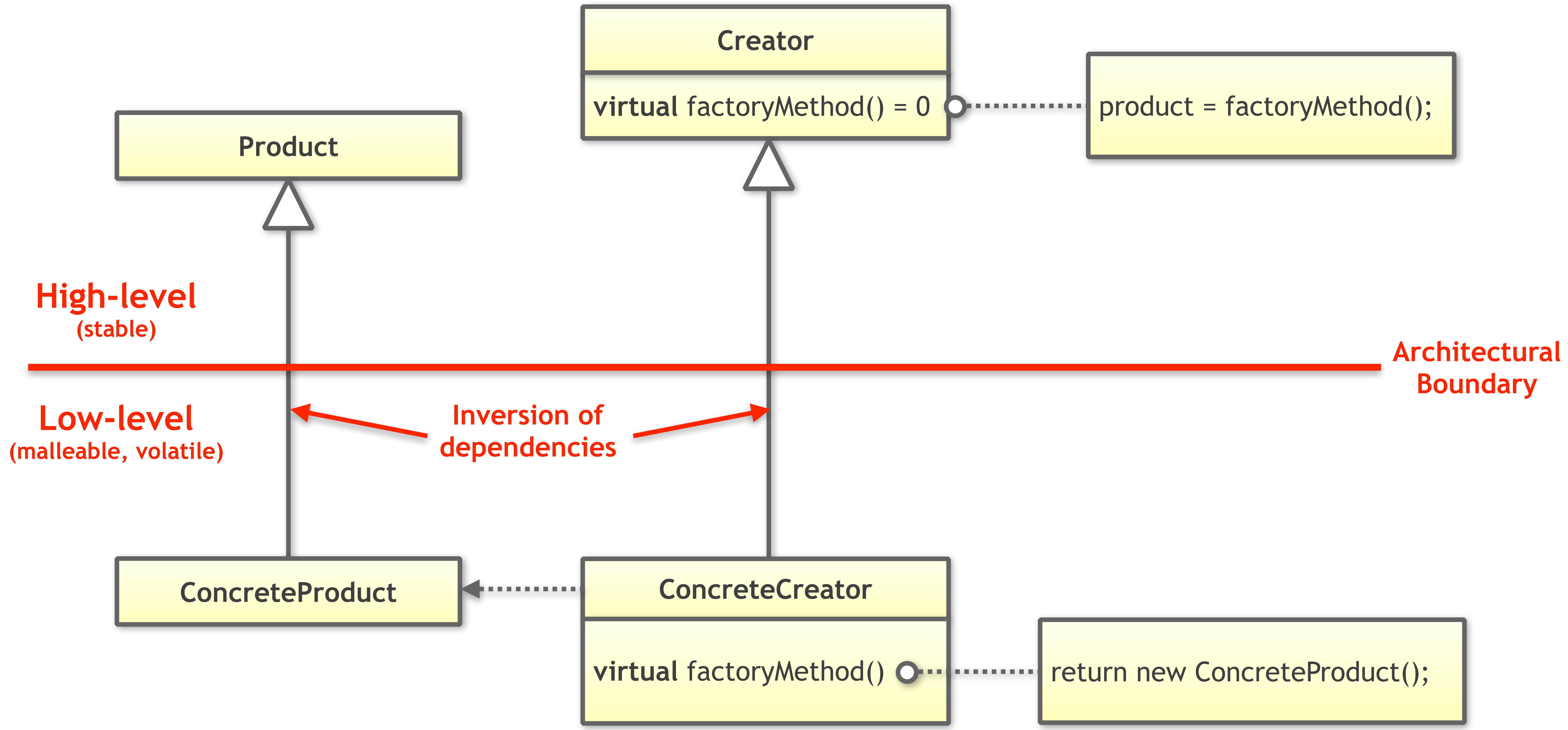
“Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.”

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

The Classic Factory Method Design Pattern


Creator
virtual factoryMethod() = 0
product = factoryMethod();
Product
ConcreteProduct
ConcreteCreator
virtual factoryMethod()
return new ConcreteProduct();

The Classic Factory Method Design Pattern



Guideline: The purpose of a design pattern is to introduce a fitting abstraction for a well known problem.

Guideline: The name of a design pattern conveys the intent of the abstraction.

Guideline: `std::make_unique()` is an implementation pattern, not a design pattern.

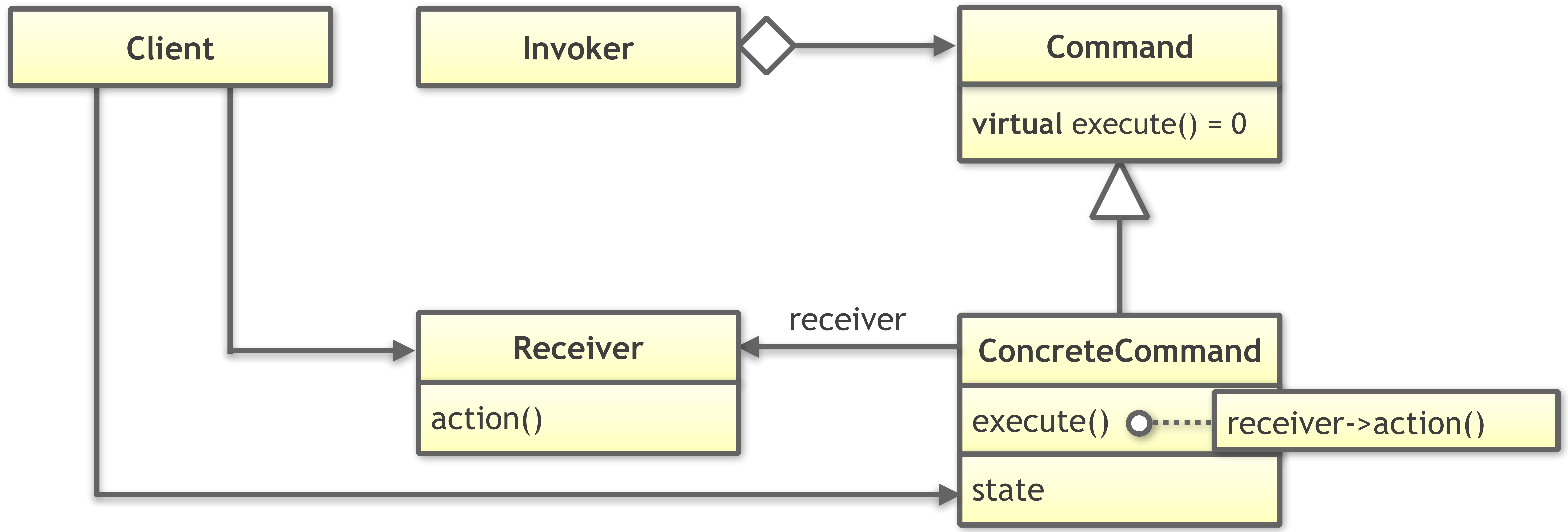
`std::make_unique()` …

- ~~… improves exception safety;~~
- ~~… fulfills the Single-Responsibility Principle (SRP);~~
- … is a factory function, but not a design pattern.

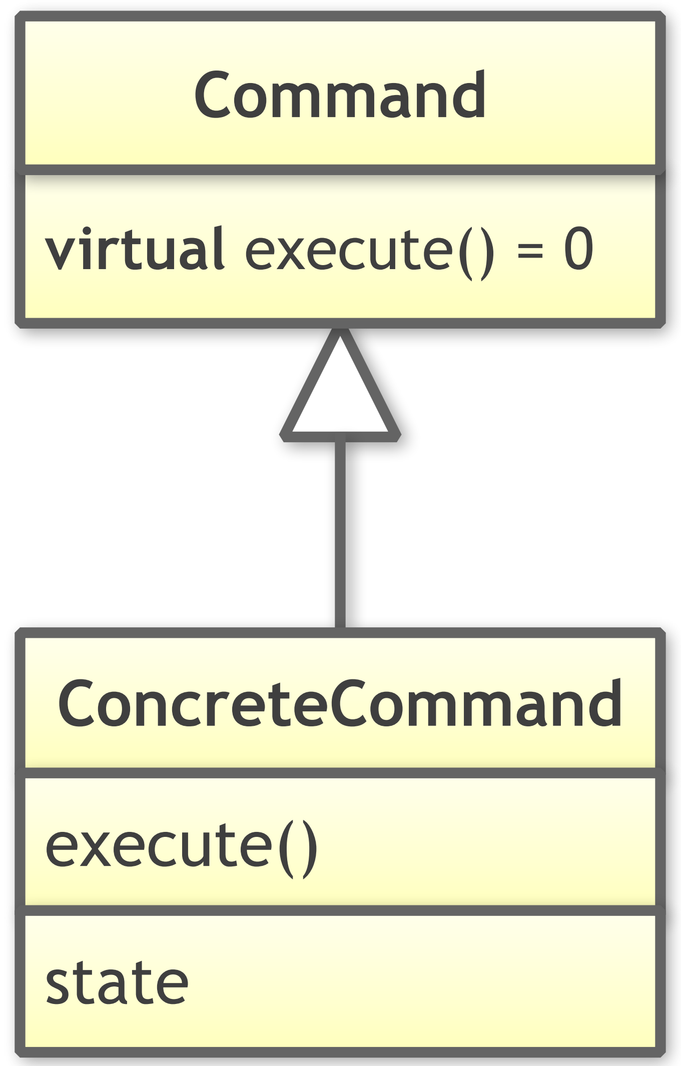
Common misconceptions about design patterns:

- Design patterns are limited to runtime polymorphism;
- Design patterns are limited to OO programming;
- Design patterns are language specific idioms;
- Design patterns can be recognized by their structure.

