Basic C++

Design Patterns

Dr. Porkoláb Zoltán Károly

gsd@inf.elte.hu

http://gsd.web.elte.hu

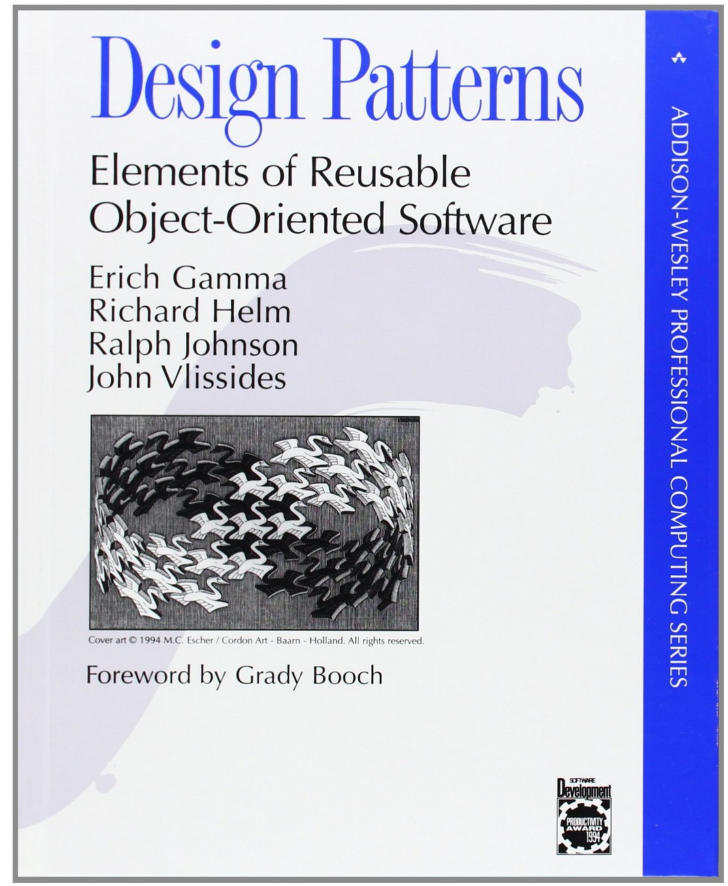
Outline

- On design patterns in general
- Creational patterns
  - Factory method
  - Singleton
- Structural patterns
  - Adaptor
  - Bridge
- Behavioral patterns
  - Visitor
  - Iterator
  - Strategy/Policy
- C++ specific patterns
  - Attorney-client pattern
  - Mixin
  - CRTP

History of design patterns

- 1994: “Gang of four”: Design Patterns, Elements of reusable Object-Oriented Software

  (Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, foreword: Grady Booch)
- Christopher Alexander used the term for Architecture
- Late 90s: design pattern boom
- 2010s: bit of disappointment



What is a design pattern?

- Each pattern describes a problem which occurs over and over again in our environment (Alexander)
- Description of communicating objects objects and classes that are customized to solve a general design problem in a particular context (GoF)
- Originally used examples in C++
- But they mention MVC in Smalltalk

What is a design pattern 1.

- Pattern name and classification
- Intent
- Alternative names (“also knows as”)
- Motivation
- Applicability

What is a design pattern 2.

- Structure
- Participants
- Collaborations
- Consequences
- Implementation
- Sample code
- Known uses
- Related patterns

Original GoF patterns

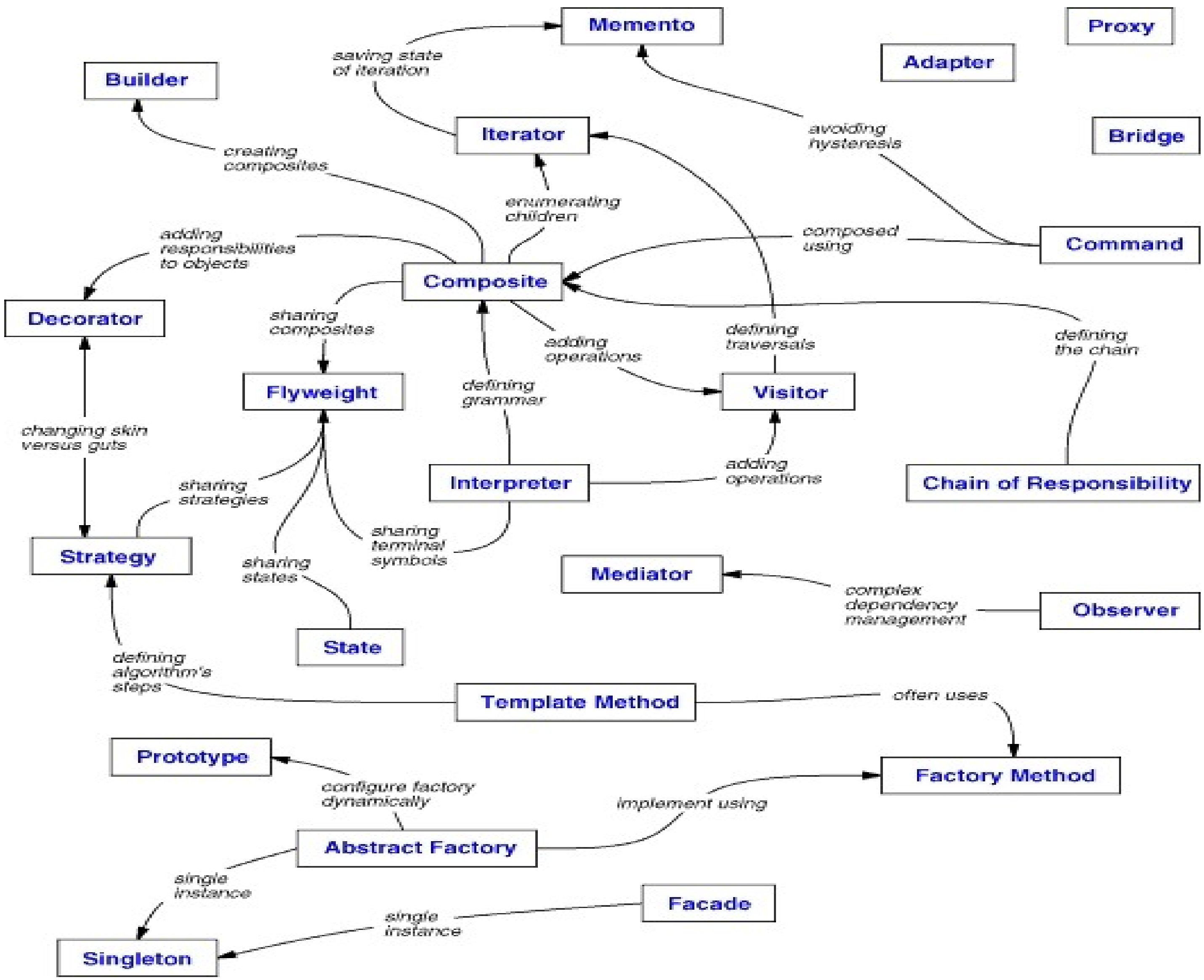
- Creational
  - Abstract factory
  - Builder
  - Factory method
  - Prototype
  - Singleton
- Structural
  - Adapter
  - Bridge
  - Composite
  - Decorator
  - Facade
  - Flyweight
  - Proxy
- Behavioral
  - Chain of Responsibility
  - Command
  - Interpreter
  - Iterator
  - Mediator
  - Memento
  - Observer
  - State
  - Strategy
  - Template Method
  - Visitor

Pattern categorization

Design Pattern Space

		Purpose		
		Creational	Structural	Behavioural
Scope	Class	Factory method	Adapter (class)	Interpreter Template method
	Object	Abstract factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Façade Flyweight Proxy	Chain of responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Pattern relationships



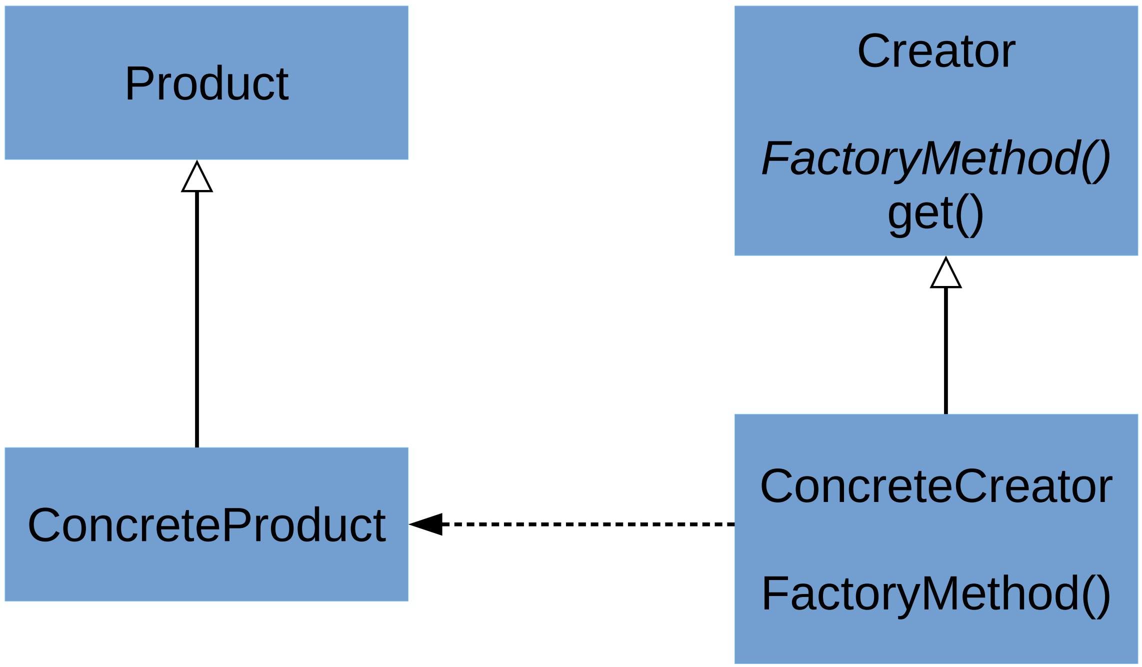
Creational patterns

- Abstract instantiation process
- Make system independent how objects created, implemented
- Encapsulates which concrete classes the system use (instead uses the interface)
- Alternatives
  - Prototype vs abstract factory
- Complementary
  - Builder, prototype, singleton

Factory method

- Intent
  - Define an interface for creating an object, but let subclasses decide which class to instantiate.
  - Factory method lets a class defer instantiation to subclasses
- Participants
  - Product, ConcreteProduct, Creator, ConcreteCreator

Factory method



Original implementation

```
class Creator
{
public:
    virtual Product *Create(ProductId);  // Optionally pure virtual
};

// Optional default implementation
Product *Creator::Create(ProductId id)
{
    if ( ONE_ID == id )   return new ProductOne;
    if ( TWO_ID == id )   return new ProductTwo;

    // others …

    return 0;
}
```

Original implementation

```
class ConcreteCreator : public Creator
{
public:
   virtual Product *Create(ProductId);
};

Product *ConcreteCreator::Create(ProductId id)
{
   // Optionally changes default implementation
   if ( ONE_ID == id )   return new ProductOnePrime;

   // Optionally changes default implementation
   if ( TWO_ID == id )   return new ProductTwoPrime;

   // Optionally extends default implementation
   if ( ONE_AND_HALF_ID == id )   return new ProductOneAndHalf;

   // if none of them …
   return Cretator::Create(id);
}
```

Templated implementation

```
class Creator
{
public:
   virtual Product *Create(ProductId) = 0;
};

template <class TheProduct>
class StandardCreator : public Creator
{
public:
   virtual Product *Create(ProductId) = 0;
}

template <class TheProduct>
Product  *StandardCreator::CreateProduct( )
{
   return new TheProduct;
}

// user defines his own creator
StandardCreator<MyProduct>  myCreator;
```

Problems

- Calling the factory method during construction
  - The virtual function may call improper implementation
  - Possible solution: use an additional get() method to access (and lazy create) the object.
- Constructors require parameters from outer word
- Constructors require different parameters (Constructor problem)
- We want to write template Factory methods
- Perfect forwarding

Constructor problem

```
template <typename T1, typename T2>
Product *factory(T1 e1, T2 e2)
{
    return new Product(e1, e2);
}
```

We need a number of overloaded versions, however

```
template <typename T1, typename T2>
Product *factory(T1& e1, T2& e2)
{
    return new Product(e1, e2);
}
template <typename T1, typename T2>
Product *factory(const T1& e1, const T2& e2)
{
    return new Product(e1, e2);
}
template <typename T1, typename T2>
Product *factory(T1&& e1, T2&& e2)
{
    return new Product(e1, e2);
}
```

Forwarding reference

Forwarding reference is used in case of type deduction

```
class X;
Void f(X&& param)        // rvalue reference
X&& var1 = X();          // rvalue reference
auto&& var2 = var1;      // NOT rvalue reference: forwarding reference
template <typename T>
void f(std::vector<T>&& param); // rvalue reference (1)
template <typename T>
void f(T&& param);        // NOT rvalue reference: forwarding reference (2)
template <typename T>
void f(const T&& param); // rvalue reference

X var;
f(var);                  // lvalue passed: param type is: X& (2)
f(std::move(var));       // rvalue passed: param type is: X&& (2)
std::vector<int> v;
f(v);                    // syntax error: can't bind lvalue to rvalue  (1)
```

Forwarding(Universal) reference

Forwarding reference is used in case of type deduction

```
template <class T, class Allocator = allocator<T>>
class vector
{
public:
  void push_back(T&& x);  // rvalue reference, no type deduction here
  // ...
};

template <class T, class Allocator = allocator<T>>
class vector
{
public:
  template <class... Args>
  void emplace_back(Args&&... args);    // forwarding reference, type deduction
  // ...
};
```

Constructor problem

```
template<typename T, typename Arg>
shared_ptr<T> factory(Arg arg)
{
  return shared_ptr<T>(new T(arg));    // call T(arg) by value. Bad!
}

// A half-good solution is passing arg by reference:
template<typename T, typename Arg>
shared_ptr<T> factory(Arg& arg)
{
  return shared_ptr<T>(new T(arg));
}

But this does not work for rvalue parameters:

factory<X>(f());        // error if f() returns by value
factory<X>(42);         // error
```

Perfect forwarding

```
// If f() called on "lvalue of A"  T --> A&      argument type --> A&
// If f() called on "rvalue of A"  T --> A       argument type --> A

template<typename T, typename Arg>
shared_ptr<T> factory(Arg&& arg)
{
  return shared_ptr<T>(new T(std::forward<Arg>(arg)));
}
template<class S>
S&& forward(typename remove_reference<S>::type& a) noexcept
{
  return static_cast<S&&>(a);
}

// Reference collapsing:

A& &   --> A&
A& &&  --> A&
A&& &  --> A&
A&& && --> A&&
```

Using variadic templates 1.

```
#include <memory>

class Base { ... };
class Derived1 : public Base { ... };
class Derived2 : public Base { ... };


template <typename... Ts>
std::unique_ptr<Base>  makeBase( Ts&&... params)  { ... }

void f() // client code: we want to use this way
{
  auto pBase = makeBase( /* arguments */ );
}
// destroy object
```

Using variadic templates 2.

```
auto delBase = [](Base *pBase)   // optional deleter
               {
                   makeLogEntry(pBase);
                   delete pBase; // delete object
               };

template <typename... Ts>
std::unique_ptr<Base, decltype(delBase)>  makeBase( Ts&&... params)
{
  std::unique_ptr<Base, decltype(delBase)> pBase(nullptr, delBase);

  if ( /* Derived1 */ )
  {
    pBase.reset(new Derived1( std::forward<Ts>(params)... ));
  }
  else if ( /* Derived2 */ )
  {
    pBase.reset(new Derived2( std::forward<Ts>(params)... ));
  }
  return pBase;
}
```

Related

- Abstract factory
- Prototype (Cloning)

Cloning – “Virtual” constructors

- Constructors are not virtual
- But sometimes we need similar behavior

```
std::vector<Base*> source;
std::vector<Base*> target;

source.push_back(new Derived1());
source.push_back(new Derived2());
source.push_back(new Derived3());

// should create new instances of the
// corresponding Derived classes and
// place them to target
deep_copy( target, source);
```

Wrong approach

```
deep_copy( std::vector<Base*> &target,
           const std::vector<Base*> &source)
{
  for(auto i = source.begin(); i!=source.end(); ++i)
  {
    target.push_back(new Base(**i)); // creates Base()
  }
}
```

Wrong approach 2

```
deep_copy( std::vector<Base*> &target,
           const std::vector<Base*> &source)
{
  for(auto i = source.begin(); i!=source.end(); ++i)
  {
    if ( Derived1 *dp = dynamic_cast<Derived1*>(*i) )
      target.push_back(new Derived1(*dp));
    else if ( Derived2 *dp = dynamic_cast<Derived2*>(*i) )
      target.push_back(new Derived2(*dp));
    else if ( Derived3 *dp = dynamic_cast<Derived3*>(*i) )
      target.push_back(new Derived3(*dp));
  }
}
```