The Classic Command Pattern



The Classic Command Pattern



An Example from the Standard Library

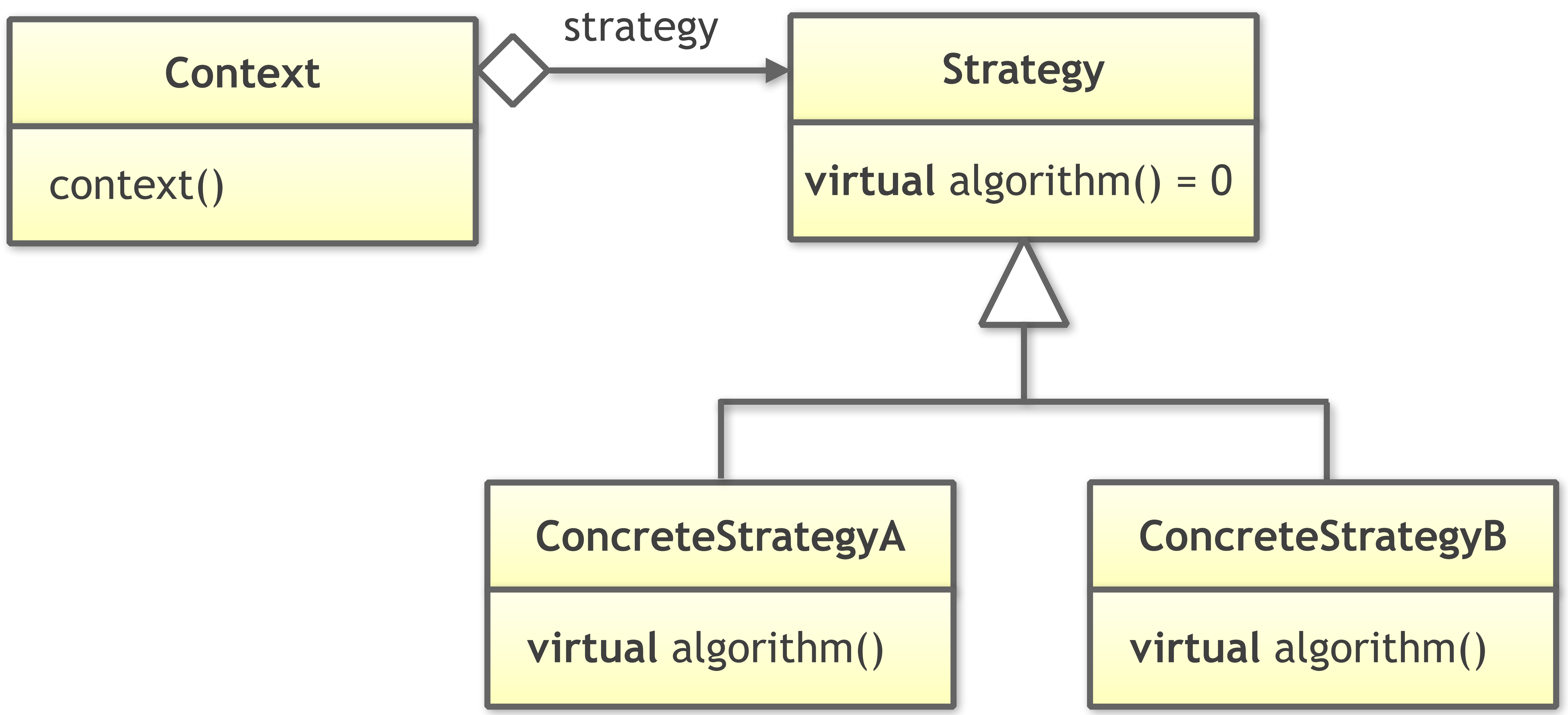
Alternatively we could use static polymorphism:

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

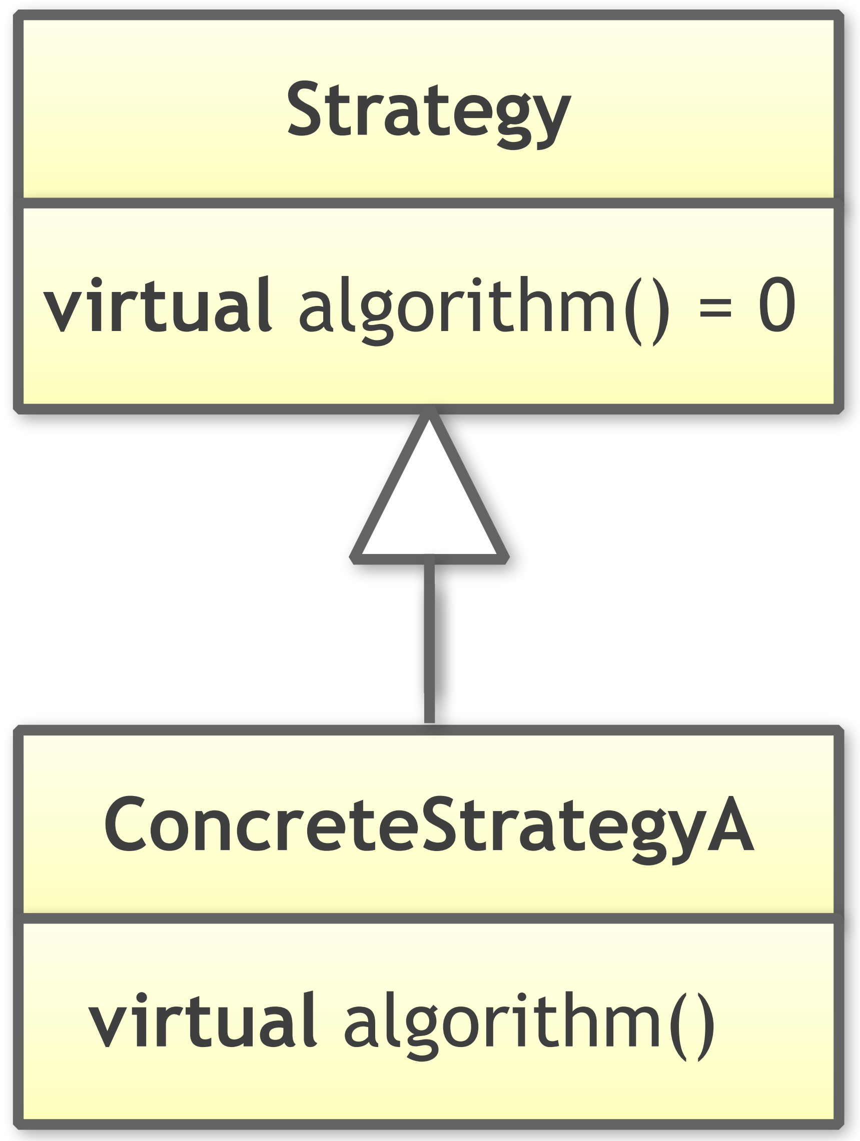
This form of the command pattern is used in the standard library:

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

std::for_each( begin(numbers), end(numbers)
             , [](int& i){ i*=10; } );
```

The Classic Strategy Design Pattern



The Classic Strategy Design Pattern



An Example from the Standard Library

Alternatively we could use static polymorphism:

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

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

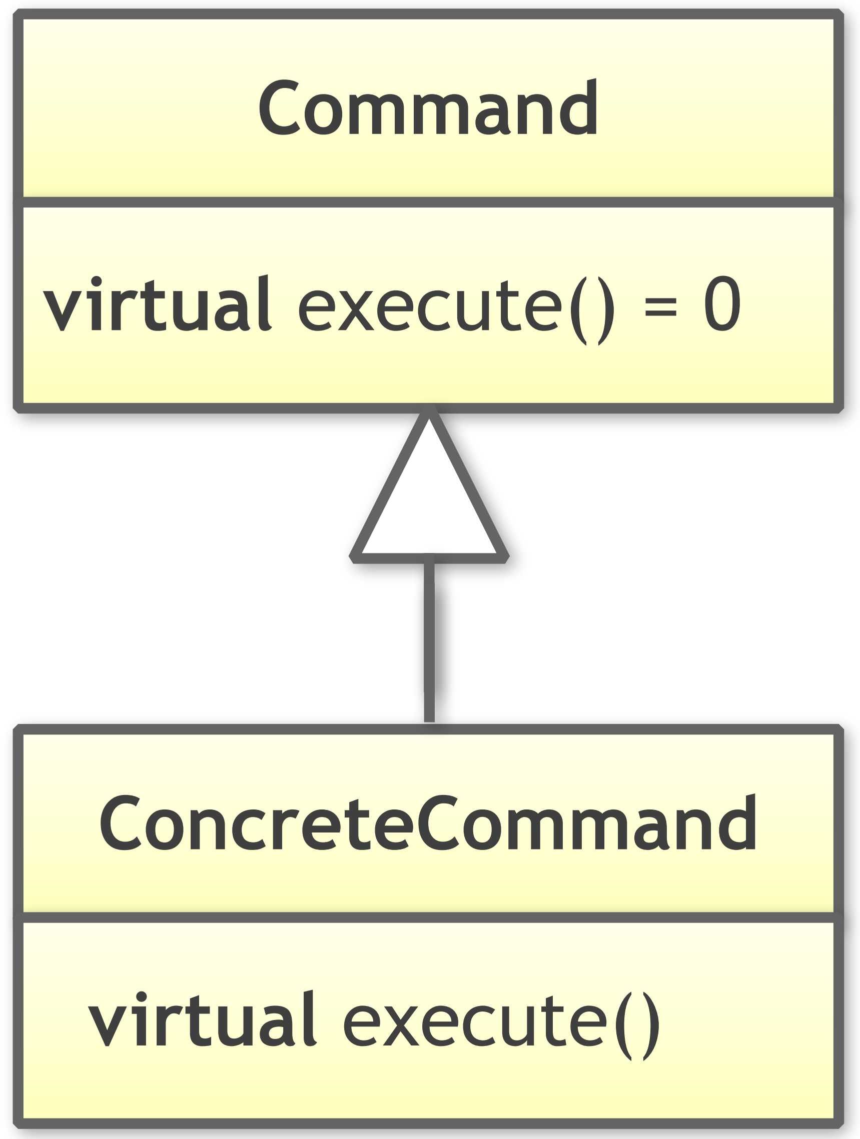
```
std::vector<int> numbers{ 1, 2, 3, 4, 5, 6, 7 };