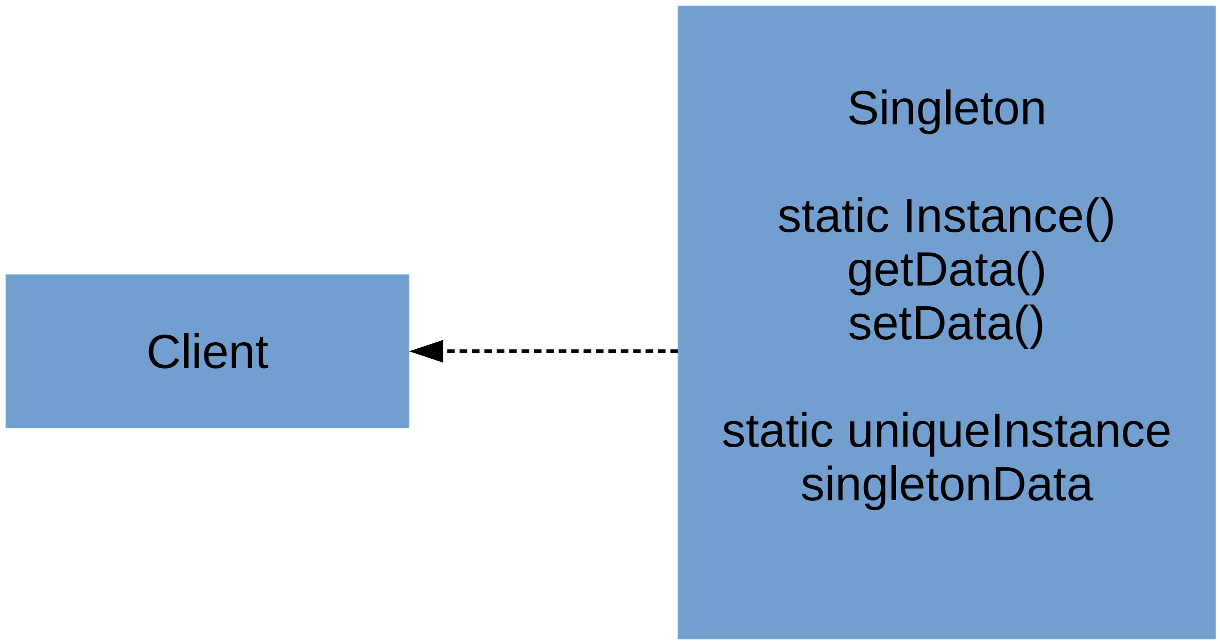
Cloning

```
class Base
{
public:
  virtual Base* clone() const = 0;
};

class Derived : public Base
{
public:
  virtual Derived* clone() const { return new Derived(*this); }
};

deep_copy( std::vector<Base*> &target, const std::vector<Base*>
&source)
{
  for(auto i = source.begin(); i!=source.end(); ++i)
  {
    target.push_back((*i)->clone()); // inserts Derived()
  }
}
```

Singleton

- Intent
  - Ensure the class has only one instance
  - Provide a global point of access
- Participants
  - Singleton

Singleton


Client
Singleton
static Instance()
getData()
setData()
static uniqueInstance
singletonData

Discussion

- Why we cannot use globals?
  - We might not have information to construct the Singleton (e.g. constructor parameters come from outside)
  - Global declaration order is not defined across translation units

Original implementation

```
// in singleton.h:
class Singleton
{
public:
    static Singleton *instance();
    void other_method();
    // other methods ...
private:
    static Singleton *pinstance;
};

// in singleton.cpp:
Singleton *Singleton::pinstance = 0;

Singleton *Singleton::instance()
{
    if ( 0 == pinstance )
    {
        pinstance = new Singleton;  // lazy initialization
    }
    return pinstance;
}

// Usage:

   Singleton::istance()-> other_method();
```

Thread safe singleton construction

```
// in singleton.h:
class Singleton
{
public:
    static Singleton *instance();
    void other_method();
    // other methods ...
private:
    static Singleton *pinstance;
    static Mutex      lock_;
};

// in singleton.cpp:
Singleton *Singleton::pinstance = 0;

Singleton *Singleton::instance()
{
    Guard<Mutex> guard(lock_);  // constructor acquires lock_
    // this is now the critical section
    if ( 0 == pinstance )
    {
        pinstance = new Singleton;  // lazy initialization
    }
    return pinstance;
} // destructor releases lock_
```

Double checked locking pattern

```
Singleton *Singleton::instance()   // Do not use it, it is wrong!
{
    if ( 0 == pinstance )
    {
        Guard<Mutex> guard(lock_);  // constructor acquires lock_
        // this is now the critical section

        if ( 0 == pinstance )   // re-check pinstance
        {
            pinstance = new Singleton;  // lazy initialization
        }
    }                               // destructor releases lock_
    return pinstance;
}
```

```
Singleton::istance()-> other_method();  // does not lock usually
```

Problems with DCLP

```
  if ( 0 == pinstance )
  {
        // ...
        pinstance = new Singleton;  // atomic?
        // ...
  }
  return pinstance;

// might use half-initialized pointer value
Singleton::istance()-> other_method();
```

- Pointer assignment may not be atomic
  - If can check an invalid, but not null pointer value

New expression

```
pinstance = new Singleton;  // how this is compiled?
```

- New expression include many steps
  - (1) Allocation space with ::operator new()
  - (2) Run of constructor
  - (3) Returning the pointer

- If the compiler does (1) + (3) and leaves (2) as the last step the pointer points to uninitialized memory area

Observable behavior in C++98

```
void foo()
{
    int x = 0, y = 0;         // (1)
    x = 5;                    // (2)
    y = 10;                   // (3)
    printf( "%d,%d", x, y); // (4)
}
```

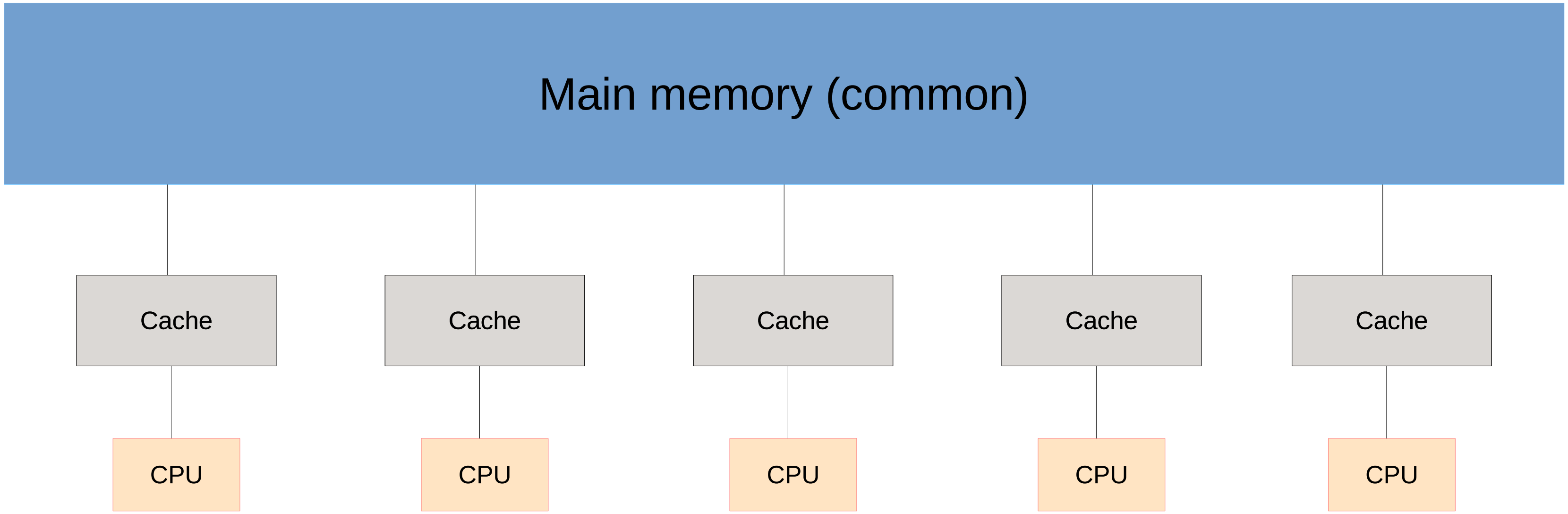
- What is visible for the outer word
  - I/O operations
  - Read/write volatile objects
- Defined by a singled-threaded mind

Sequence point

```
if ( 0 == pinstance )   // re-check pinstance
{
    // pinstance = new Singleton;
    Singleton *temp = operator new( sizeof(Singleton) );
    new (temp) Singleton;  // run the constructor
    pinstance = temp;
}
```

- The compiler can completely optimize out temp
- Even if we are using volatile temp we have issues

Modern hardware architecture



Singleton pattern

```
Singleton *Singleton::instance()
{
    Singleton *temp = pInstance;    // read pInstance

    Acquire();      // prevent visibility of later memory operations
                    // from moving up from this point

    if ( 0 == temp )
    {
        Guard<Mutex> guard(lock_);
        // this is now the critical section

        if ( 0 == pinstance )   // re-check pinstance
        {
            temp = new Singleton;

            Release();  // prevent visibility of earlier memory operations
                        // from moving down from this point

            pinstance = temp;   // write pInstance
        }
    }
    return pinstance;
}
```

Concurrent singleton in C++11

```
template <typename T>
class MySingleton
{
public:
    std::shared_ptr<T> instance()
    {
        std::call_once( resource_init_flag, init_resource);
        return resource_ptr;
    }
private:
    void init_resource()
    {
        resource_ptr.reset( new T(...) );
    }
    std::shared_ptr<T> resource_ptr;
    std::once_flag     resource_init_flag; // can't be moved or copied
};
```

Meyers singleton

```
// Meyers singleton:
// C++11 guaranties: local static is initialized in a thread safe way
//
class MySingleton;
MySingleton& MySingletonInstance()
{
    static MySingleton _instance;
    return _instance;
}
```

Static initialization/destruction

- Static objects inside translation unit constructed in a well-defined order
- No ordering between translation units
- Issues:
  - (1) Constructor of static refers other source's static
  - (2) Destruction order
- Lazy singleton solves (1)
- Phoenix pattern solves (2)
- std::quick_exit() solves (3)

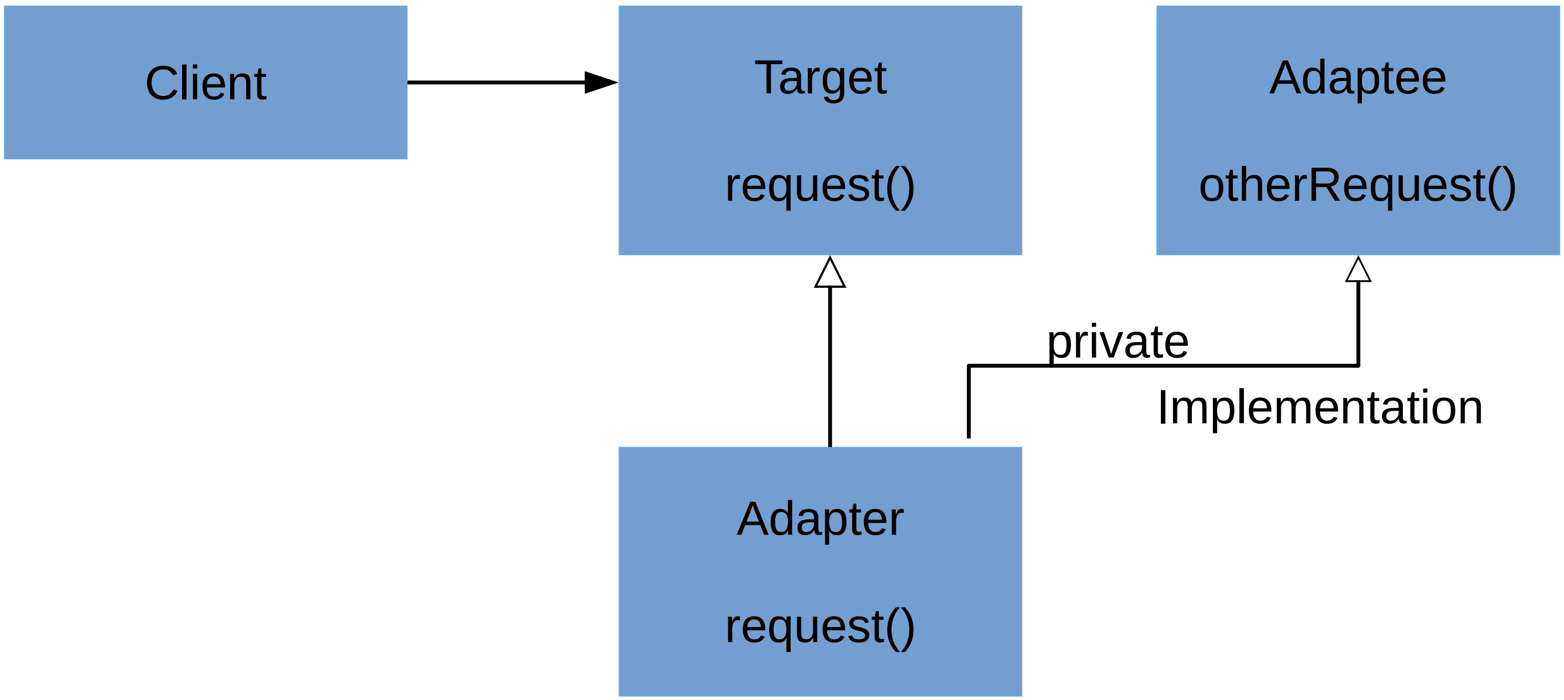
Schwartz counter

```
// init.h
class InitMngr
{
public:
  InitMngr()  { if ( !count_++ ) init();    }
  ~InitMngr() { if ( !--count_ ) cleanup(); }
  void init();
  void cleanup();
private:
  static long count_;  // one per process
};
namespace { InitMngr initMngr; }   // one per file inclusion


// init.cpp
long InitMngr::count_ = 0;
void InitMngr::init()    {  /* initialization */ }
void InitMngr::cleanup() {  /* cleanup */        }
```

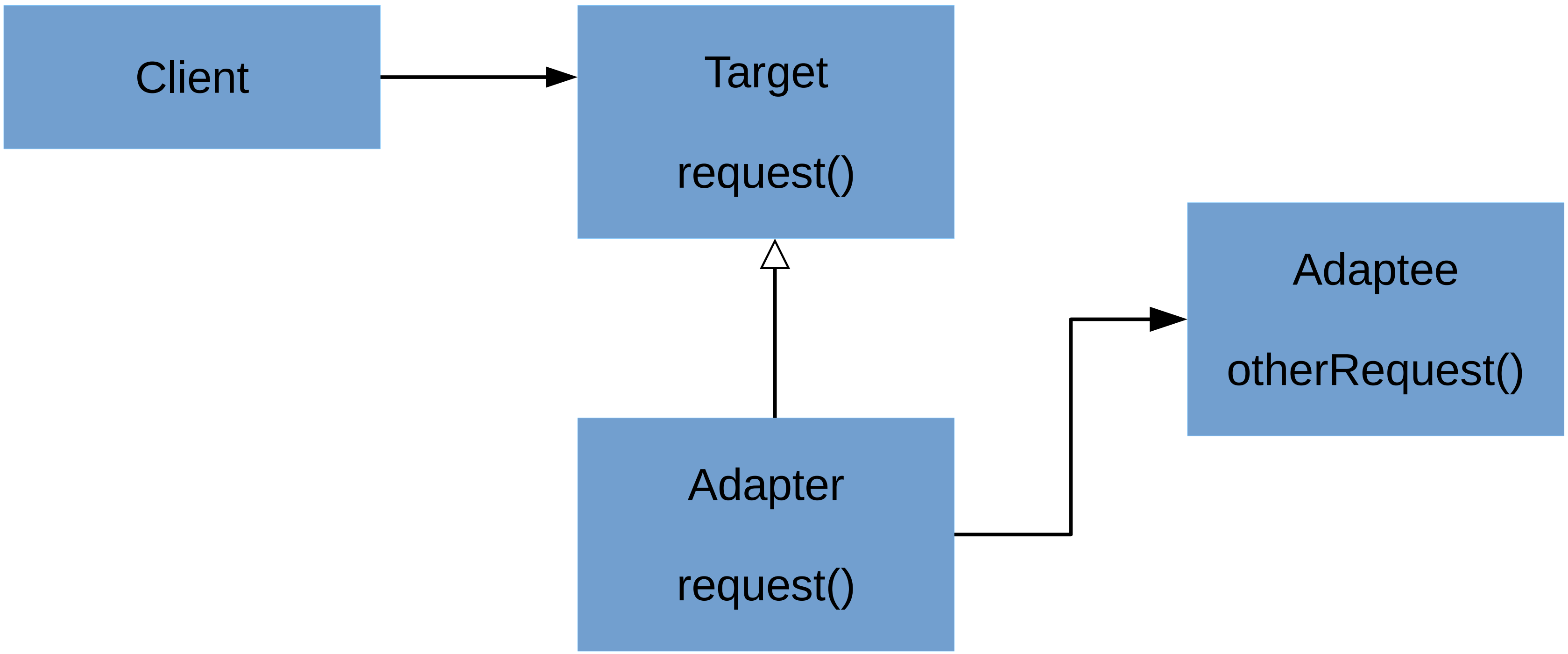
Adapter

- Intent
  - Convert the interface of a class into another interface the clients expect
  - Make possible to work otherwise incompatible interfaces
- Also known: wrapper
- Participants
  - Target, Client, Adapter, Adaptee
- Two variants:
  - Class adapter
  - Object adapter

Class Adapter



Object Adapter



Discussion

- Class and Object adapters have different trade-offs
- Class adapter
  - Adapts Adaptee to Target by committing to a concrete Adapter class.
  - Won't work with all the subclasses of Adaptee class
  - Introduces only one object and no additional pointer indirection is needed to get to the adaptee
- Object adapter
  - A single Adapter works with subclasses of Adaptee (if any)
  - Adaptee behavior is harder to override (but possible when subclassing Adaptee and Adapter refers to subclasses)
- Pluggable adapters

C++ Container Adaptors: stack

```
template <typename _Tp, typename _Sequence = deque<_Tp> >
class stack
{
// public type declarations, etc...
protected:
  _Sequence c;
public:
  explicit  stack(const _Sequence& _c )  :  c(_c)  { }
  explicit  stack(_Sequence&& _c )  :  c(std::move(_c))  { }

  bool empty() const { return c.empty();
  size_type size() const { return c.size(); }

  reference top() { return c.back(); }
  const_reference top() const { return c.back(); }

  void push( const value_type& x) { c.push_back(x); }
  void push( value_type&& x) { c.push_back(std::move(x)); }

  template<typename... Args>
  void emplace(Args&&... args) {c.emplace_back(std::forward<Args>(args)...);}

  void pop() { c.pop_back(); }
};
// global operators, etc...
```

Priority queue

```
template <typename _Tp, typename _Sequence = vector<_Tp>,
          typename _Compare  = less<typename _Sequence::value_type> >
class priority_queue
{
// public type declarations, etc...
protected:
  _Sequence c;
  _Compare  comp;
public:
  explicit  stack(const _Compare& x, const _Sequence& s ) : c(s), comp(x)
  { std::make_heap(c.begin(), c.end(), comp); }
  // ...
  bool empty() const { return c.empty();
  size_type size() const { return c.size(); }
  const_reference top() const { return c.front(); }   // only top() const
  // ...
  void push( const value_type& x) {
    c.push_back(x);
    std::push_heap(c.begin(), c.end(), comp);
  }
  void pop() {
    std::pop_heap(c.begin(), c.end(), comp);
    c.push_back(x);
  }
};
// global operators, etc...
```

Member function adapter

```
// shape hierarchy
class shape {
   virtual void draw() const;
};
class circle : public shape {
   virtual void draw() const;
};
class square : public shape {
   virtual void draw() const;
};
void f() {
  std::vector<shape*> v;
  v.push_back(&s); v.push_back(&c); // ...

  // Call draw on each one
  std::for_each(v.begin(), v.end(), std::mem_fun(&shape::draw));
}

std::set< std::list<int>* > s;
// put elements to list
...
std::for_each(s.begin(), s.end(), std::mem_fun(&std::list<int>::sort));
```

Member function adapter

```
template<typename _Ret, typename _Tp>
class mem_fun_t : public unary_function<_Tp*, _Ret>
{
public:
  explicit mem_fun_t(_Ret (_Tp::*pf)()) : _M_f(pf) { }
  _Ret operator()(_Tp* p) const { return (p->*_M_f)(); }
private:
  _Ret (_Tp::*_M_f)();
};

template<typename _Ret, typename _Tp>
class const_mem_fun_t : public unary_function<const _Tp*, _Ret>
{
public:
  explicit const_mem_fun_t(_Ret (_Tp::*pf)() const) : _M_f(pf) { }
  _Ret operator()(const _Tp* p) const { return (p->*_M_f)(); }
private:
  _Ret (_Tp::*_M_f)() const;
};
```

Example: merge two files

```cpp
#include <iostream>
#include <fstream>
#include <string>

using namespace std;

// simple merge
int main()
{
    string s1, s2;
    ifstream f1("file1.txt");
    ifstream f2("file2.txt");

    f1 >> s1; f2 >> s2;
    while (f1 || f2)
    {
        if (f1 && ((s1 <= s2) || !f2))
        {
            cout << s1 << endl;
            f1 >> s1;
        }
        if (f2 && ((s1 >= s2) || !f1))
        {
            cout << s2 << endl;
            f2 >> s2;
        }
    }
    return 0;
}
```

Example: naïve STL

```
#include <iostream>
#include <fstream>
#include <string>
#include <algorithm>    // merge( b1, e1, b2, e2, b3 [,opc_rend])
#include <vector>

using namespace std;
int main()
{
    ifstream if1("file1.txt");
    ifstream if2("file2.txt");

    string s;
    vector<string> v1;
    while ( if1 >> s ) v1.push_back(s);
    vector<string> v2;
    while ( if2 >> s ) v2.push_back(s);

    // allocate the space for the result
    vector<string> v3(v1.size() + v2.size());   // very expensive...

    merge( v1.begin(),v1.end(),v2.begin(),v2.end(),v3.begin());   // v3[i] = *c

    for ( int i = 0; i < v3.size(); ++i)
        cout << v3[i] << endl;

    return 0;
}
```

Example: inserters

```
#include <iostream>
#include <fstream>
#include <string>
#include <algorithm>
#include <vector>

using namespace std;

int main()
{
    ifstream if1("file1.txt");
    ifstream if2("file2.txt");

    string s;
    vector<string> v1;
    while ( if1 >> s ) v1.push_back(s);
    vector<string> v2;
    while ( if2 >> s ) v2.push_back(s);
    vector<string> v3;
    v3.reserve( v1.size() + v2.size() );    // allocates but not construct, size == 0

    merge(v1.begin(),v1.end(),v2.begin(),v2.end(),back_inserter(v3)); // v3.push_back(*c)

    for ( int i = 0; i < v3.size(); ++i)
        cout << v3[i] << endl;

    return 0;
}
```

Back inserter implementation

```
template<typename _Container>
class back_insert_iterator : public iterator<output_iterator_tag, void, void, void, void>
{
protected:
    _Container* container;
public:
    explicit back_insert_iterator(_Container& __x) : container(&__x) { }

    back_insert_iterator&  operator=(const typename _Container::value_type& __value)
    {
        container->push_back(__value);
        return *this;
    }
    back_insert_iterator&  operator=(typename _Container::value_type&& __value)
    {
        container->push_back(std::move(__value));
        return *this;
    }
    back_insert_iterator&  operator*()      { return *this; }
    back_insert_iterator&  operator++()     { return *this; }
    back_insert_iterator   operator++(int)  { return *this; }
};
template<typename _Container>
inline back_insert_iterator<_Container>  back_inserter(_Container& __x)
{
    return back_insert_iterator<_Container>(__x);
}
```

Example: stream iterator

```
#include <iostream>
#include <fstream>
#include <string>
#include <algorithm>
#include <iterator>     // input- and output-iterators

using namespace std;

int main()
{
    ifstream if1("file1.txt");
    ifstream if2("file2.txt");

    // istream_iterator(if1) -> if1 >> *current
    // istream_iterator() -> EOF
    // ostream_iterator(of,x) -> of << *current << x
    merge( istream_iterator<string>(if1), istream_iterator<string>(),
           istream_iterator<string>(if2), istream_iterator<string>(),
           ostream_iterator<string>(cout,"\n") );

    return 0;
}
```

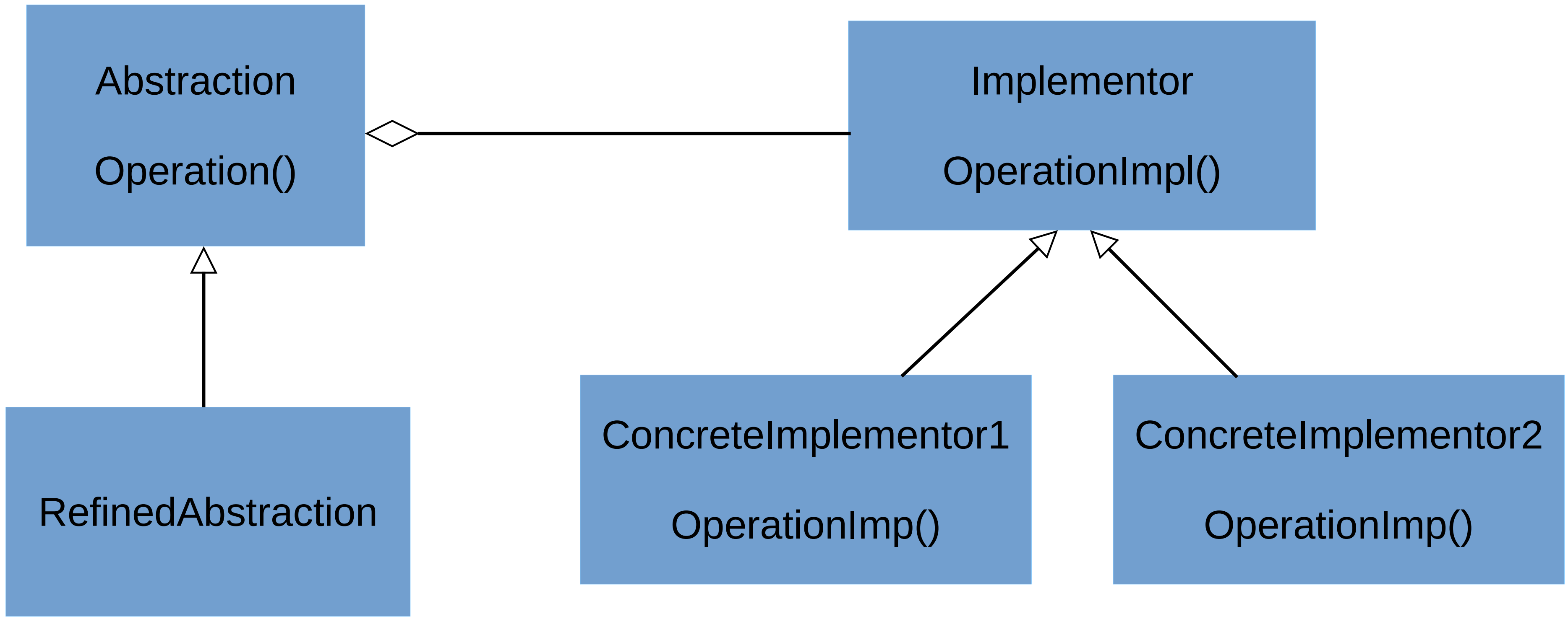
Example: comparator

```
#include <iostream>
#include <fstream>
#include <string>
#include <cctype>
#include <algorithm>
#include <iterator>

struct my_less  // function object: "functor"
{
    bool operator()(const std::string& s1, const std::string& s2)  {
        std::string us1 = s1;
        std::string us2 = s2;
        transform( s1.begin(), s1.end(), us1.begin(), toupper);  // TODO: use <locale>
        transform( s2.begin(), s2.end(), us2.begin(), toupper);
        return us1 < us2;
    }
};

int main()
{
    ifstream if1("file1.txt");
    ifstream if2("file2.txt");

    merge( istream_iterator<string>(if1), istream_iterator<string>(),
           istream_iterator<string>(if2), istream_iterator<string>(),
           ostream_iterator<string>(cout,"\n"), my_less() );
    return 0;
}
```

Bridge

- Intent
  - Decouple an abstraction from its implementation so that the two can vary independently
  - Make possible to work otherwise incompatible interfaces
- Also known: handle/body
- Participants
  - Abstraction, RefineAbstraction, Implementor, ConcreteImplementor(s)

Bridge

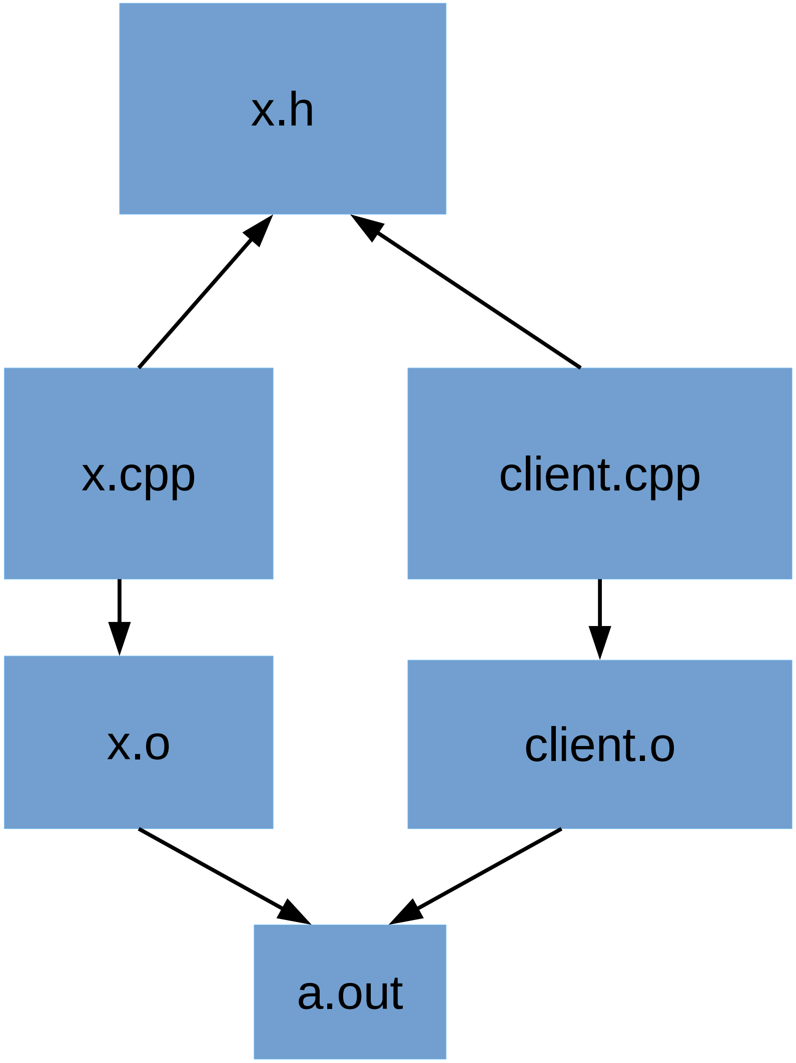


Abstraction
Operation()
Implementor
OperationImpl()
RefinedAbstraction
ConcreteImplementor1
OperationImp()
ConcreteImplementor2
OperationImp()

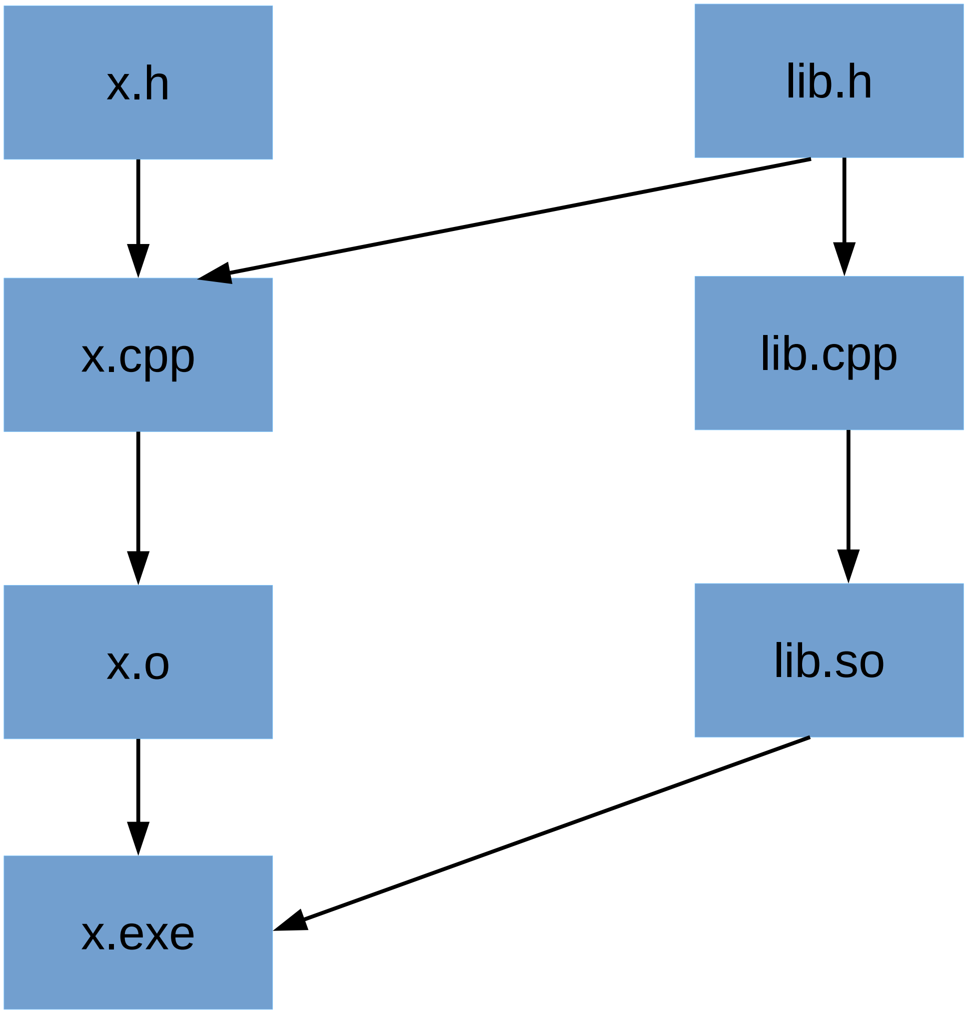
Discussion

- PIMPL
- Binary compatibility
- Compilation firewall (opaque type)