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

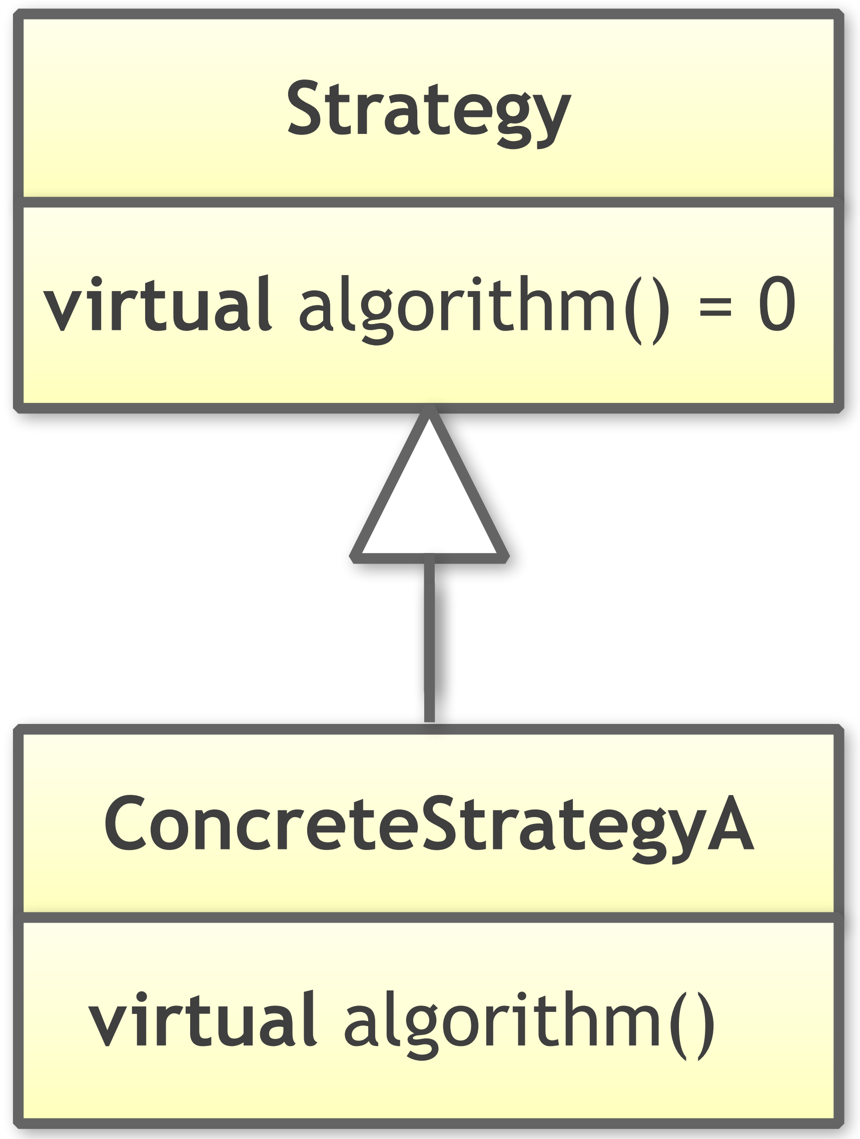
Wait a second: Isn’t this the Command design pattern? 🤔

An Example from the Standard Library

Well, it depends …

Command vs. Strategy





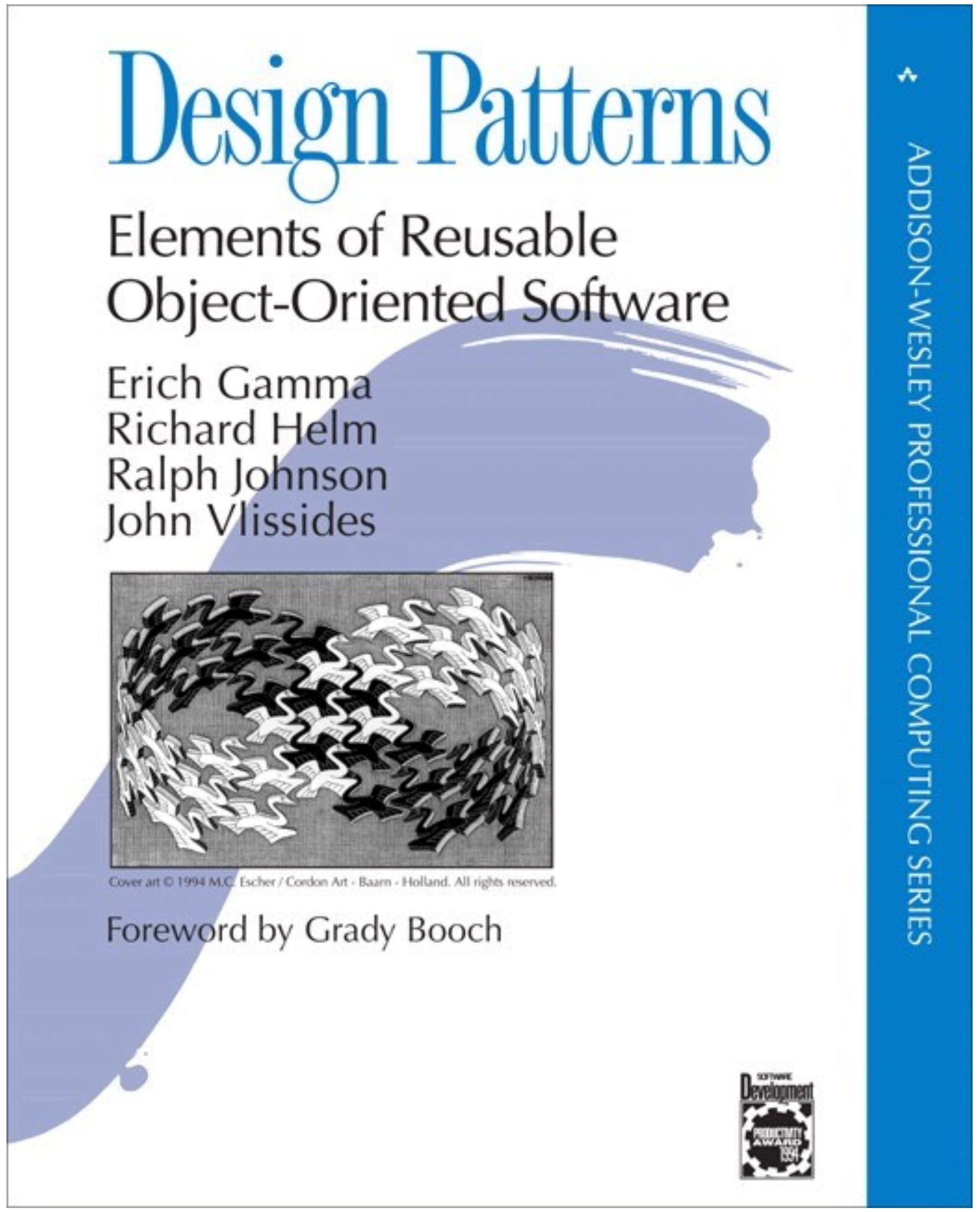
Command vs. Strategy

The Command design pattern:

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

std::for_each( begin(numbers), end(numbers)
             , [](int& i){ i*=10; } );
```

The Strategy design pattern:

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

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



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

Software Development Productivity Award 1994

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

The GoF’s Explanation

The Command Design Pattern

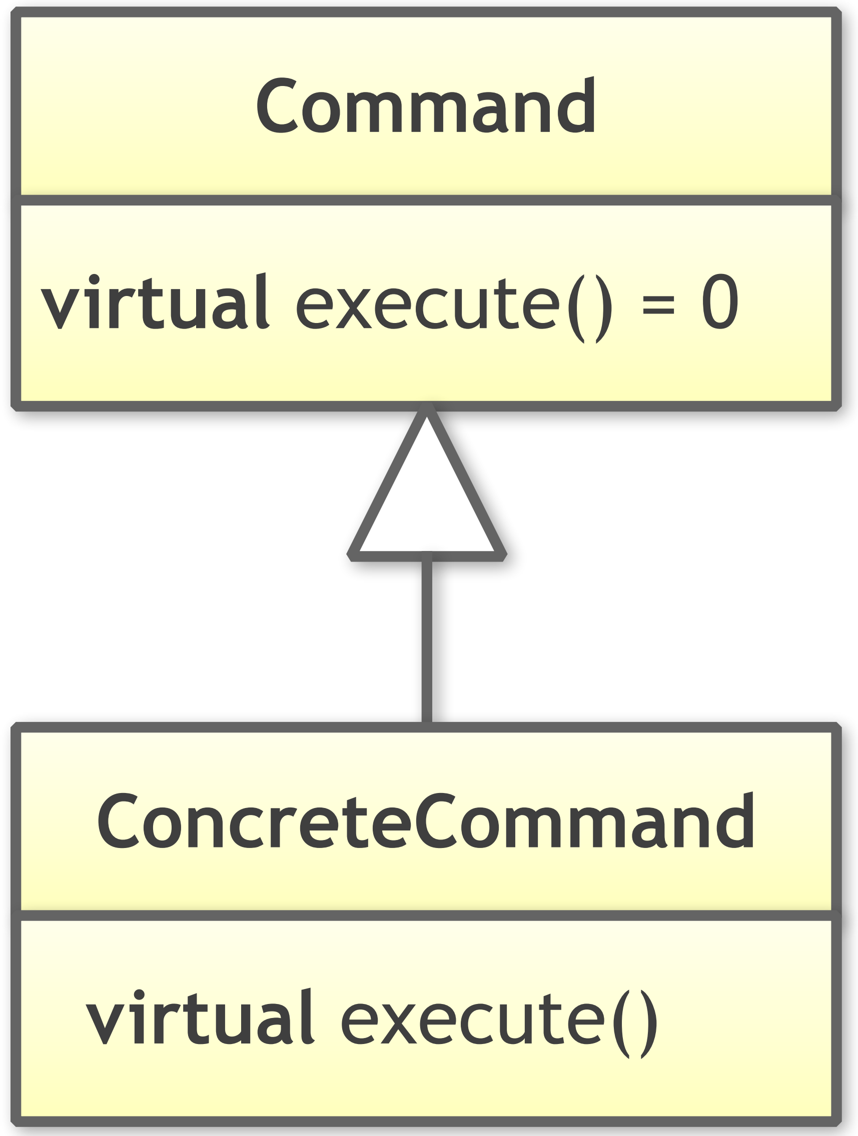
”Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.”

The Strategy Design Pattern

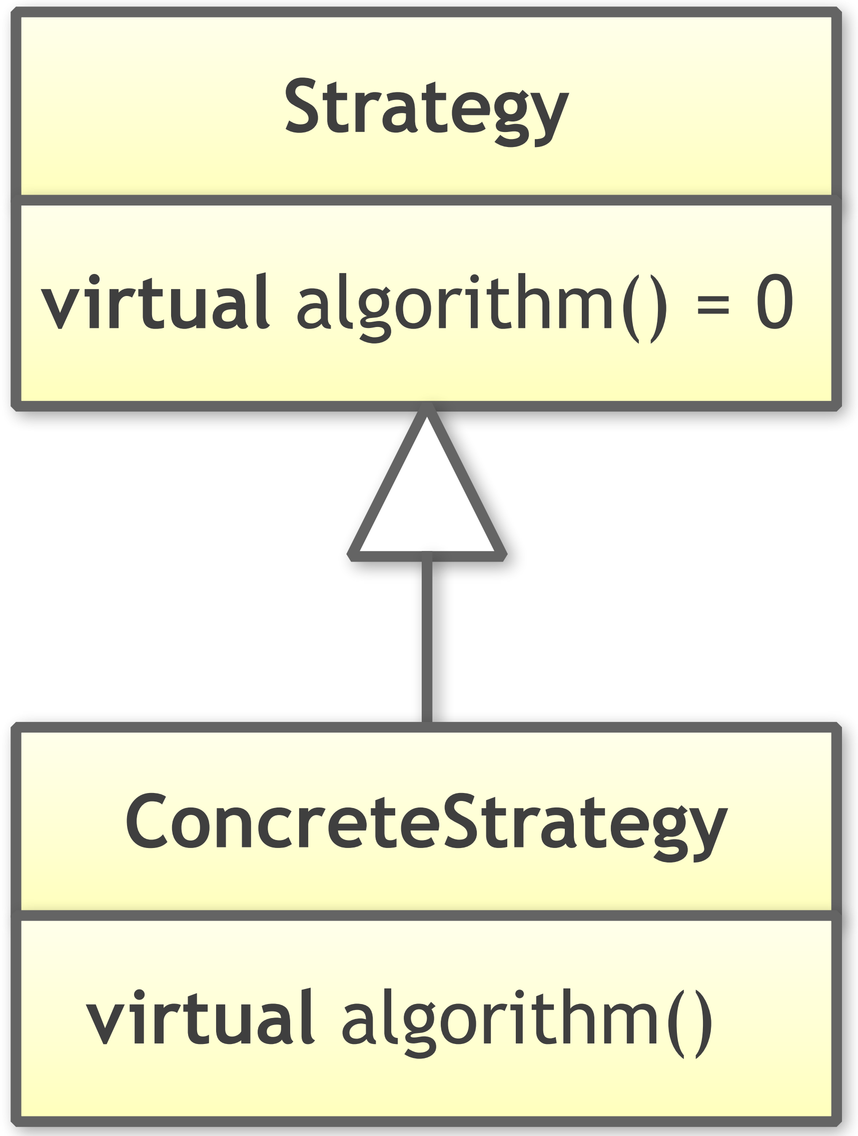
”Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.”

Command vs. Strategy

Specify **what** should be done
➜ Command design pattern



Specify **how** something should be done
➜ Strategy design pattern



Command vs. Strategy

What should I do with each element? ➜ Command design pattern

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

std::for_each( begin(numbers), end(numbers)
             , [](int& i){ i*=10; } );
```

How should I accumulate the elements? ➜ Strategy design pattern

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

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

Guidelines

Guideline: The intent of the Command design pattern is to specify **what** should be done.

Guideline: The intent of the Strategy design pattern is to specify **how** something should be done.

Guideline: Remember that the difference between design patterns often is not the structure, but the intent.

Guideline: Remember that design patterns are neither limited to object-oriented programming, nor dynamic polymorphism.

Guidelines

Guideline: Consider to include the name of the design pattern into the class name to help to convey the intent.

Command vs. Strategy

What should I do with each element? ➜ Command design pattern

```
template< class InputIt, class UnaryFunction >
constexpr UnaryFunction
   for_each( InputIt first, InputIt last, UnaryFunction f );
```

How should I accumulate the elements? ➜ Strategy design pattern

```
template< class InputIt, class T, class BinaryOperation >
constexpr T accumulate( InputIt first, InputIt last, T init,
                        BinaryOperation op );
```

Command vs. Strategy

What should I do with each element? ➜ Command design pattern

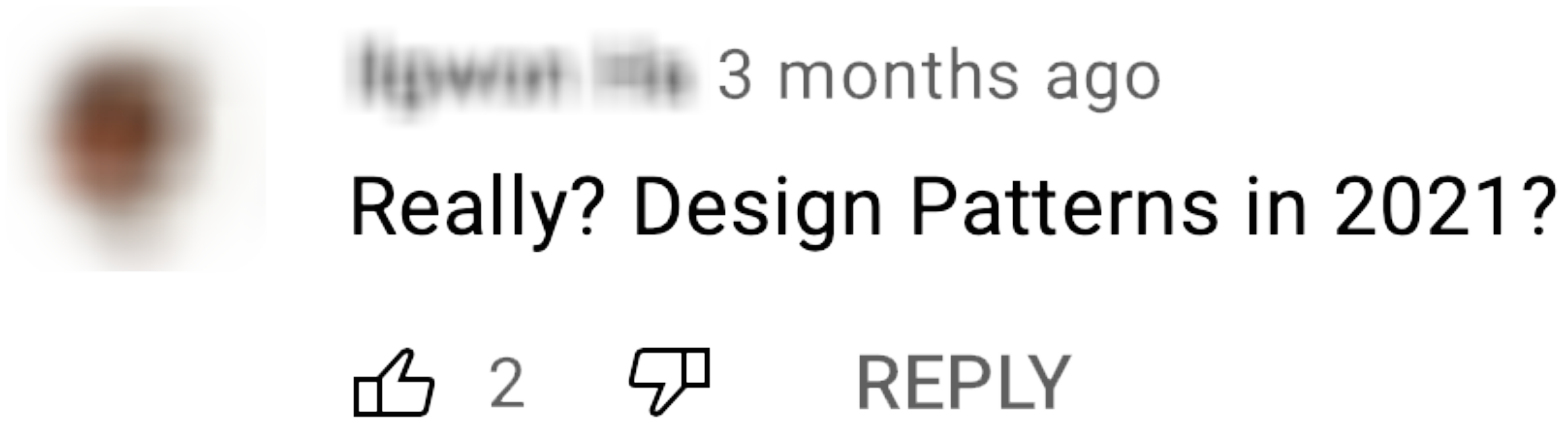
```
template< class InputIt, class UnaryCommand >
constexpr UnaryCommand
   for_each( InputIt first, InputIt last, UnaryCommand f );
```

How should I accumulate the elements? ➜ Strategy design pattern

```
template< class InputIt, class T, class BinaryReductionStrategy >
constexpr T accumulate( InputIt first, InputIt last, T init,
                        BinaryReductionStrategy op );
```

Further misconceptions about design patterns:

- Design patterns are outdated;
- Design patterns have become obsolete.



[illegible] 3 months ago

Really? Design Patterns in 2021?

“Design patterns are everywhere!”

(Klaus Iglberger)

Challenge:

Name as many **different design patterns** as possible that are used within the C++ standard library!

You have **20 seconds…**

Go!