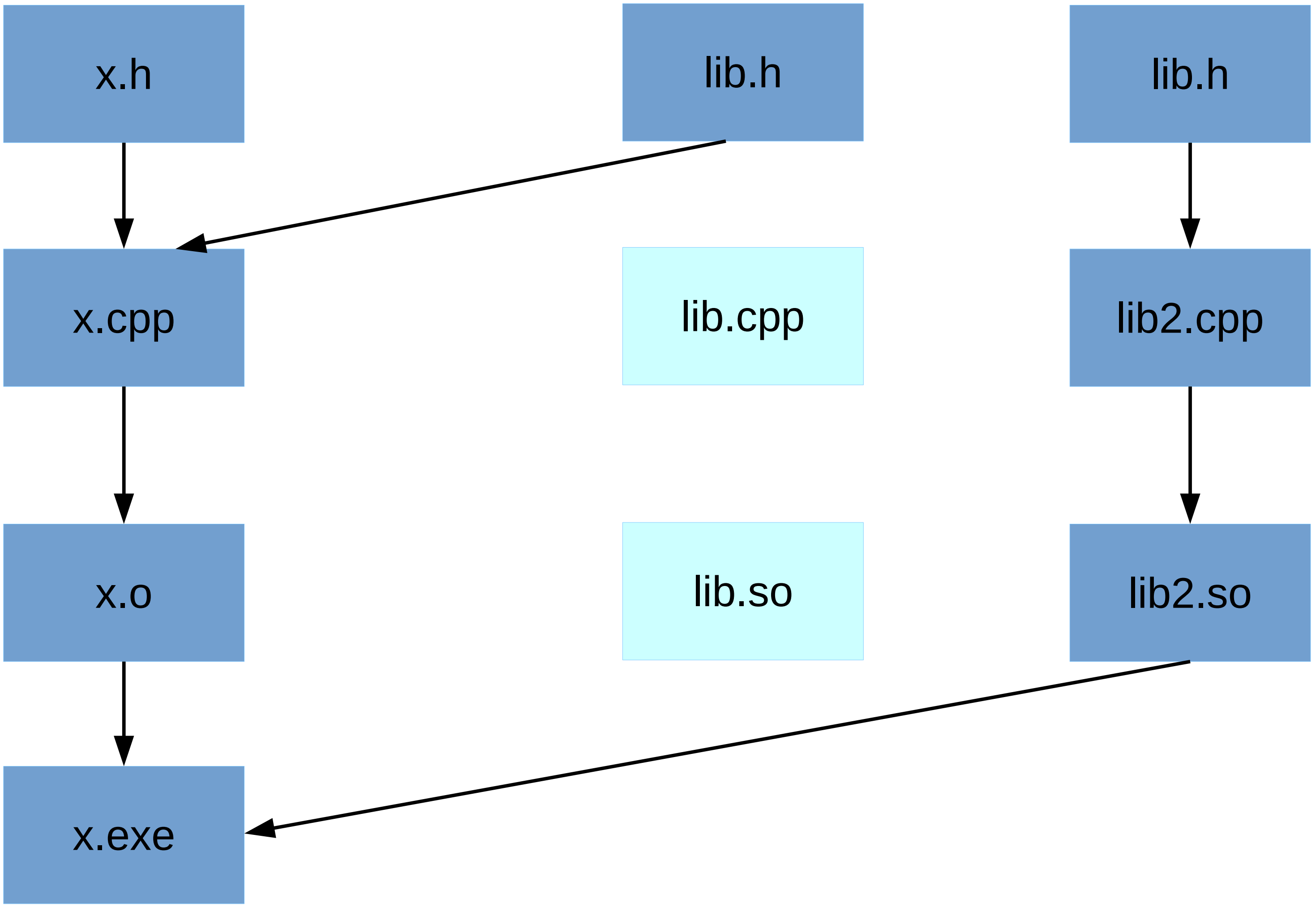
PIMPL



Binary compatibility



Binary compatibility



PIMPL

```
// file x.h
class X
{
    // public and protected members
private:
    // pointer to forward declared class
    struct XImpl;
    XImpl *pimpl_;  // opaque pointer
};



// file ximpl.cpp
struct X::XImpl // not neccessary to declare as "class"
{
    // private members; fully hidden
    // can be changed at without
    // recompiling clients
};
```

Headers with PIMPL

```
#include <iosfwd>
#include "a.h"      // class A
#include "b.h"      // class B
class C;
class E;
class X : public A, private B
{
public:
        X( const C&);
    B   f(int, char*);
    C   f(int, C);
    C&  g(B);
    E   h(E);
    virtual std::ostream& print(std::ostream&) const;
private:
    struct Ximpl;
    XImpl *pimpl_; // opaque pointer to forward-declared class
};


//========= file x.cpp ========================
#include <list>
#include "x.h"
#include "c.h"      // class C
#include "d.h"      // class D
struct Ximpl
{
    std::list<C>    clist_;
    D               d_;
};
```

PIMPL with unique_ptr

```
// file x.h
class X
{
public:
    X();
private:
    // pointer to forward declared class
    struct Ximpl;
    unique_ptr<Ximpl> pimpl_;  // opaque pointer
};

// file ximpl.cpp
struct X::XImpl   // not neccessary to declare as "class"
{
    X::X()
    // can be changed at without
    // recompiling clients
};
X::X() : pimpl_(std::make_unique<XImpl>()) {}
```

PIMPL with unique_ptr

```
#include “x.h”

X  xObj;   // Error: incomplete type!
```

PIMPL with unique_ptr

```
// file x.h
class X
{
public:
    X();
    ~X();  // declaration only!
private:
    // pointer to forward declared class
    struct Ximpl;
    unique_ptr<Ximpl> pimpl_;  // opaque pointer
};

// file ximpl.cpp
struct X::XImpl   // not neccessary to declare as "class"
{
    X::X()
    // can be changed at without
    // recompiling clients
};
X::X() : pimpl_(std::make_unique<XImpl>()) {}
X::~X() {}  // definition, also works: X::~X() = default
```

PIMPL with unique_ptr

```
// file x.h
class X
{
public:
    X();
    ~X();
    X(X&& rhs);
    X& operator=(X&& rhs);
private:
    // pointer to forward declared class
    struct Ximpl;
    unique_ptr<Ximpl> pimpl_;  // opaque pointer
};

// file ximpl.cpp
X::X() : pimpl_(std::make_unique<XImpl>()) {}
X::~X() = default;
X::X(X&& rhs) = default;
X& X::operator=(X&& rhs) = default;
```

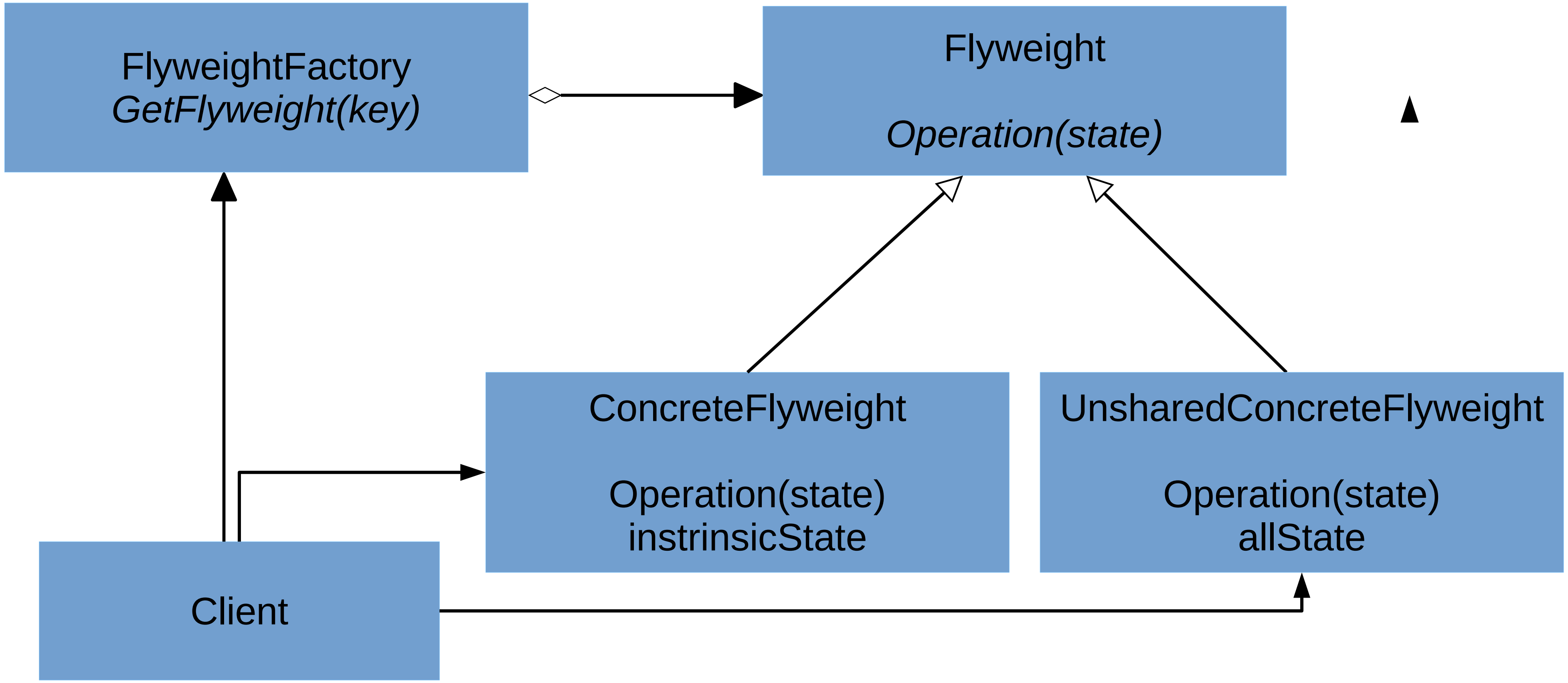
PIMPL with unique_ptr

```
// file x.h
class X
{
public:
    X();
    ~X();
    X(X&& rhs);
    X& operator=(X&& rhs);
    X(const X& rhs);
    X& operator=(const X& rhs);
private:
    // pointer to forward declared class
    struct Ximpl;
    unique_ptr<Ximpl> pimpl_;  // opaque pointer
};

// file ximpl.cpp
X::X(const X& rhs) : pimpl_(std::make_unique<XImpl>(*rhs.pImpl_) {}
X& X::operator=(const X& rhs){ *pImpl_ = *rhs.pImpl_; return *this; }
```

Fast PIMPL

```
// file x.h
class X
{
  // public and protected members
private:
  static const size_t XImplSize = 128;
  char ximpl_[XImplSize]; // instead opaque pointer
};

// file ximpl.cpp
struct XImpl   // not neccessary to declare as "class"
{
  Ximpl::Ximpl(X *tp) : _self(tp) {
    static_assert (XImplSize >= sizeof(XImpl));
    // ...
  };
  X *_self; // might be different than XImpl::this
};
X::X()  { new (ximpl_) XImpl(this); }
X::~X() { (reinterpret_cast<XImpl*>(ximpl_)->~XImpl(); }
```

Fast PIMPL

- ```
  // file x.h
  class X
  {
    // public and protected members
  private:
    static const size_t XImplSize = 128;
    alignas(std::max_align_t) char ximpl_[XImplSize];
  };

  // file ximpl.cpp
  struct XImpl   // not neccessary to declare as "class"
  {
    Ximpl::Ximpl(X *tp) : _self(tp) {
      static_assert (XImplSize >= sizeof(XImpl));
      // ...
    };
    X *_self; // might be different than XImpl::this
  };
  X::X()  { new (ximpl_) XImpl(this); }
  X::~X() { (reinterpret_cast<XImpl*>(ximpl_)->~XImpl(); }
  ```
  ```

Flyweight

- Intent
  - Use sharing to support large number of fine-grained objects efficiently
- Also known: -
- Applicability
  - Application uses a large number of objects
  - Many object shares the same state
- Restrictions
  - The application does not depend on object identity
  - Distinct objects may be shared, equality may true for them

Flyweight


FlyweightFactory
GetFlyweight(key)
Flyweight
Operation(state)
ConcreteFlyweight
Operation(state)
instrinsicState
UnsharedConcreteFlyweight
Operation(state)
allState
Client

Usage of Boost.flyweight

```
#include <string>
#include <vector>

using namespace boost::flyweights;

struct person
{
  int id_;
  std::string city_;
};

int main()
{
  std::vector<person> persons;

  for (int i = 0; i < 100000; ++i)
    persons.push_back({i, "Budapest"});
};
```

Usage of Boost.flyweight

```
#include <boost/flyweight.hpp>
#include <string>
#include <vector>
#include <utility>

using namespace boost::flyweights;

struct person
{
  person(int id, std::string city) : id_{id}, city_{std::move(city)} {}

  int id_;
  flyweight<std::string> city_;
};

int main()
{
  std::vector<person> persons;

  for (int i = 0; i < 100000; ++i)
    persons.push_back({i, "Budapest"});
};
```

Usage of Boost.flyweight

```
#include <boost/flyweight.hpp>
#include <string>
#include <vector>
#include <utility>

using namespace boost::flyweights;

struct person
{
  person(int id, std::string city, std::string country) :
       id_{id}, city_{std::move(city)}, country_{std::move(country)} {}

  int id_;
  flyweight<std::string> city_;      // the same container of strings
  flyweight<std::string> country_;   // will be used for city and country
};

int main()
{
  std::vector<person> persons;

  for (int i = 0; i < 100000; ++i)
    persons.push_back({i, "Budapest", "Hungary"});
};
```

Usage of Boost.flyweight

```
#include <boost/flyweight.hpp>
#include <string>
#include <vector>
#include <utility>

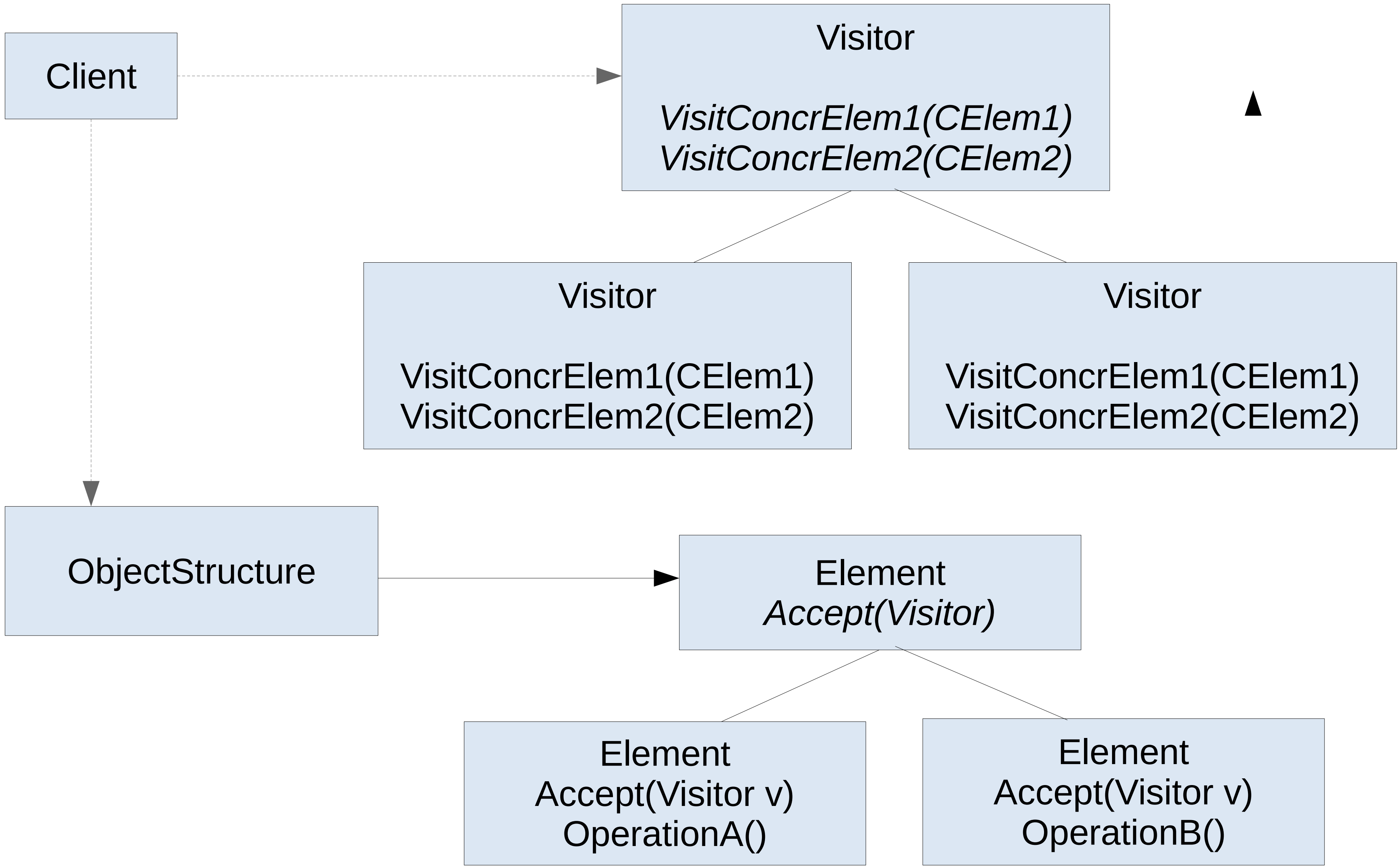
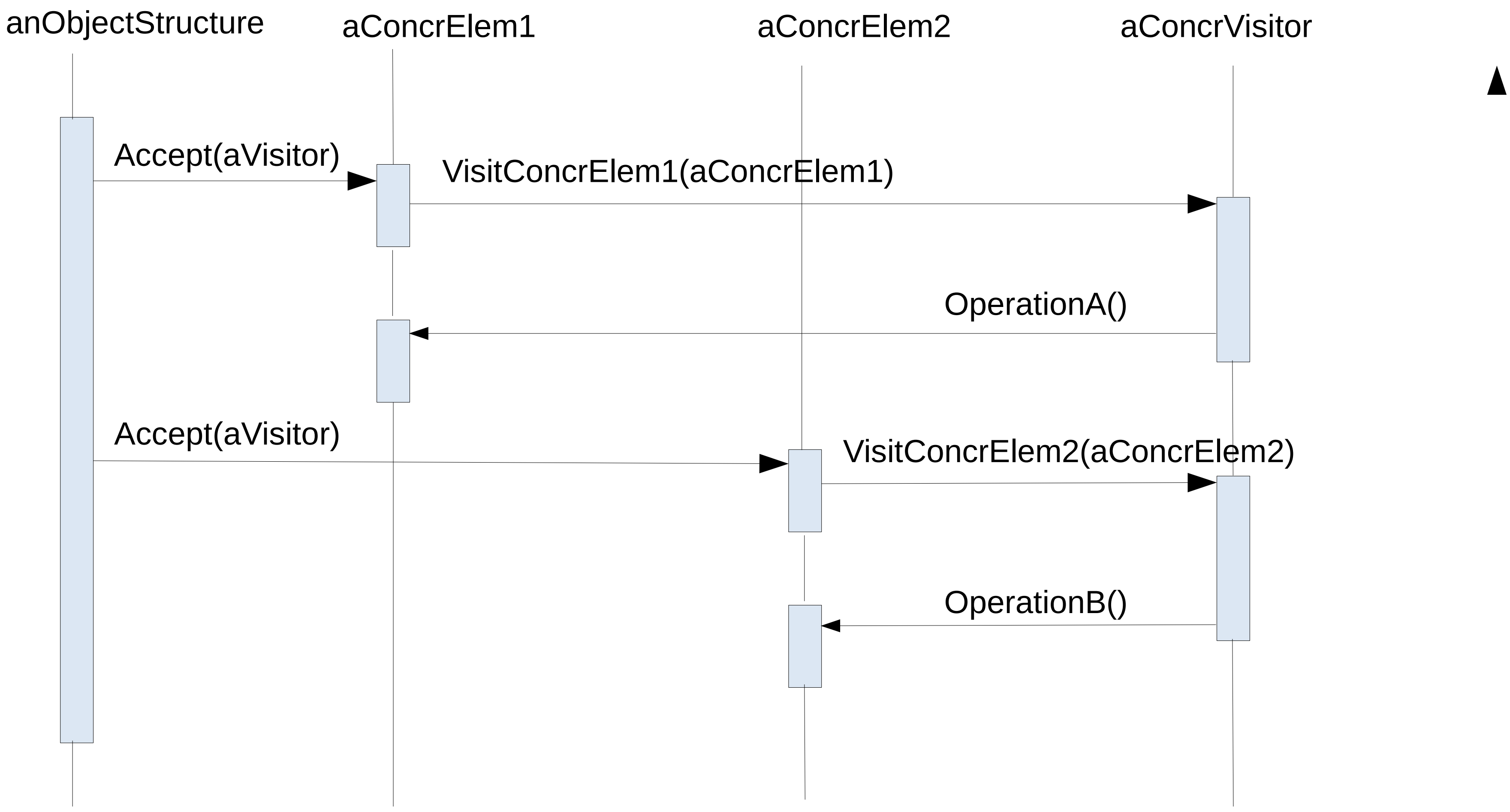
using namespace boost::flyweights;

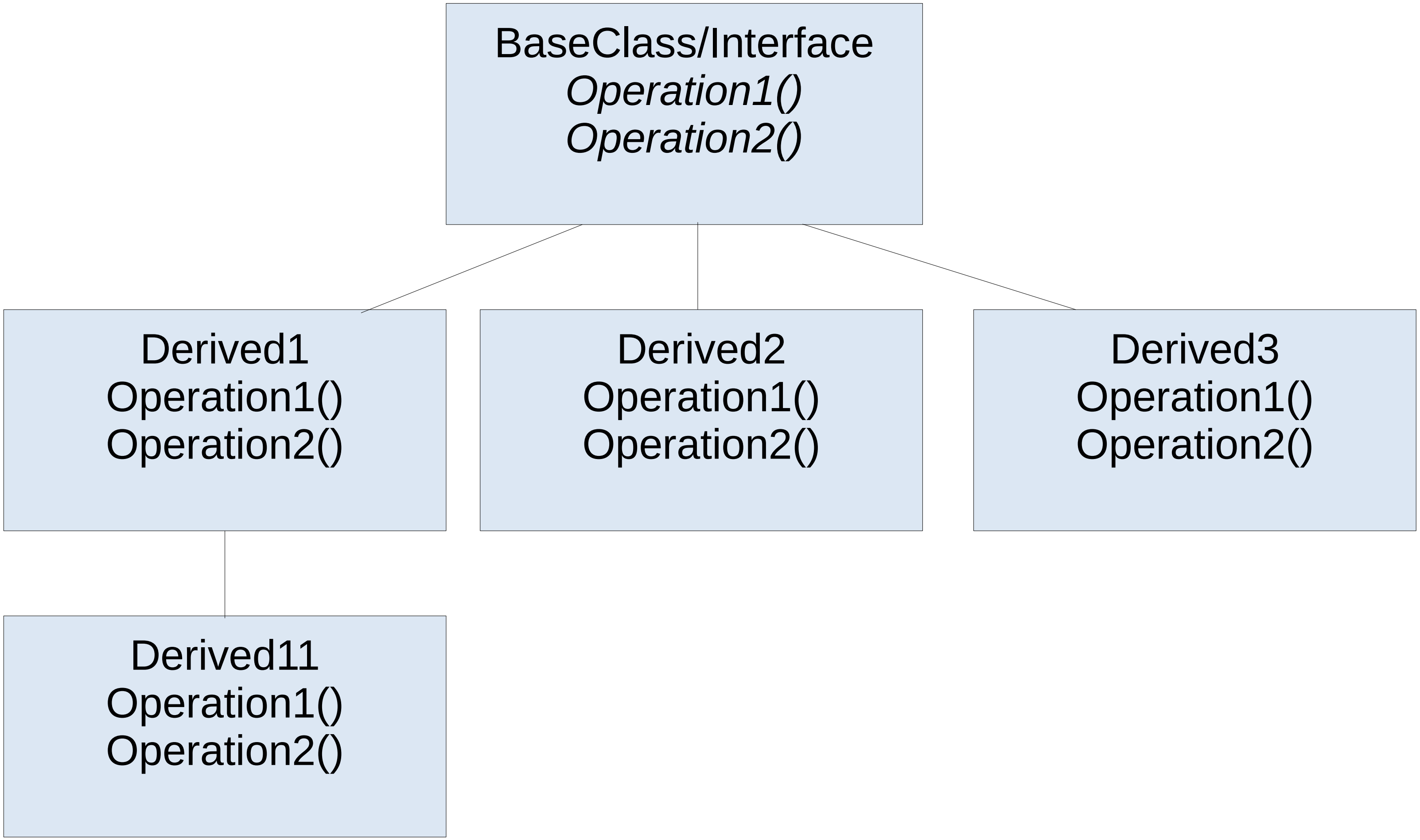
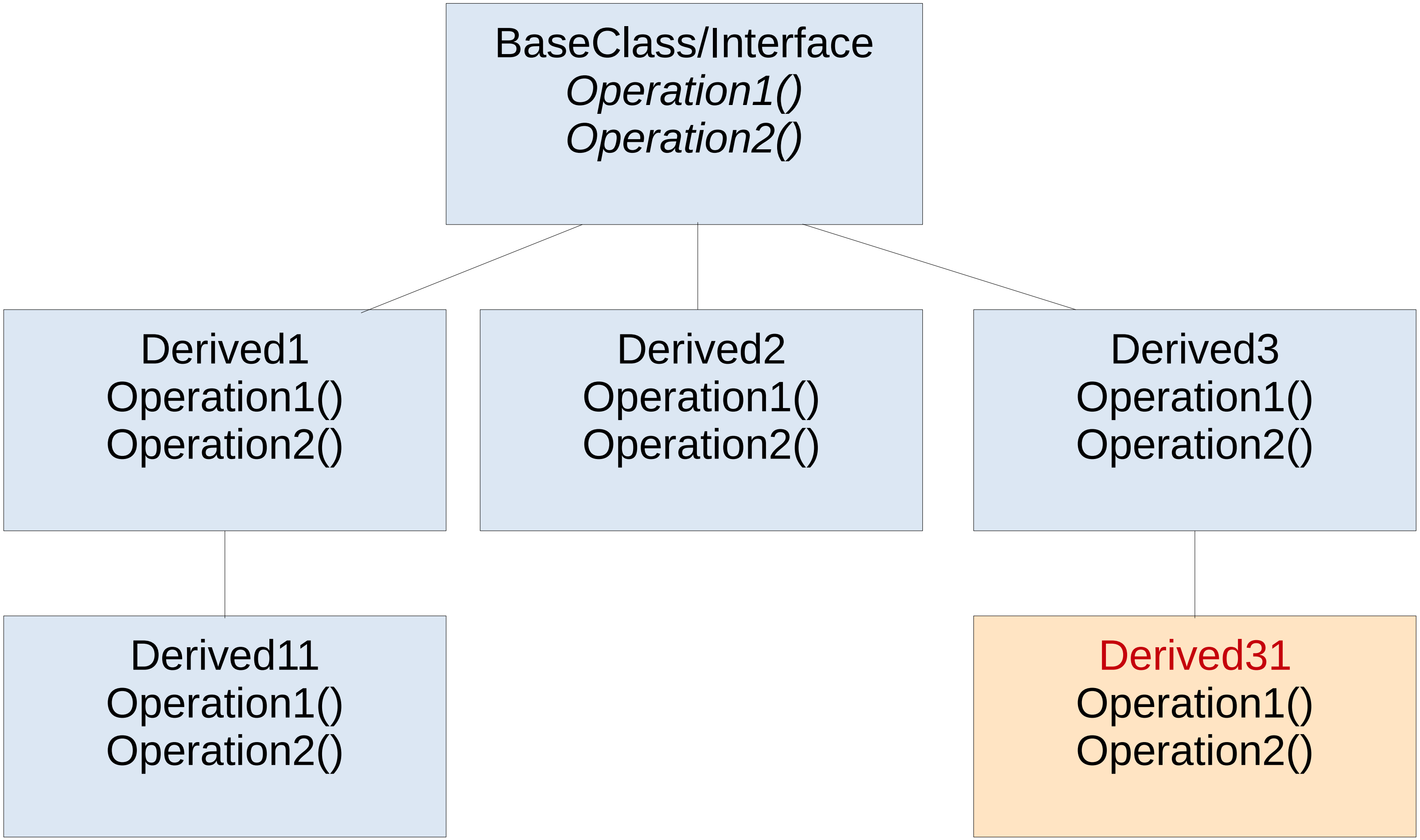
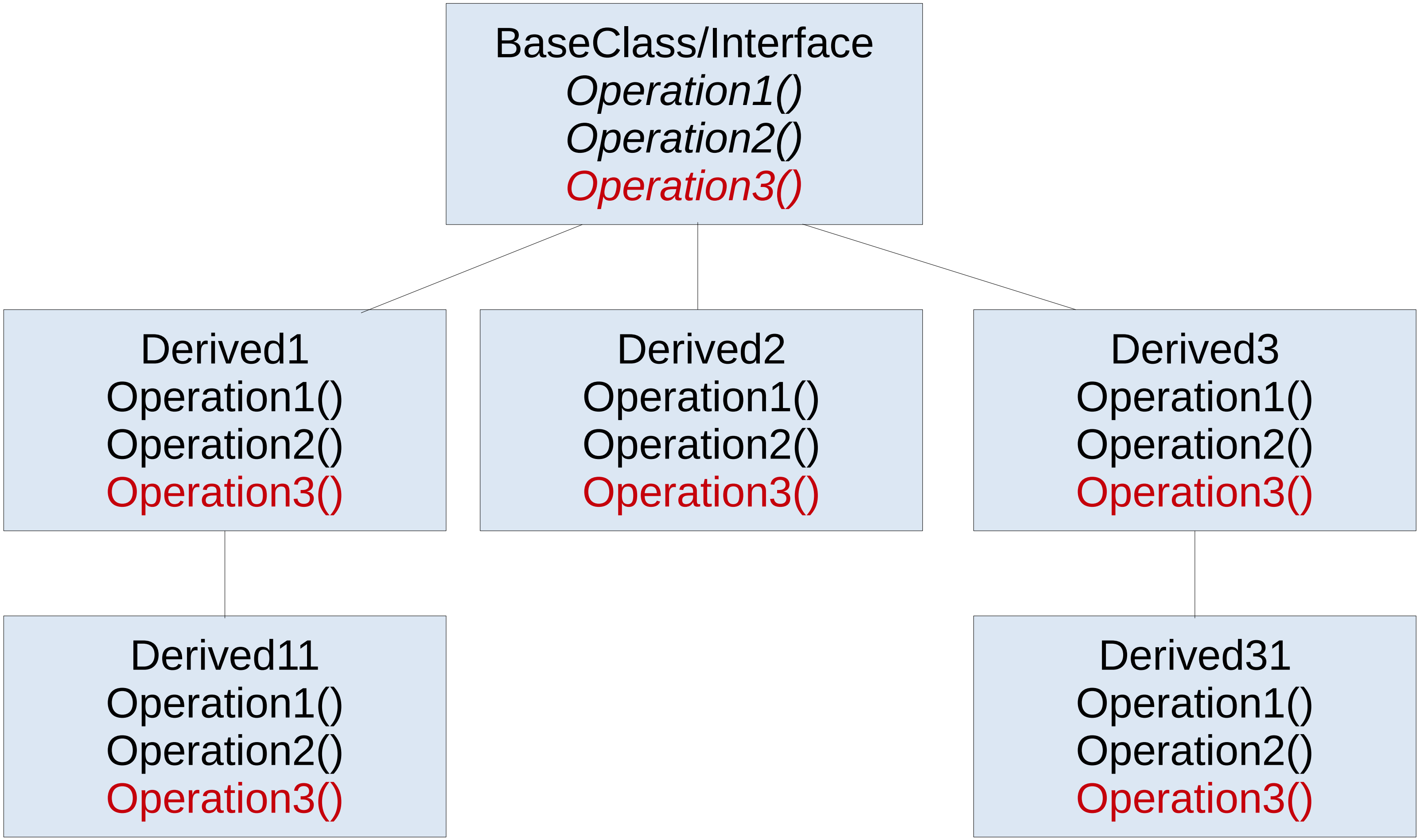
struct city_tag {};
struct country_tag {};

struct person
{
  person(int id, std::string city, std::string country) :
      id_{id}, city_{std::move(city)}, country_{std::move(country)} {}

  int id_;
  flyweight<std::string, tag<city_tag>    city_;      // different containers
  flyweight<std::string, tag<country_tag> country_;   // for city and country
};

int main()
{
  std::vector<person> persons;

  for (int i = 0; i < 100000; ++i)
    persons.push_back({i, "Budapest", "Hungary"});
};
```

Visitor

- Intent
  - Represent an operation to be performed on the elements of an object structure. Visitor let you define a new operation without changing the classes of the elements on which it operates
- Also known: -
- Participants
  - Visitor, ConcreteVisitor, Element, ConcreteElement, ObjectStructure

Visitor


Client
Visitor
VisitConcrElem1(CElem1)
VisitConcrElem2(CElem2)
Visitor
VisitConcrElem1(CElem1)
VisitConcrElem2(CElem2)
Visitor
VisitConcrElem1(CElem1)
VisitConcrElem2(CElem2)
ObjectStructure
Element
Accept(Visitor)
Element
Accept(Visitor v)
OperationA()
Element
Accept(Visitor v)
OperationB()

Applicability

- Visitor
  - Object structure contains many classes with differing interfaces, you want to perform operations on these objects that depend on the concrete classes
  - Many unrelated operations need to be performed on the object structure
  - We do not want to “pollute” the classes with these operations
  - If many applications use the object structure, visitor can be used only by applications require the operations
  - The object structure is stable, new operations appear frequently

Collaboration diagram



Discussion

- Visitor makes adding new operations easy to a full object structure
  - Simply adding a new Visitor
- Visitor gathers related operations and separate unrelated ones
  - Related operations are defined in the same Visitor hierarchy
- Adding new ConcreteElement classes is hard
  - New abstract operation is required to add to Visitor
  - Sometimes default behavior is ok, but mostly not works
- Visiting across class hierarchies.
- Visitors can accumulate state
- Breaking encapsulation

The expression problem