1. The Strategy Design Pattern

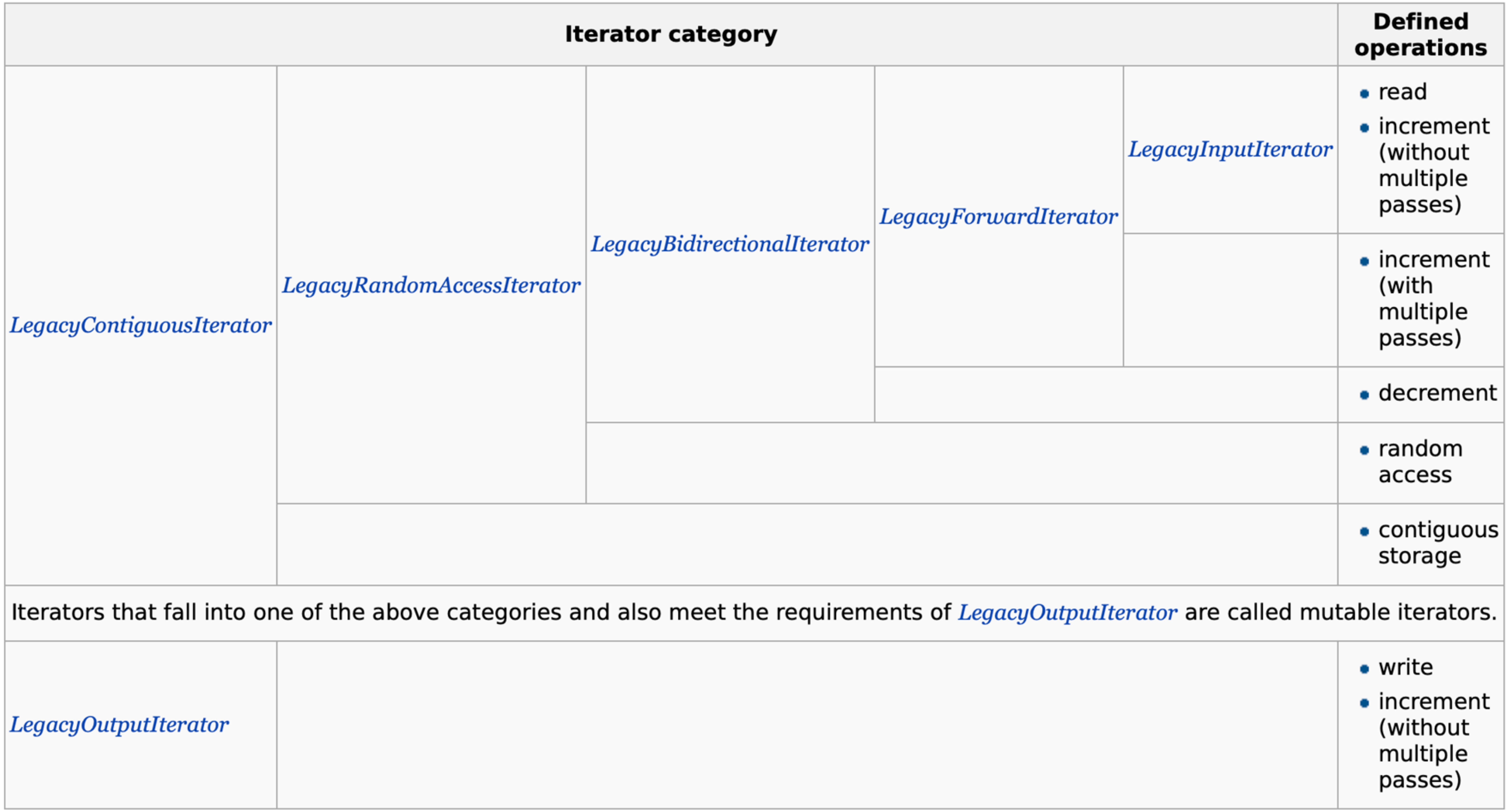
```
template< class InputIt, class T, class BinaryOperation >
constexpr T accumulate( InputIt first, InputIt last, T init,
                        BinaryOperation op );


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


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


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

2. The Command Design Pattern

```
template< class InputIt, class UnaryFunction >
constexpr UnaryFunction
   for_each( InputIt first, InputIt last, UnaryFunction f );
```

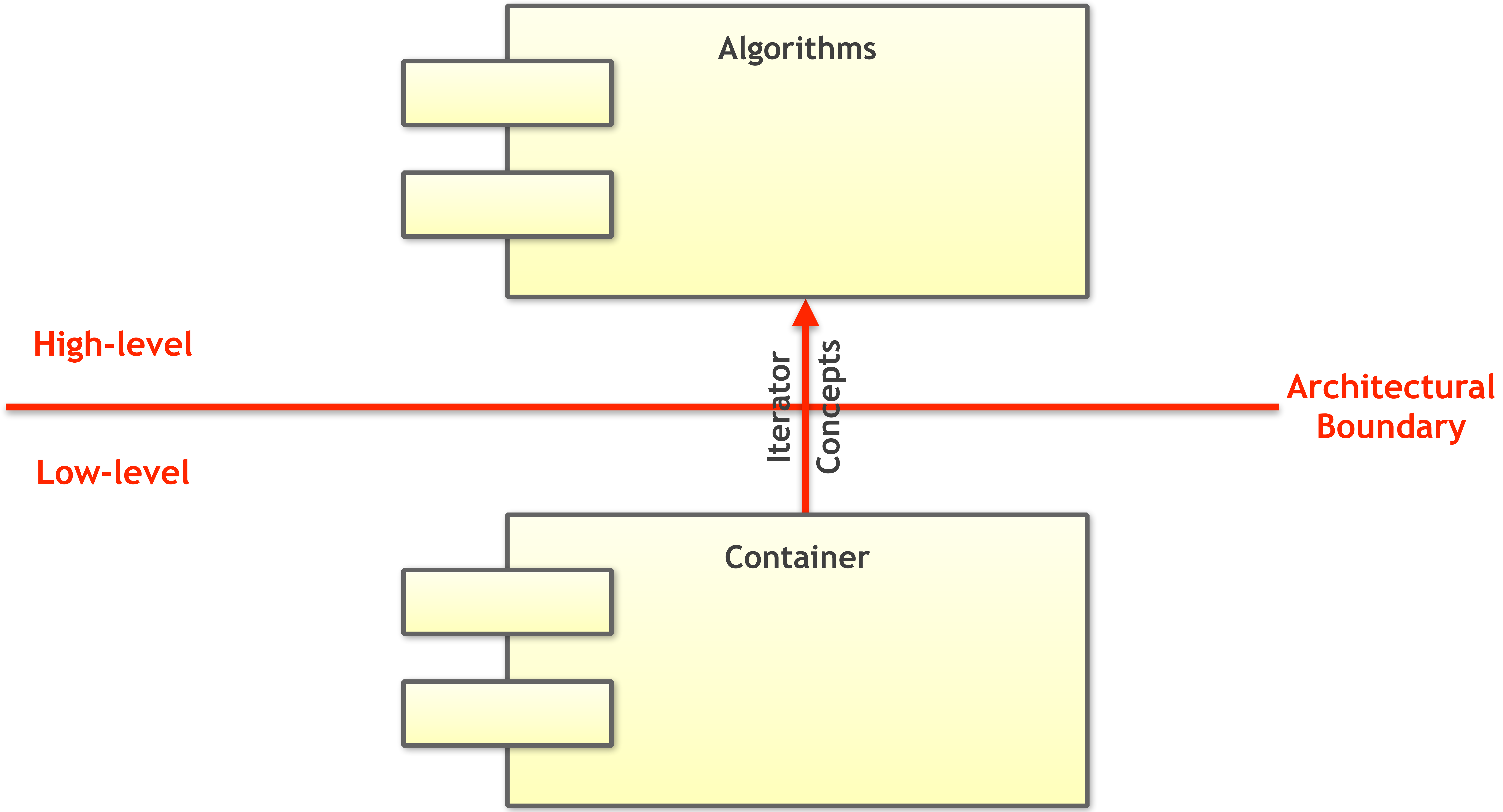
```
template< class InputIt, class OutputIt, class UnaryOperation >
OutputIt transform( InputIt first1, InputIt last1, OutputIt d_first,
                    UnaryOperation unary_op );
```

3. The Iterator Design Pattern



Iterator category					Defined operations
LegacyContiguousIterator	*LegacyRandomAccessIterator*	*LegacyBidirectionalIterator*	*LegacyForwardIterator*	*LegacyInputIterator*	• read • increment (without multiple passes)
					• increment (with multiple passes)
					• decrement
					• random access
					• contiguous storage
Iterators that fall into one of the above categories and also meet the requirements of *LegacyOutputIterator* are called mutable iterators.					
LegacyOutputIterator					• write • increment (without multiple passes)

3. The Iterator Design Pattern



4. The Adapter Design Pattern

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

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

template<
    class T,
    class Container = std::vector<T>,
    class Compare = std::less<typename Container::value_type>
> class priority_queue;

namespace pmr {
    template <class T>
    using vector = std::vector<T, std::pmr::polymorphic_allocator<T>>;
}
```