```
class Expr {   // expression evaluator: interpreter design pattern
public:
  virtual std::string ToString() const = 0;  // string representation
  virtual double Eval() const = 0;           // evaluating the expression
};

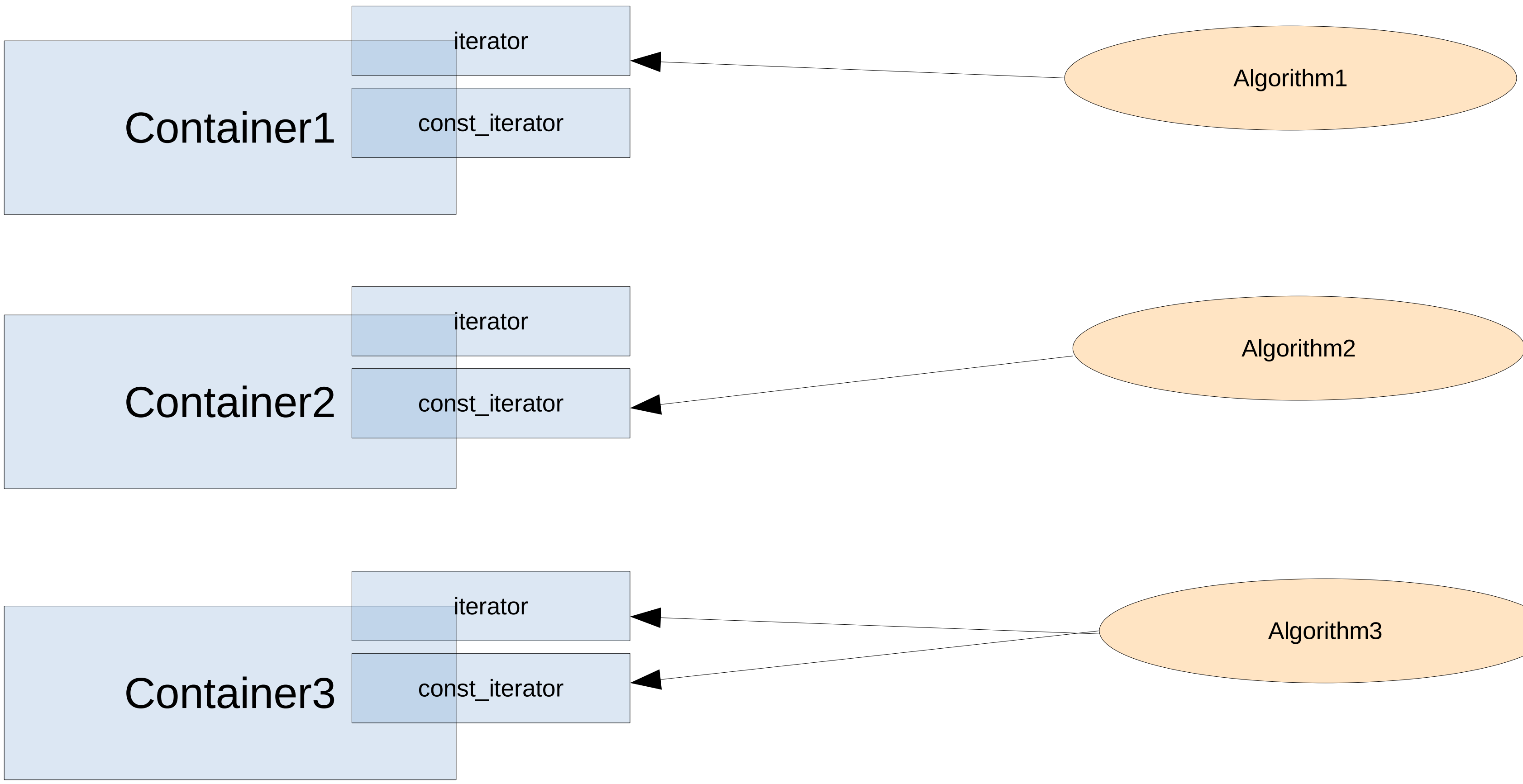
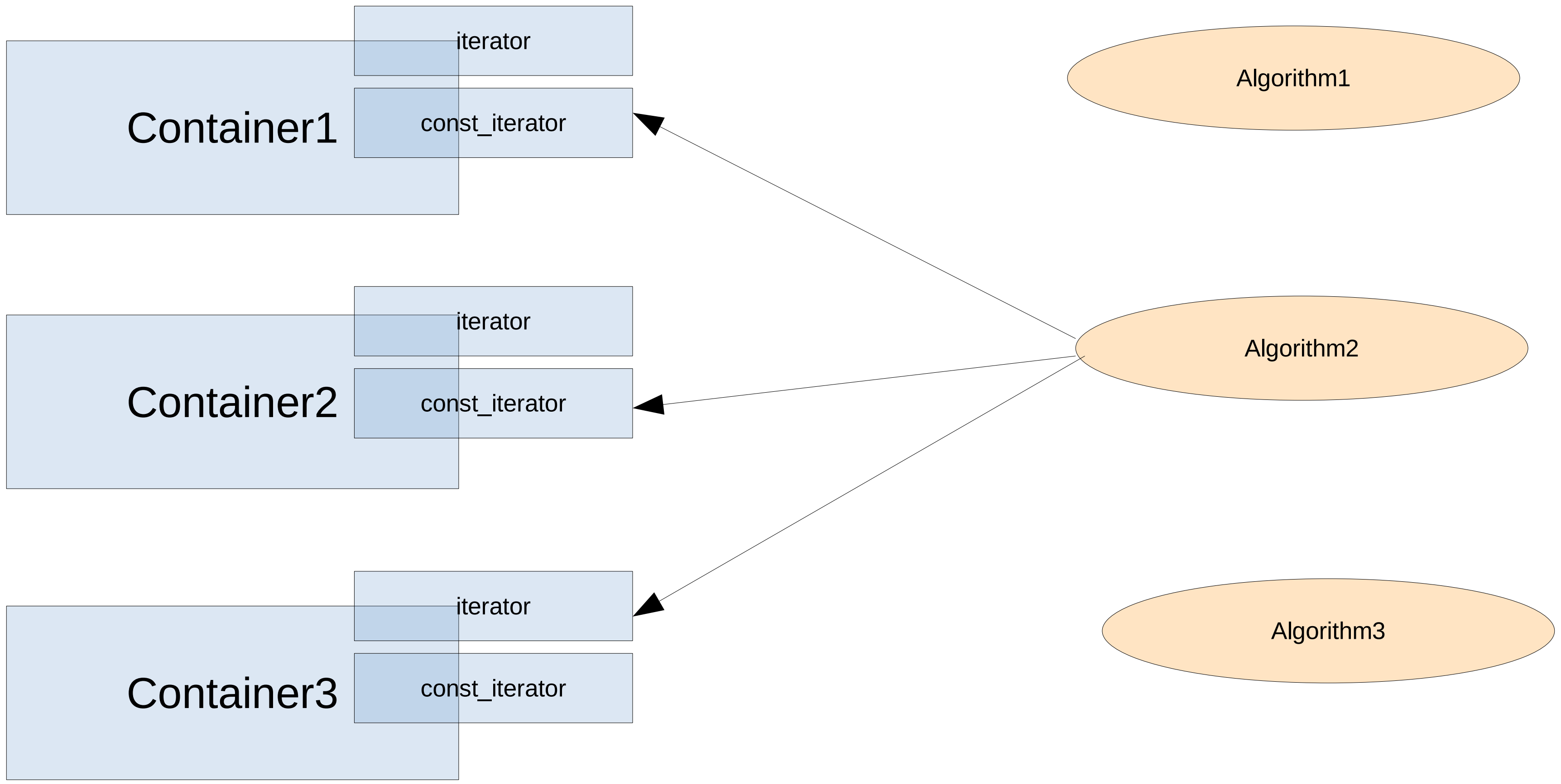
class Constant : public Expr {
public:
  Constant(double value) : value_(value) {}
  std::string ToString() const {
    std::ostringstream ss;
    ss << value_;
    return ss.str();
  }
  double Eval() const {
    return value_;
  }
private:
  double value_;
};
```

The expression problem

```
class BinaryPlus : public Expr {
public:
  BinaryPlus(const Expr& lhs, const Expr& rhs) : lhs_(lhs), rhs_(rhs) {}
  std::string ToString() const {
    return lhs_.ToString() + " + " + rhs_.ToString();
  }
  double Eval() const {
    return lhs_.Eval() + rhs_.Eval();
  }
private:
  const Expr& lhs_;
  const Expr& rhs_;
};
```

- How to extend this system?
  - Adding new expression types: BinaryMinus, Function call,
  - Adding new operations: TypeCheck, Serialize, Compile

The expression problem



The expression problem



The expression problem



The expression problem

- Philip Wadler: expression problem mail, 1990
- Shriram Krishnamurthi, Matthias Felleisen, Daniel P. Friedman: "Synthesizing Object-Oriented and Functional Design to Promote Re-Use", 1998

Solving the expression problem with Visitor

```
class ExprVisitor {
public:
  virtual void VisitConstant(const Constant& c) = 0;
  virtual void VisitBinaryPlus(const BinaryPlus& bp) = 0;
};

class Expr {
public:
  virtual void Accept(ExprVisitor* visitor) const = 0;
};

class Constant : public Expr {
public:
  Constant(double value) : value_(value) {}
  void Accept(ExprVisitor* visitor) const {
    visitor->VisitConstant(*this);
  }
  double GetValue() const {
    return value_;
  }
private:
  double value_;
};
```

Solving the expression problem with Visitor

```
class Evaluator : public ExprVisitor
{
public:
  double GetValueForExpr(const Expr& e) {
    return value_map_[&e];
  }
  void VisitConstant(const Constant& c) {
    value_map_[&c] = c.GetValue();
  }
  void VisitBinaryPlus(const BinaryPlus& bp) {
    bp.GetLhs().Accept(this);
    bp.GetRhs().Accept(this);
    value_map_[&bp] = value_map_[&(bp.GetLhs())]+value_map_[&(bp.GetRhs())];
  }
private:
  std::map<const Expr*, double> value_map_;
};
```

Solving the expression problem with Visitor: extension

```
class Evaluator : public ExprVisitor {  /* the same as earlier */ };

// This is the new ("extended") expression we're adding.
class FunctionCall : public Expr
{
public:
  FunctionCall(const std::string& name, const Expr& argument)
                                    : name_(name), argument_(argument) {}
  void Accept(ExprVisitor* visitor) const {
    ExprVisitorWithFunctionCall* v =
                       dynamic_cast<ExprVisitorWithFunctionCall*>(visitor);
    if (v == nullptr) {
      std::cerr << "Fatal: visitor is not ExprVisitorWithFunctionCall\n";
      exit(1);
    }
    v->VisitFunctionCall(*this);
  }
private:
  std::string name_;
  const Expr& argument_;
};
```

Solving the expression problem with Visitor: extension

```
class ExprVisitorWithFunctionCall : virtual public ExprVisitor
{
public:
  virtual void VisitFunctionCall(const FunctionCall& fc) = 0;
};



// Inherits Evaluator and implements ExprVisitorWithFunctionCall interface
class EvaluatorWithFunctionCall : public ExprVisitorWithFunctionCall,
                                  public Evaluator
{
public:
  void VisitFunctionCall(const FunctionCall& fc) {
    std::cout << "Visiting FunctionCall!!\n";
  }
};
```

The STL solution



The STL solution



std::variant

```
void f()
{
    std::variant<std::string, double> s1("Hello");      // conversion works
    std::variant<std::string, const char *> s2("Hello");// choose const char *

    s1 = "Hallo";  // conversion when non-ambigous

    std::cout << std::boolalpha
              << "variant holds double? "
              << std::holds_alternative<double>(s1) << '\n'
              << "variant holds string? "
              << std::holds_alternative<std::string>(s1) << '\n';
        //    << std::holds_alternative<int>(s1) << '\n';  // compile error
}

variant holds double? false
variant holds string? true
```

std::visit

```
using var_t = std::variant<int, long, double, std::string>; // the variant to visit
template<class T> struct always_false : std::false_type {}; // helper type for the visitor

int main()
{
    std::vector<var_t> vec = {10, 15l, 1.5, "hello"};
    for(auto& v: vec)
    {
        // 1. void visitor, only called for side-effects (here, for I/O)
        std::visit([](auto&& arg){std::cout << arg;}, v);

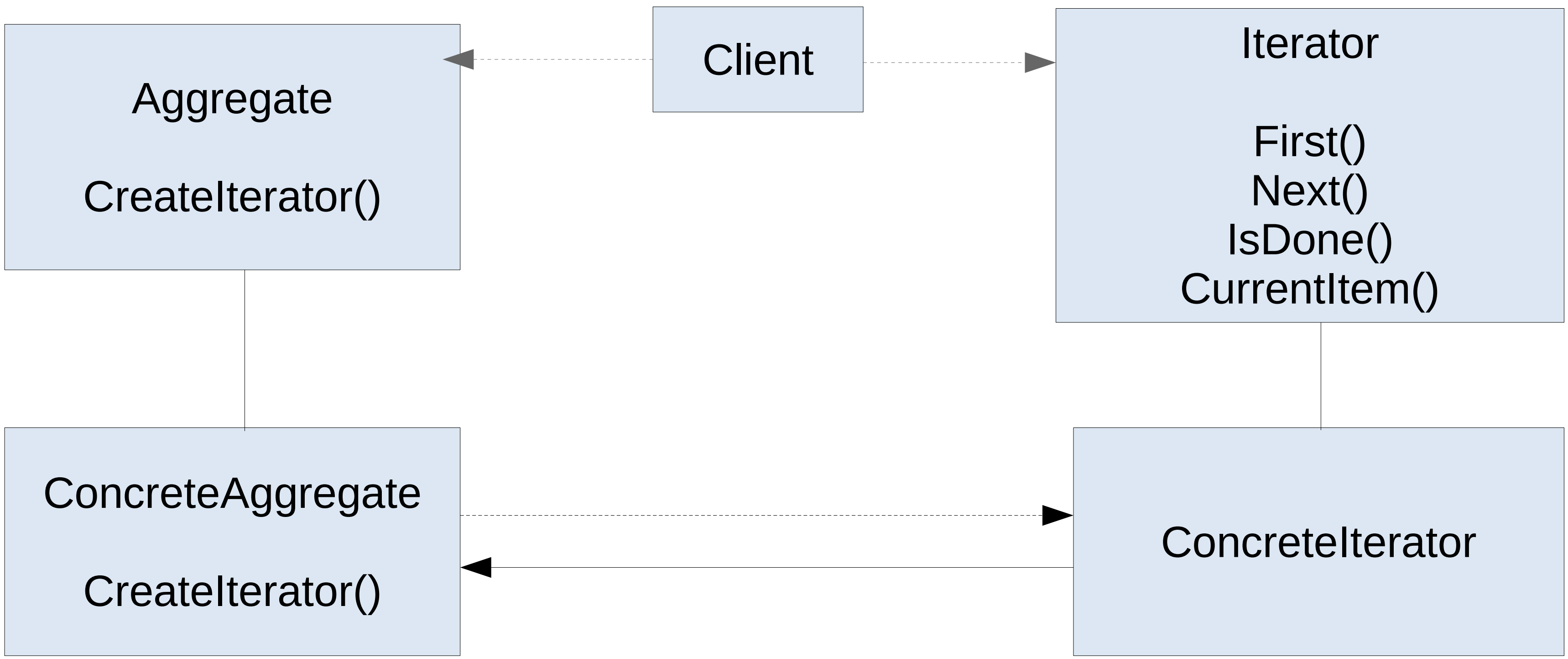
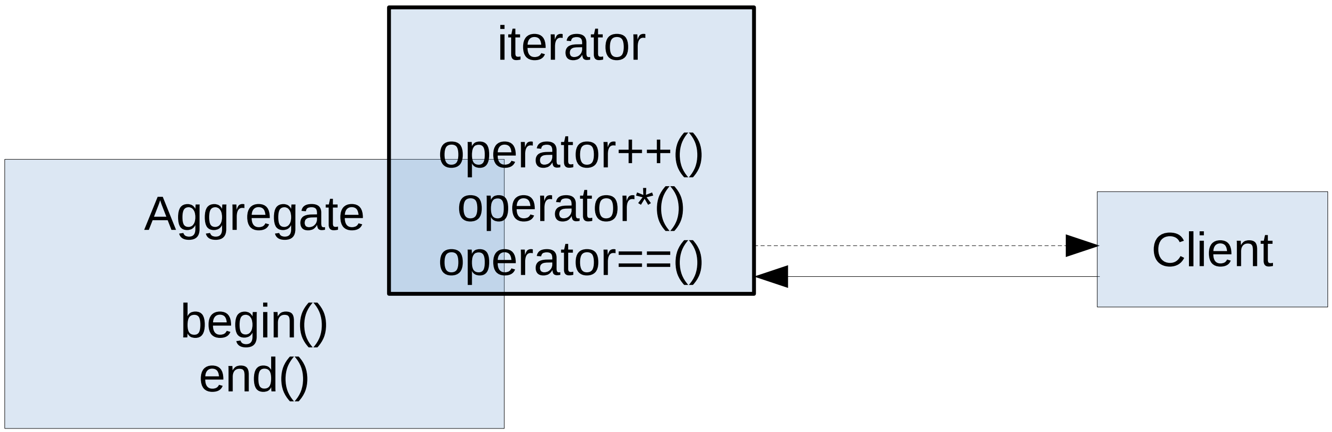
        // 2. value-returning visitor, demonstrates the idiom of returning another variant
        var_t w = std::visit([](auto&& arg) -> var_t {return arg + arg;}, v);