5. The Decorator Design Pattern

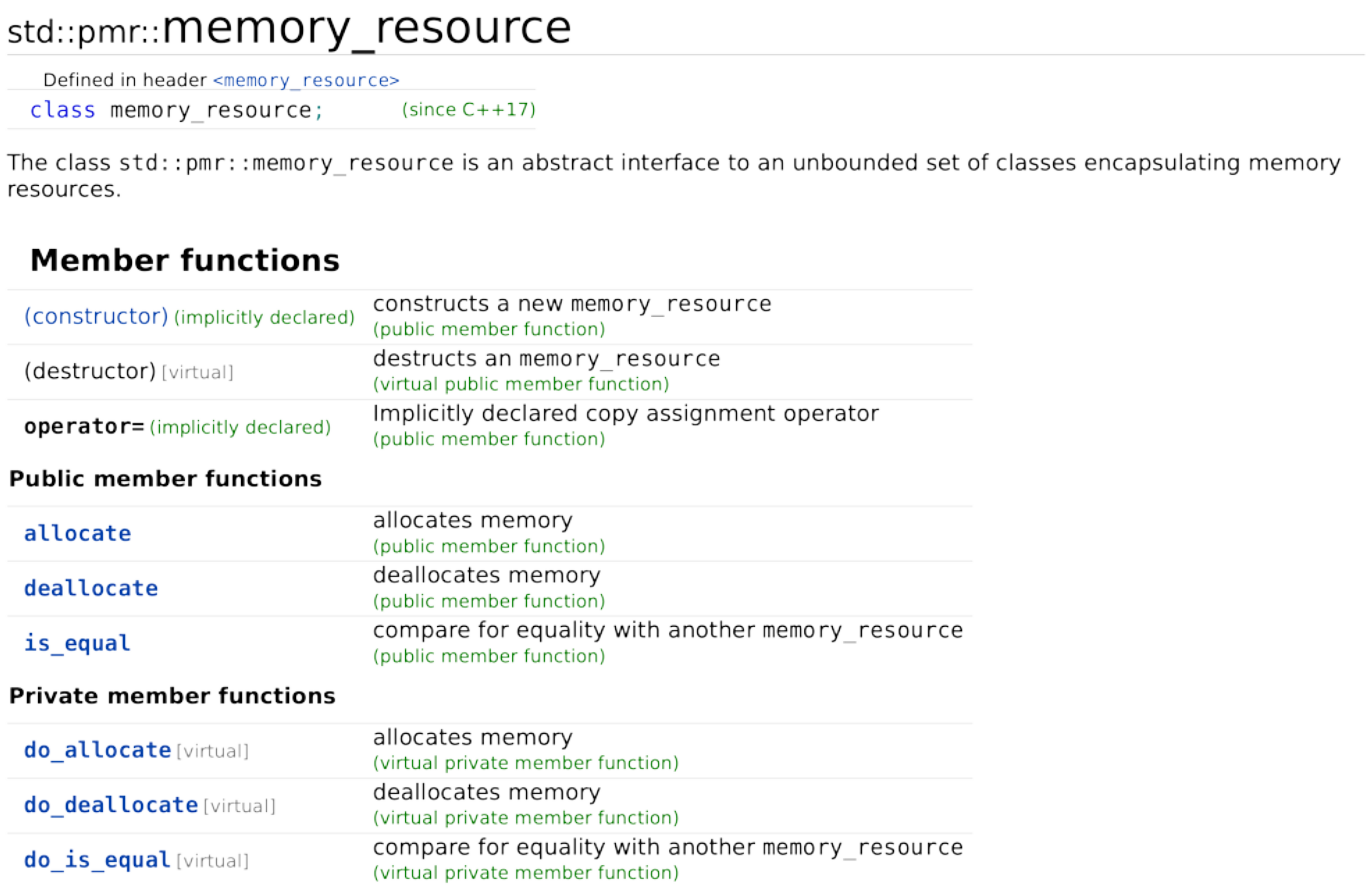
```
// ...
#include <memory_resource>


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;
}
```

5. The Decorator Design Pattern

Yes, there is a Singleton, but Singleton is not a design pattern!

```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;
}
```

6. The Template Method Design Pattern



std::pmr::memory_resource

Defined in header `<memory_resource>`

```
class memory_resource;
```
(since C++17)

The class `std::pmr::memory_resource` is an abstract interface to an unbounded set of classes encapsulating memory resources.

Member functions

(constructor) (implicitly declared)	constructs a new `memory_resource` (public member function)
(destructor) [virtual]	destructs an `memory_resource` (virtual public member function)
operator= (implicitly declared)	Implicitly declared copy assignment operator (public member function)

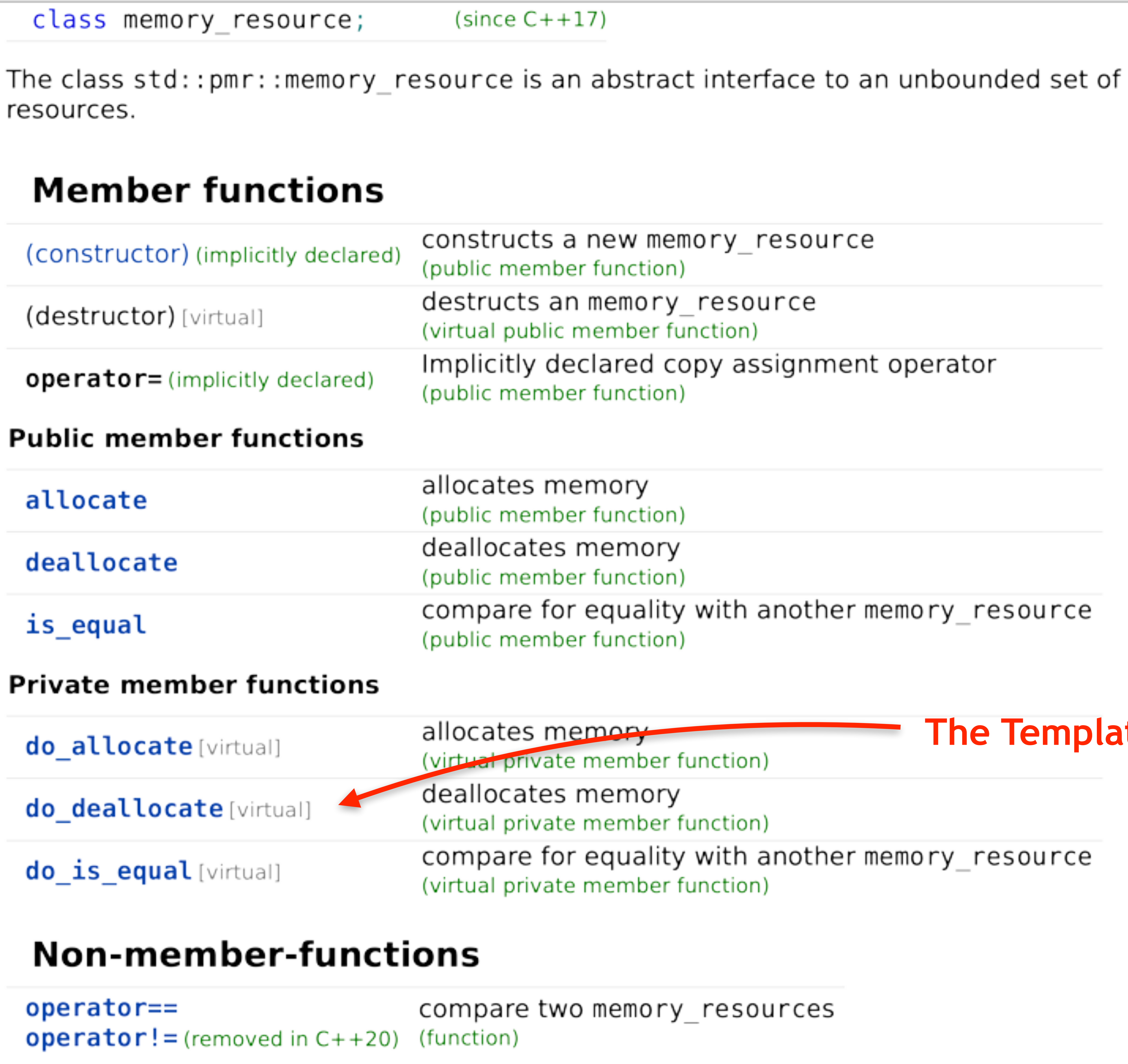
Public member functions

allocate	allocates memory (public member function)
deallocate	deallocates memory (public member function)
is_equal	compare for equality with another `memory_resource` (public member function)

Private member functions

do_allocate [virtual]	allocates memory (virtual private member function)
do_deallocate [virtual]	deallocates memory (virtual private member function)
do_is_equal [virtual]	compare for equality with another `memory_resource` (virtual private member function)

6. The Template Method Design Pattern



```
class memory_resource;
```
(since C++17)

The class `std::pmr::memory_resource` is an abstract interface to an unbounded set of classes encapsulating memory resources.

Member functions

(constructor) (implicitly declared)	constructs a new `memory_resource` (public member function)
(destructor) [virtual]	destructs an `memory_resource` (virtual public member function)
`operator=` (implicitly declared)	Implicitly declared copy assignment operator (public member function)
Public member functions	
`allocate`	allocates memory (public member function)
`deallocate`	deallocates memory (public member function)
`is_equal`	compare for equality with another `memory_resource` (public member function)
Private member functions	
`do_allocate` [virtual]	allocates memory (virtual private member function)
`do_deallocate` [virtual]	deallocates memory (virtual private member function)
`do_is_equal` [virtual]	compare for equality with another `memory_resource` (virtual private member function)

Non-member-functions

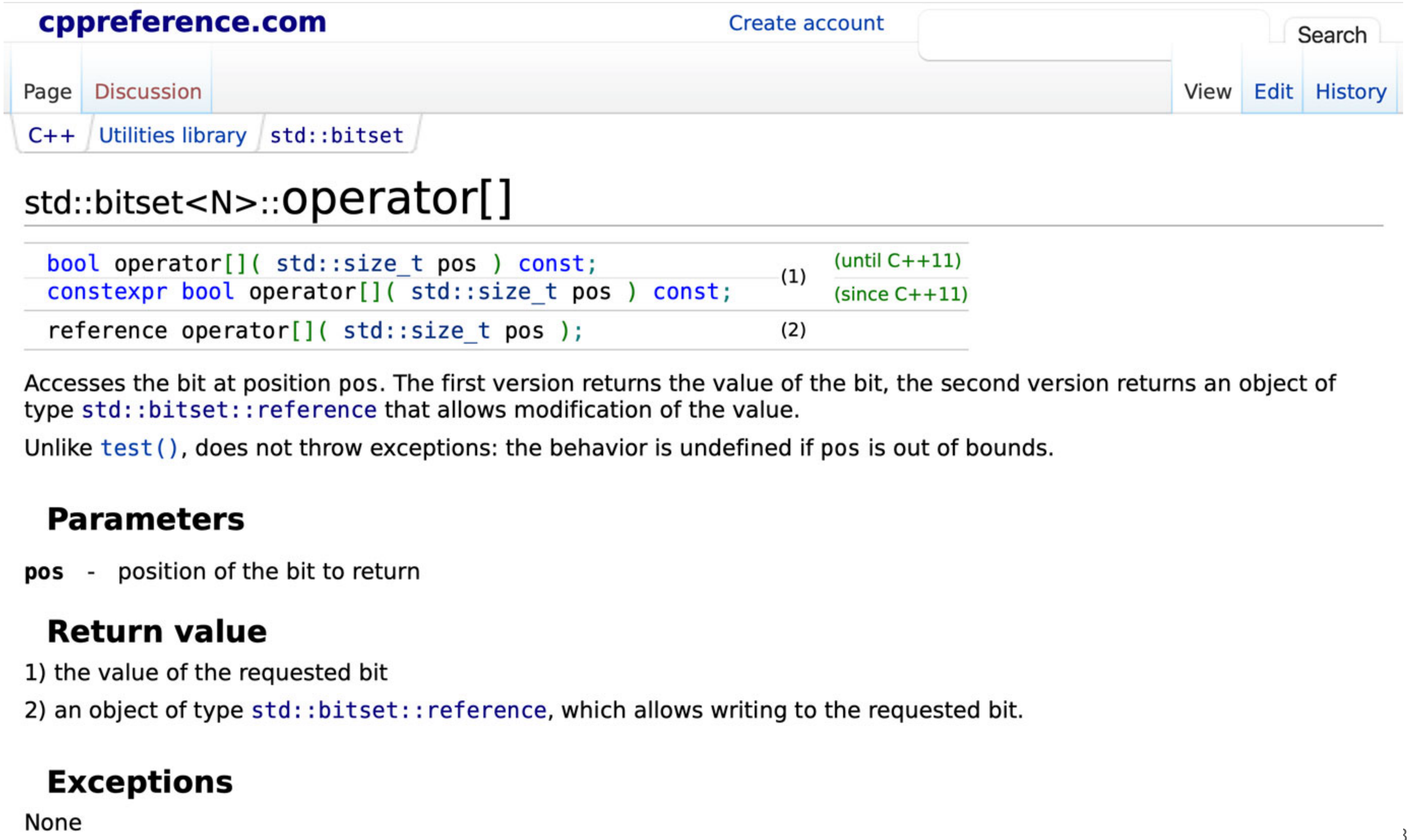
`operator==` **`operator!=`** (removed in C++20)	compare two `memory_resource`s (function)

The Template Method Design Pattern

7. The Proxy Design Pattern

```
std::vector<bool> vec{ false, true, false, true };

                              // Accessing an element returns
auto&& element = vec[2];      //  a proxy representing a ‘bool’

element = true;    // Sets the element (2) to ‘true’
```

7. The Proxy Design Pattern



std::bitset<N>::operator[]

```
bool operator[]( std::size_t pos ) const;                 (1)  (until C++11)
constexpr bool operator[]( std::size_t pos ) const;            (since C++11)
reference operator[]( std::size_t pos );                  (2)
```

Accesses the bit at position pos. The first version returns the value of the bit, the second version returns an object of type `std::bitset::reference` that allows modification of the value.

Unlike `test()`, does not throw exceptions: the behavior is undefined if pos is out of bounds.

Parameters

pos	-	position of the bit to return

Return value

1) the value of the requested bit

2) an object of type `std::bitset::reference`, which allows writing to the requested bit.

Exceptions

None

8. …

```
namespace std {

template< typename Fn >
class function;

template< typename R, typename... Args >
class function<R(Args...)>
{
 public:
   template< typename F >
   function( F&& f )
      : pimpl_( std::make_unique<Model<Fn>>(
                   std::forward<F>(f) ) )
   {}

   R operator()( Args... args ) const {
      return pimpl_->invoke( std::forward<Args>( args )... );
   }

   ~function() = default;
   function( function&& ) = default;
   function& operator=( function&& ) = default;

   function( function const& other )
      : pimpl_( other.pimpl_->clone() )
   {}

   function& operator=( const function& other )
   {
      function tmp( other );
      std::swap( pimpl_, tmp.pimpl_ );
      return *this;
```

8. …

```
namespace std {

template< typename Fn >
class function;

template< typename R, typename... Args >
class function<R(Args...)>
{
 public:
   template< typename F >
   function( F&& f )
      : pimpl_( std::make_unique<Model<Fn>>(
                   std::forward<F>(f) ) )
   {}

   R operator()( Args... args ) const {
      return pimpl_->invoke( std::forward<Args>( args )... );
   }

   ~function() = default;
   function( function&& ) = default;
   function& operator=( function&& ) = default;

   function( function const& other )
      : pimpl_( other.pimpl_->clone() )
   {}

   function& operator=( const function& other )
   {
      function tmp( other );
      std::swap( pimpl_, tmp.pimpl_ );
      return *this;
```

8. …

```
namespace std {

template< typename Fn >
class function;

template< typename R, typename... Args >
class function<R(Args...)>
{
 public:
   template< typename F >
   function( F&& f )
      : pimpl_( std::make_unique<Model<Fn>>(
                     std::forward<F>(f) ) )
   {}

   R operator()( Args... args ) const {
      return pimpl_->invoke( std::forward<Args>( args )... );
   }

   ~function() = default;
   function( function&& ) = default;
   function& operator=( function&& ) = default;

   function( function const& other )
      : pimpl_( other.pimpl_->clone() )
   {}

   function& operator=( const function& other )
   {
      function tmp( other );
      std::swap( pimpl_, tmp.pimpl_ );
      return *this;
```

8. ...

```
private:
  class Concept
  {
   public:
     virtual ~Concept() = default;
     virtual std::unique_ptr<Concept> clone() const = 0;
     virtual R invoke( Args... ) const = 0;
  };

  template< typename F >
  class Model final : public Concept
  {
   public:
     explicit Model( F f )
        : fn_( std::move(f) )
     {}

     std::unique_ptr<Concept> clone() const final {
        return std::make_unique<Model>( fn_ );
     }
     R invoke( Args... args ) const final {
        return fn_( std::forward<Args>( args )... );
     }

   private:
     Fn fn_;
  };

  std::unique_ptr<Concept> pimpl_;
};

} // namespace std
```

8. …

```
 private:
   class Concept
   {
    public:
      virtual ~Concept() = default;
      virtual std::unique_ptr<Concept> clone() const = 0;
      virtual R invoke( Args... ) const = 0;
   };

   template< typename F >
   class Model final : public Concept
   {
    public:
      explicit Model( F f )
         : fn_( std::move(f) )
      {}

      std::unique_ptr<Concept> clone() const final {
         return std::make_unique<Model>( fn_ );
      }
      R invoke( Args... args ) const final {
         return fn_( std::forward<Args>( args )... );
      }

    private:
      Fn fn_;
   };

   std::unique_ptr<Concept> pimpl_;
};

} // namespace std
```

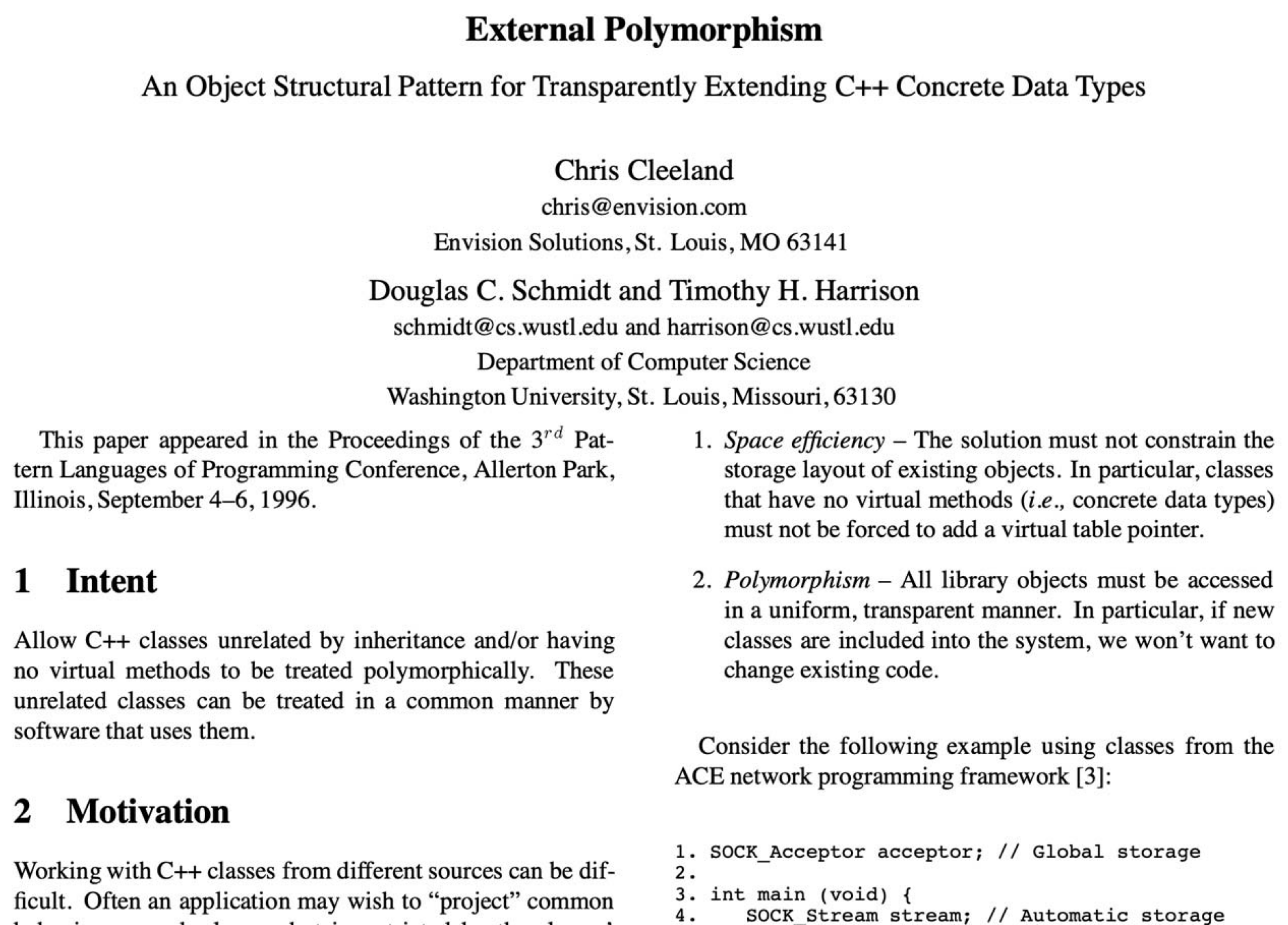
8. The External Polymorphism Design Pattern

```
 private:
   class Concept
   {
    public:
      virtual ~Concept() = default;
      virtual std::unique_ptr<Concept> clone() const = 0;
      virtual R invoke( Args... ) const = 0;
   };

   template< typename F >
   class Model final : public Concept
   {
    public:
      explicit Model( F f )
         : fn_( std::move(f) )
      {}

      std::unique_ptr<Concept> clone() const final {
         return std::make_unique<Model>( fn_ );
      }
      R invoke( Args... args ) const final {
         return fn_( std::forward<Args>( args )... );
      }

    private:
      Fn fn_;
   };

   std::unique_ptr<Concept> pimpl_;
};

} // namespace std
```