        // 3. type-matching visitor: a lambda that handles each type differently
        std::cout << ". After doubling, variant holds ";
        std::visit([](auto&& arg) {
            using T = std::decay_t<decltype(arg)>;
            if constexpr (std::is_same_v<T, int>)
                std::cout << "int with value " << arg << '\n';
            else if constexpr (std::is_same_v<T, long>)
                std::cout << "long with value " << arg << '\n';
            else if constexpr (std::is_same_v<T, double>)
                std::cout << "double with value " << arg << '\n';
            else if constexpr (std::is_same_v<T, std::string>)
                std::cout << "std::string with value " << std::quoted(arg) << '\n';
            else
                static_assert(always_false<T>::value, "non-exhaustive visitor!");
        }, w);
    }
}
```

std::visit

```
using var_t = std::variant<int, long, double, std::string>; // the variant to visit

// helper type for the visitor
template<class... Ts> struct overloaded : Ts... { using Ts::operator()...; };
template<class... Ts> overloaded(Ts...) -> overloaded<Ts...>;

int main()
{
    std::vector<var_t> vec = {10, 15L, 1.5, "hello"};

    for (auto& v: vec)
    {
        // type-matching visitor: a class with 3 overloaded operator()'s
        std::visit(overloaded {
            [](auto arg) { std::cout << arg << ' '; },
            [](double arg) { std::cout << std::fixed << arg << ' '; },
            [](const std::string& arg) { std::cout << std::quoted(arg) << ' '; },
        }, v);
    }

}
```

```
10 15 1.500000 “hello”
```

Iterator

- Intent
  - Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation
- Also known: Cursor
- Participants
  - Iterator, ConcreteIterator, Aggregate, ConcreteAggregate

Iterator


Aggregate
CreateIterator()
Client
Iterator
First()
Next()
IsDone()
CurrentItem()
ConcreteAggregate
CreateIterator()
ConcreteIterator

Applicability

- Iterator
  - Supports variations in the traversal of an Aggregate. Replacing the iterator instance may apply a different traversal (e.g. reverse iterators).
  - Simplify the interface of Aggregate
  - Multiple traversals can be pending on the Aggregate

Discussion

- Who controls the iteration?
  - External iterator: the client controls the iteration
  - Internal iterator: the iterator controls the iteration
  - External iterators are more flexible, internal iterators easier to use
- Who defines the traversal algorithm?
  - The aggregation: cursor
  - Iterator: how to access the private data of the aggregate?
- Polymorphic iterator?
- Iterator invalidation?

Iterator in STL



Simple solution

```
int t[] = { 1, 3, 5, ... };

// find the first occurance of a value
int *pi = find( t, t+sizeof(t)/sizeof(t[0]), 55);

if ( pi )
{
  *pi = 56
}

// a very specific algorithm
int *find( int *begin, int *end, int x)
{
  while ( begin != end )
  {
    if ( *begin == x )
    {
      return begin;
    }
    ++begin;
  }
  return 0;
}
```

Simple solution

```
int t[] = { 1, 3, 5, ... };

// find the first occurance of a value
int *pi = find( t, t+sizeof(t)/sizeof(t[0]), 55);

if ( pi )
{
  *pi = 56
}
```

Template based

```
double t[] = { 1.0, 3.14, 5.55, ... };

// find the first occurance of a value
double *pi = find( t, t+sizeof(t)/sizeof(t[0]), 55.5);

if ( pi )
{
  *pi = 56.5
}

// Templated algorithm
template <typename T>
T *find( T *begin, T *end, const T& x)
{
  while ( begin != end )
  {
    if ( *begin == x )
    {
      return begin;
    }
    ++begin;
  }
  return 0;
}
```

Iterator based

```
std::list<int> li = { 1, 3, 5, ... };

// find the first occurance of a value
auto it = find( li.begin(), li.end(), 55);

if ( li.end() != i )
{
  *i = 56
}

// Iterator based algorithm
template <typename It, typename T>
It find( It begin, It end, const T& x)
{
  while ( begin != end )
  {
    if ( *begin == x )
    {
      return begin;
    }
    ++begin;
  }
  return end;  // not 0
}
```

Universal usage

```
std::list<int> li = { 1, 3, 5, ... };
std::vector<double> vd = { 1.0, 3.3, 5.5, ... };

template <typename Container>
auto generic_find( const Container& c, typename Container::value_type& v)
{
  return std::find( c.begin(), c.end(), v);
}

template<class C> typename C::value_type generic_sum(const C& c)
{
  typename C::value_type s{};
  typename C::const_iterator p = c.begin();

  while ( p != c.end() )
  {
    s += *p;
    ++p;
  }
  return s;
}

auto i = generic_find(li, 5);
auto d = generic_sum(vd);
```

Reverse iterator

- ```
  std::list<int> li = { 1, 3, 5, ... };
  std::vector<double> vd = { 1.0, 3.3, 5.5, ... };

  template<class C>
  typename C::iterator find_last(C& c, const typename C::value_type& v)
  {
    typename C::reverse_iterator p = c.rbegin(); // view sequence in revers

    while ( p != c.rend() )
    {
      if ( *p == v )
      {
        typename C::iterator i = p.base();
        return --i;
      }
      ++p;            // note: increment, not decrement (–)
    }
    return c.end();   // use c.end() to indicate "not found”
  }
  ```
  ```

Iterators are const safe

```
//  C++11

std::vector<int>  v1(4,5);
auto i = std::find( v1.begin(), v1.end(), 3);
// i is vector::iterator

const std::vector<int>  v2(4,5);
auto j = std::find( v2.begin(), v2.end(), 3);
// j is vector::const_iterator

auto k = std::find( v1.cbegin(), v1.cend(), 3);
// k is vector::const_iterator
```

Ranges

- There are a number of issues with Iterators
  - The syntax is unnecessary complex
  - Danger of mismatch of iterator pairs
  - Iterator adapters are hard to implement
- There are nonstandard implementations
  - Boost
  - Adobe
  - D language
  - Eric Niebler std::range C++20 candidate

Better composition of STL

```
std::vector<int>  v;

// C++ iterators
std::sort( v.begin(), v.end() );

// C++ ranges
std::sort( v );


// lazy computational pipeline
int total = std::accumulate(
          view::iota(1)  |
          view::transform( [](int x) { return x*x; } ) |
          view::take(10),   0);
```

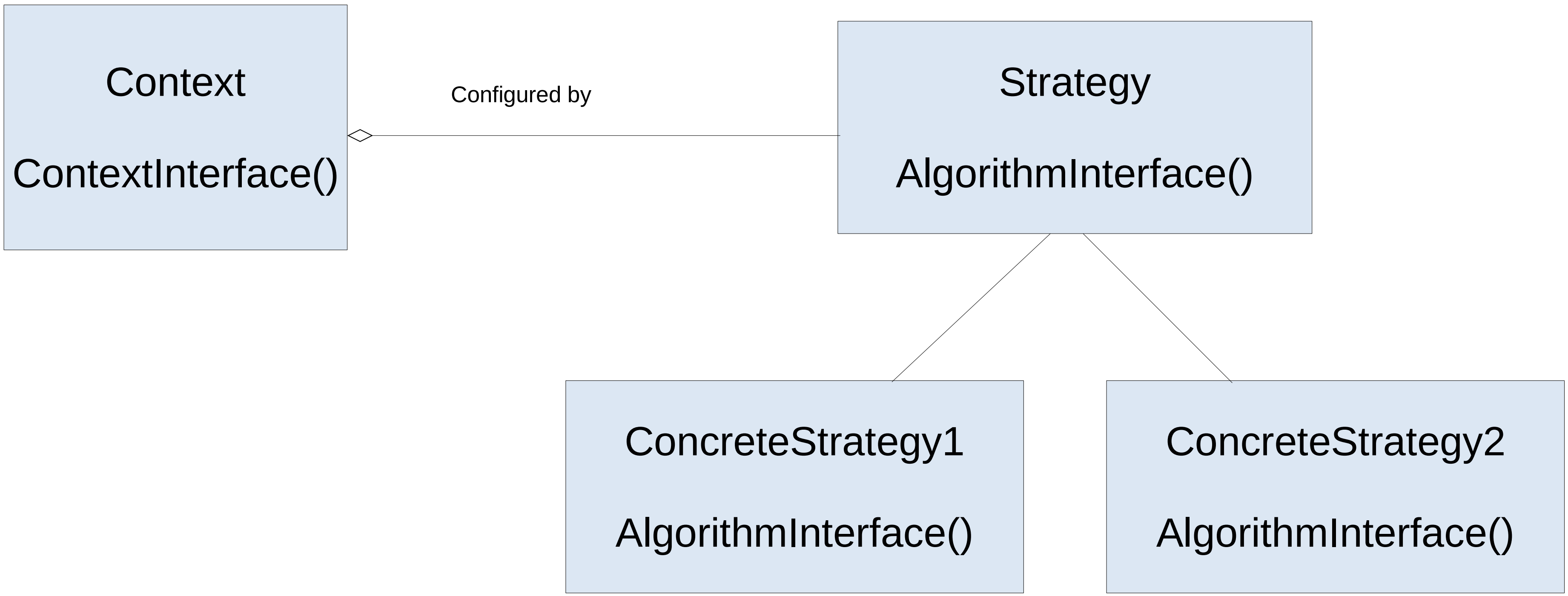
Possible implementations

- Pair of iterators
  - Range [i , j)
  - Valid only if j is reachable from i
- Iterator + count of elements
- Iterator + predicate
  - Predicate may be statefull
  - Indicates when the range exhausted
- Can implement other types
  - Iterator and sentinel element (like ‘\0’)
  - Range of disjoint ranges

Strategy

- Intent
  - Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
- Also known: Policy
- Participants
  - Strategy, ConcreteStrategy, Context

Strategy


Context
ContextInterface()
Configured by
Strategy
AlgorithmInterface()
ConcreteStrategy1
AlgorithmInterface()
ConcreteStrategy2
AlgorithmInterface()

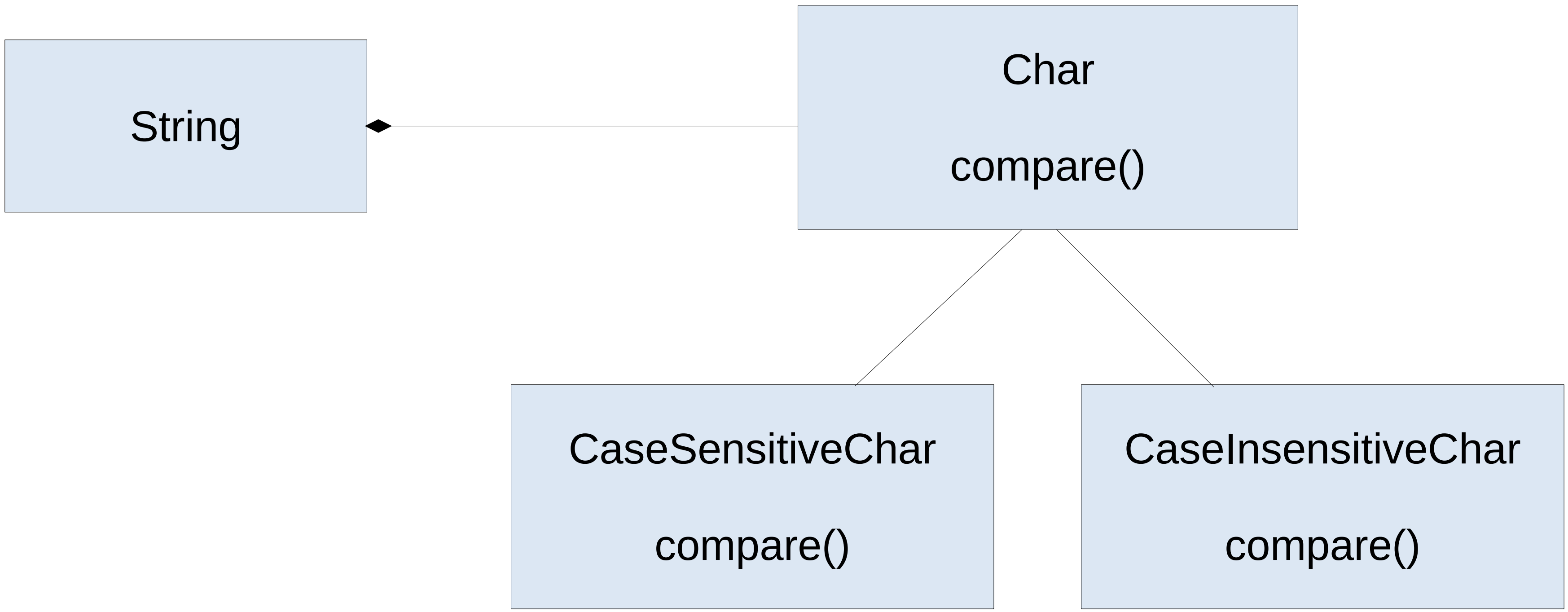
Applicability

- Use the Strategy pattern when
  - Many related classes differ only in their behavior. Strategy provide a way to configure a class with one of many behaviors.
  - We need different variants of an algorithm. Variants can be implemented as a class hierarchy.
  - Algorithm uses data that the client should not know about.
  - Class defines many behaviors and these appear as multiple conditional statements.