The External Polymorphism Design Pattern

8. The External Polymorphism Design Pattern

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



External Polymorphism

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

Chris Cleeland
chris@envision.com
Envision Solutions, St. Louis, MO 63141

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

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

1 Intent

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

2 Motivation

Working with C++ classes from different sources can be difficult. Often an application may wish to "project" common

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

2. *Polymorphism* – All library objects must be accessed in a uniform, transparent manner. In particular, if new classes are included into the system, we won't want to change existing code.

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

```
SOCK_Acceptor acceptor; // Global storage

int main (void) {
   SOCK_Stream stream; // Automatic storage
```

9. The Bridge Design Pattern

```
   class Concept
   {
    public:
      virtual ~Concept() = default;
      virtual std::unique_ptr<Concept> clone() const = 0;
      virtual R invoke( Args... ) const = 0;
   };

   template< typename F >
   class Model final : public Concept
   {
    public:
      explicit Model( F f )
         : fn_( std::move(f) )
      {}

      std::unique_ptr<Concept> clone() const final {
         return std::make_unique<Model>( fn_ );
      }
      R invoke( Args... args ) const final {
         return fn_( std::forward<Args>( args )... );
      }

    private:
      Fn fn_;
   };

   std::unique_ptr<Concept> pimpl_;
};

} // namespace std
```

The Bridge Design Pattern

10. The Prototype Design Pattern

```
  function( function const& other )
     : pimpl_( other.pimpl_->clone() )
  {}

  function& operator=( const function& other )
  {
     function tmp( other );
     std::swap( pimpl_, tmp.pimpl_ );
     return *this;
  }

private:
  class Concept
  {
   public:
     virtual ~Concept() = default;
     virtual std::unique_ptr<Concept> clone() const = 0;
     virtual R invoke( Args... ) const = 0;
  };

  template< typename F >
  class Model final : public Concept
  {
   public:
     explicit Model( F f )
        : fn_( std::move(f) )
     {}

     std::unique_ptr<Concept> clone() const final {
        return std::make_unique<Model>( fn_ );
     }
     R invoke( Args... args ) const final {
        return fn_( std::forward<Args>( args )... );
```

The Prototype Design Pattern

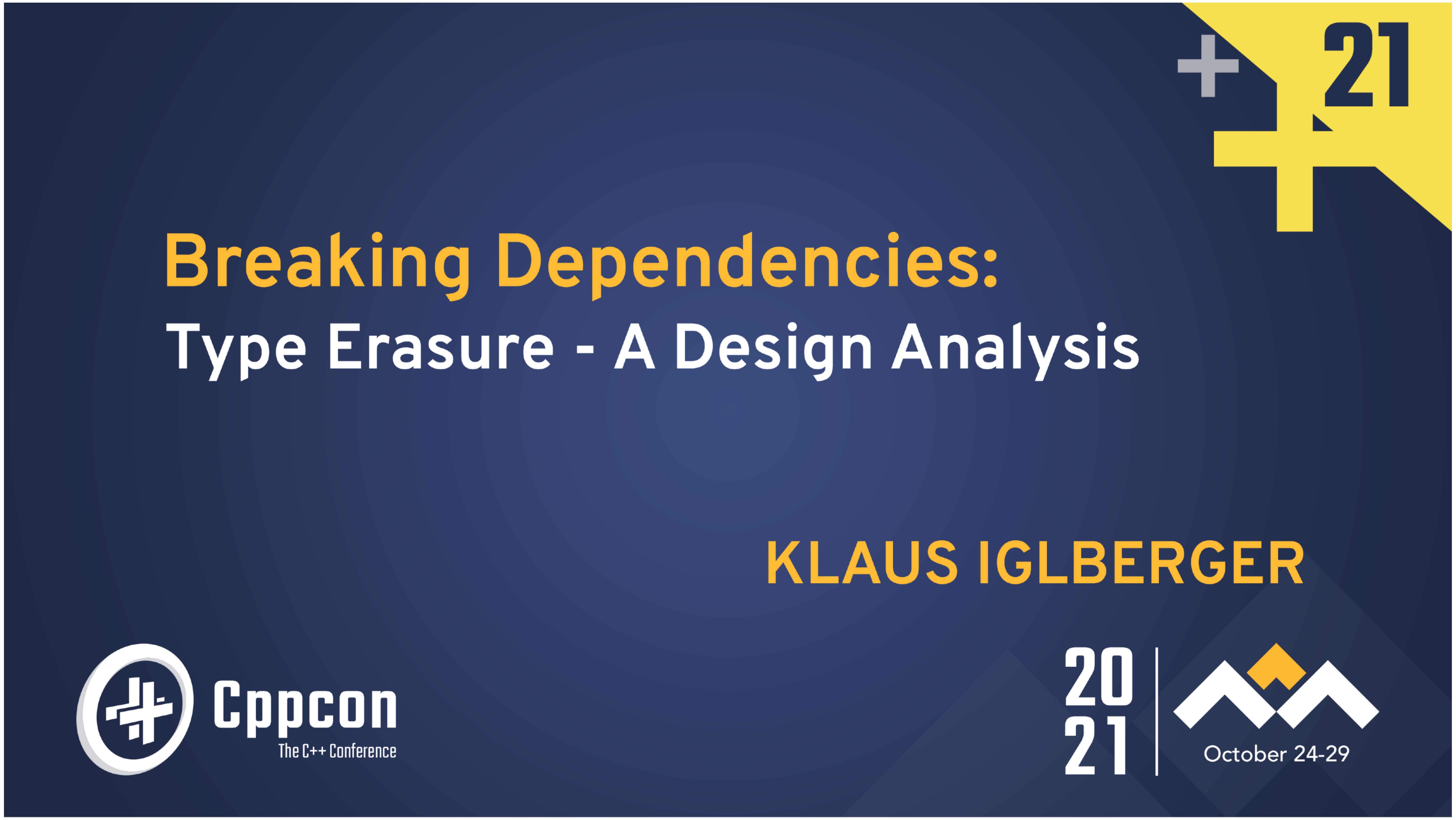
11. The Type Erasure Design Pattern

```
namespace std {

template< typename Fn >
class function;

template< typename R, typename... Args >
class function<R(Args...)>
{
 public:
   template< typename F >
   function( F&& f )
      : pimpl_( std::make_unique<Model<Fn>>(
                   std::forward<F>(f) ) )
   {}

   R operator()( Args... args ) const {
      return pimpl_->invoke( std::forward<Args>( args )... );
   }

   ~function() = default;
   function( function&& ) = default;
   function& operator=( function&& ) = default;

   function( function const& other )
      : pimpl_( other.pimpl_->clone() )
   {}

   function& operator=( const function& other )
   {
      function tmp( other );
      std::swap( pimpl_, tmp.pimpl_ );
      return *this;
```

11. The Type Erasure Design Pattern

Type Erasure is …

- … a **templated constructor;**
- … a completely **non-virtual interface;**
- **… External Polymorphism + Bridge + Prototype.**



Breaking Dependencies:

Type Erasure - A Design Analysis

KLAUS IGLBERGER

Cppcon
The C++ Conference

2021 | October 24-29

Thursday, October 28th, 7:45am MDT

Guidelines

Guideline: Design patterns are not outdated, nor obsolete. The C++ standard library is full of them.

Guideline: Design patterns are everywhere. Learn to recognize them and use according names to communicate intent.

Summary

Design patterns …

- … are about dependencies and abstractions;
- … are about intent;
- … are not limited to OO programming;
- … are not limited to dynamic polymorphism;
- … are not outdated nor obsolete;
- … are everywhere!

Design Patterns: Facts and Misconceptions

KLAUS IGLBERGER

Cppcon
The C++ Conference

2021
October 24-29