Discussion

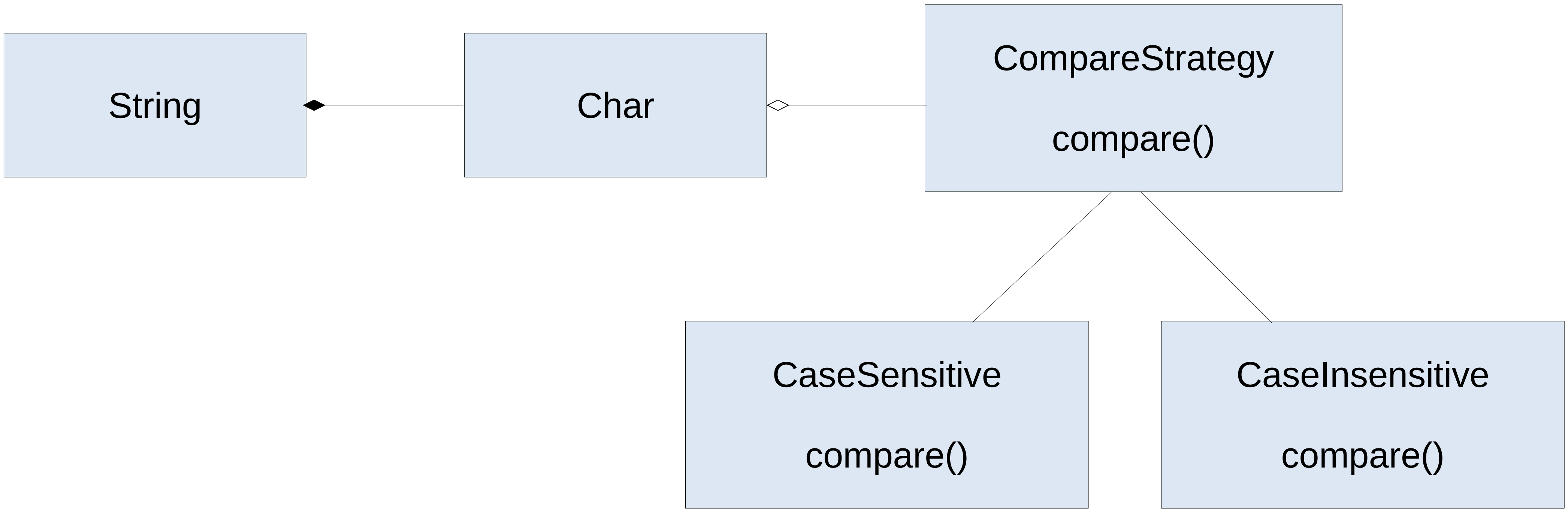
- Families of elated algorithms
- Alternatives: subclassing Context
  - Context could be subclassed and the concrete strategy could be defined there
  - Hardwire the behavior of Context, making Context harder to maintain, not able to vary the algorithm dynamically
- Strategy eliminates conditional statements
- Strategy can be template parameter

Use case example 1: char_trait



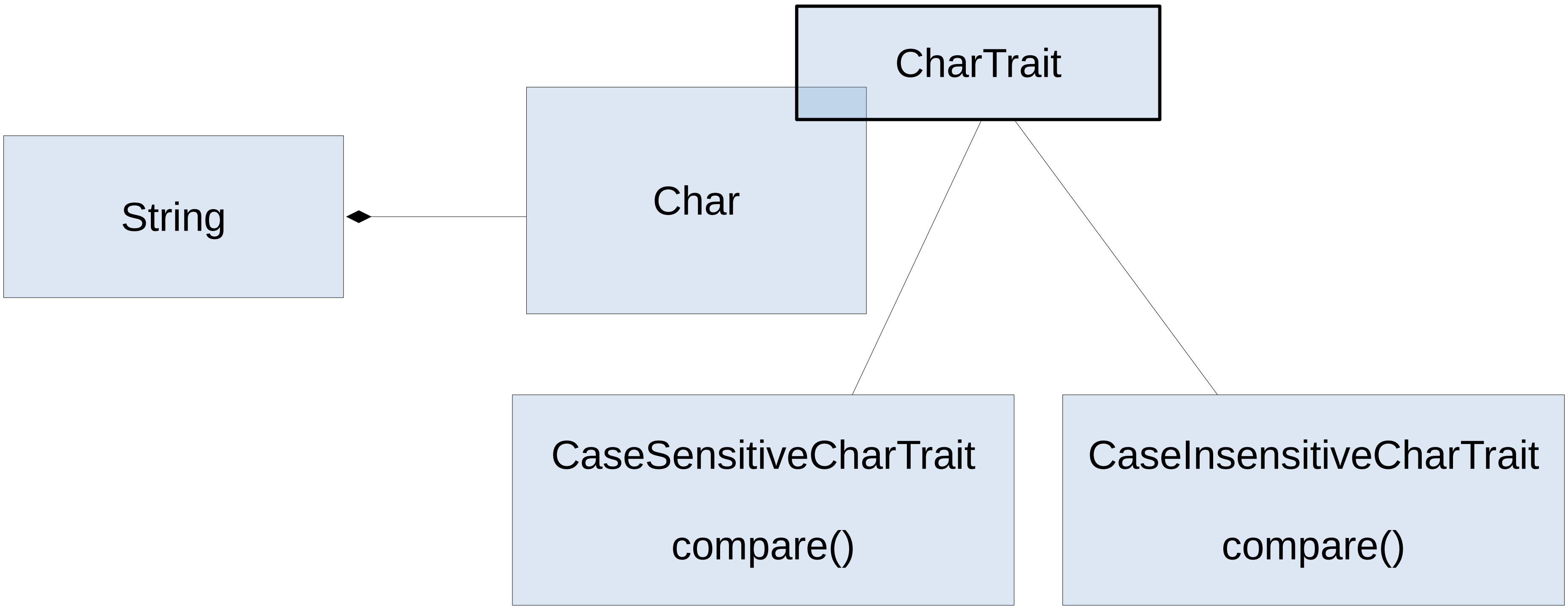
Inheritance based solution

Use case example 1: char_trait



Strategy based solution

Use case example 1: char_trait



Template based solution

Use case example 1: char_trait

```
template<class Ch> struct char_traits { };

template<> struct char_traits<char>
{
  typedef char char_type; // type of character
  static void assign(char_type&, const char_type&); // = for char_type
  typedef int int_type;   // type of integer value of character
  static char_type to_char_type(const int_type&); // int to char conv
  static int_type to_int_type(const char_type&);  // char to int conv
  static bool eq_int_type(const int_type&, const int_type&); // ==
  static bool eq(const char_type&, const char_type&); // ==
  static bool lt(const char_type&, const char_type&); // <

  // operations on s[n] arrays:
  static char_type* move(char_type* s, const char_type* s2, size_t n);
  static char_type* copy(char_type* s, const char_type* s2, size_t n);
  static char_type* assign(char_type* s, size_t n, char_type a);
  static int compare(const char_type* s, const char_type* s2, size_t n);
  static size_t length(const char_type*);
  static const char_type* find(const char_type*,int n,const char_type&);

  // i/o related, like eof(), etc...
};
```

Use case example 1: char_trait

```
template<> struct char_traits<wchar_t>
{
  typedef wchar_t char_type;
  typedef wint_t int_type;
  typedef wstreamoff off_type;
  typedef wstreampos pos_type;

  // like char_traits<char>
};

template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class std::basic_string
{
public:
    // ...
};

using std::string = std::basic_string<char>;
```

Use case example 1: char_trait

```
struct ci_char_traits : public std::char_traits<char>
{
    static bool eq( char c1, char c2) {
        return toupper(c1) == toupper(c2);
    }
    static bool lt( char c1, char c2) {
        return toupper(c1) < toupper(c2);
    }
    static int compare( const char *s1, const char *s2, size_t n) {
        return memicmp( s1, s2, n);     // non-standard !
    }
    static const char *find( const char *s, int n, char ch) {
        while ( n-- > 0  &&  toupper(*s) != toupper(ch) )
        {
            ++s;
        }
        return n > 0 ? s : 0;
    }
};

typedef std::basic_string<char, ci_char_traits> ci_string;
```

Use case example 1: char_trait

```
#include <iostream>
#include <string>
#include "cistring1.h"

using namespace std;

int main()
{
    string s1("hELlo");
    string s2("HEllo");

    cout << boolalpha << (s1 == s2) << endl;

    ci_string cs1("hELlo");
    ci_string cs2("HEllo");

    cout << boolalpha << (cs1 == cs2) << endl;

    return 0;
}
```

Use case example 2: matrix

```
template <class T>
class matrix
{
public:
    matrix( int i, int j );
    matrix( const matrix &other);
    ~matrix();
    matrix operator=( const matrix &other);
private:
    int  x;
    int  y;
    T   *v;
    void copy( const matrix &other);
    void check( int i, int j) const throw(indexError);
};
```

Specialization for POD types

```
template <class T>
void matrix<T>::copy( const matrix &other)
{
    x = other.x;
    y = other.y;
    v = new T[x*y];
    for ( int i = 0; i < x*y; ++i )
       v[i] = other.v[i];
}

// specialization for POD types
template <>
void matrix<long>::copy( const matrix &other)
{
    x = other.x;
    y = other.y;
    v = new long[x*y];
    memcpy( v, other.v, sizeof(long)*x*y);
}

template <>
void matrix<int>::copy( const matrix &other) ...
```

Trait

```
template <typename T> struct copy_trait
{
    static void copy( T* to, const T* from, int n)  {
        for( int i = 0; i < n; ++i ) to[i] = from[i];
    }
};
template <> struct copy_trait<long>
{
    static void copy( long* to, const long* from, int n) {
        memcpy( to, from, n*sizeof(long));
    }
};
template <class T, class Cpy = copy_trait<T> >
class matrix {  ... }

template <class T, class Cpy>
void matrix<T,Cpy>::copy( const matrix &other) {
    x = other.x;
    y = other.y;
    v = new T[x*y];
    Cpy::copy( v, other.v, x*y);
}
```

Policy

```
template <typename T, bool B> struct copy_trait
{
    static void copy( T* to, const T* from, int n) {
        for( int i = 0; i < n; ++i )
            to[i] = from[i];
    }
};
template <typename T> struct copy_trait<T, true>
{
    static void copy( T* to, const T* from, int n)    {
        memcpy( to, from, n*sizeof(T));
    }
};

template <typename T> struct is_pod {  enum { value = false }; };
template <> struct is_pod<long>     {  enum { value = true }; };

template <class T, class Cpy = copy_trait<T,is_pod<T>::value> >
class matrix  { ... }
```

Typelist

```
class NullType {};

// We now can construct a null-terminated list of typenames:
typedef Typelist< char,
           Typelist<signed char,
             Typelist<unsigned char, NullType>
           >
        > Charlist;

// For the easy maintenance, precompiler macros are defined
// to create Typelists:

#define TYPELIST_1(x)           Typelist< x, NullType>
#define TYPELIST_2(x, y)        Typelist< x, TYPELIST_1(y)>
#define TYPELIST_3(x, y, z)     Typelist< x, TYPELIST_2(y,z)>
#define TYPELIST_4(x, y, z, w)  Typelist< x, TYPELIST_3(y,z,w)>

// usage example
typedef TYPELIST_3(char, signed char, unsigned char)  Charlist;
```

Typelist operations

```
//  Length
template <class TList> struct Length;

template <>
struct Length<NullType>
{
    enum { value = 0 };
};

template <class T, class U>
struct Length <Typelist<T,U> >
{
    enum { value = 1 + Length<U>::value };
};

static const int len = Length<Charlist>::value;
```

Typelist operations

```
// IndexOf
template <class TList, class T> struct IndexOf;

template <class T>
struct IndexOf< NullType, T>
{
    enum { value = -1 };
};
template <class T>
struct IndexOf< Typelist<T, Tail>, T>
{
    enum { value = 0 };
};
template <class T, class Tail>
struct IndexOf< Typelist<Head, Tail>, T>
{
private:
    enum { temp = IndexOf<Tail, T>::value };
public:
    enum { value = (temp == -1 ? -1 : 1+temp) };
};

static const int IndexOf<Charlist, int>::value;           // -1
static const int IndexOf<Charlist, char>::value;          //  0
static const int IndexOf<Charlist, unsigned char>::value; //  2
```

Use case example 2: matrix

```
typedef TYPELIST_4(char, signed char, unsigned char, int)  Pod_types;

template <typename T> struct is_pod
{
    enum { value = IndexOf<Pod_types,T>::value != -1 };
};
template <typename T, bool B> struct copy_trait
{
    static void copy( T* to, const T* from, int n)   {
        for( int i = 0; i < n; ++i )
            to[i] = from[i];
    }
};
template <typename T> struct copy_trait<T, true>
{
    static void copy( T* to, const T* from, int n)   {
        memcpy( to, from, n*sizeof(T));
    }
};
template <class T, class Cpy = copy_trait<T,is_pod<T>::value> >
class matrix { ... }
```

C++ specific patterns

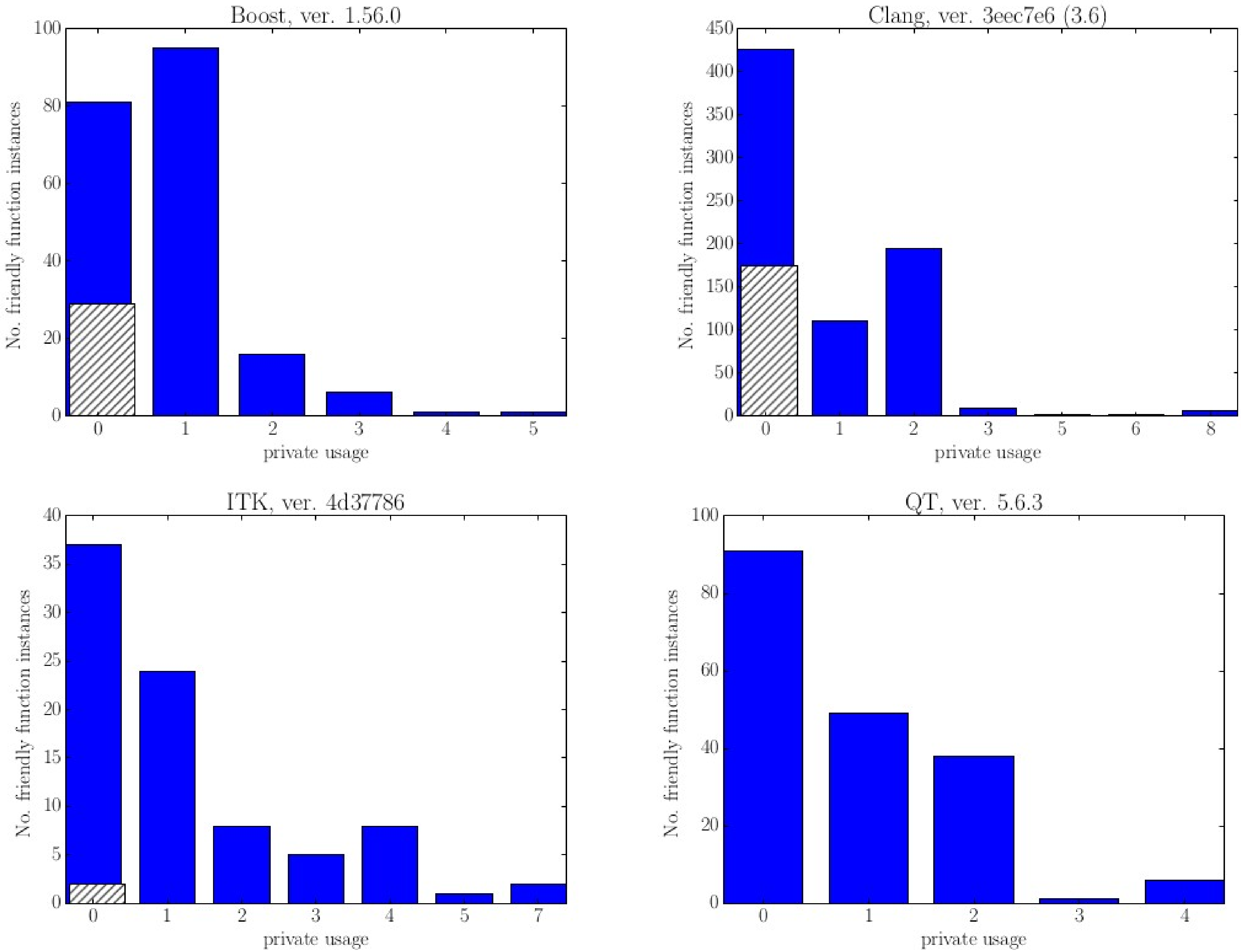
- Attorney – Client pattern
  - Selective friends
  - Friends thru inheritance
- Mixin
  - Liskov substitutional principle
  - C++11 mixins
- Curiously Recurring Template Pattern (CRTP)
  - Operator generation
  - Counting
  - Polymorphic chain
  - Static polymorphism

Attorney-Client

- Friend declaration of C++ gives a complete access to the internals of a class
- No selective access to individual fields or methods
- Control the granularity of access to the implementation details of a class

```
class X                    class FC
{                          {
friend class FC;             // this class has access for all
                             // members of class X
public:                    };
  // ...
private:
  // ...
};
```

Attorney-Client



Attorney-Client

```
class Client
{
private:
  void A(int a);
  void B(float b);
  void C(double c);
  friend class Attorney;
};

class Attorney
{
private:
  static void callA(Client &c, int a) {
    c.A(a);
  }
  static void callB(Client &c, float b) {
    c.B(b);
  }
  friend class Bar;
};

class Bar {
  // Bar now has access to only Client::A and Client::B
  // through the Attorney.
};
```

Attorney-Client

- Attorney works like a Proxy, of Client but implements only part of its interface
- The full implementation of Attorney is private – that prevents unexpected classes to have access to Client
- The Attorney can give acecss to other classes or functions
- Multiple Attorneys can be applied to Client creating different access groups
- Friendship is not inherited in C++, but Attorney-Client can solve this too

Attorney-Client

```
class Base {
private:
  virtual void Func(int x)   { /* Base method */ }
  friend class Attorney;
public:
  virtual ~Base() {}
};
```

```
class Attorney {
private:
  static void callFunc(Base &b, int x) { return b.Func(x); }
  friend int main();
};

int main(void)
{
  Base b;
  Attorney::callFunc(b, 10);
}
```

Attorney-Client

```
class Base {
private:
  virtual void Func(int x)   { /* This is called from main */ }
  friend class Attorney;
public:
  virtual ~Base() {}
};

class Derived : public Base {
private:
  virtual void Func(int x)  { /* This is called from main */ }
public:
  ~Derived() {}
};

class Attorney {
private:
  static void callFunc(Base &b, int x) { return b.Func(x); }
  friend int main();
};

int main(void)
{
  Derived d;
  Attorney::callFunc(d, 10);
}
```

Mixins

- Class inheriting from its own template parameter
- Not to mix with other mixins (e.g. Scala)
- Reversing the inheritance relationship:
  - One can define the Derived class before Base class
  - Policy/Strategy can be injected

```
template <class Base>
class Mixin : public Base  { ... };

class RealBase { ... };
Mixin<RealBase> rbm;

class Strategy { ... };
Mixin<Strategy> mws;
```

Liskov substitutional principle

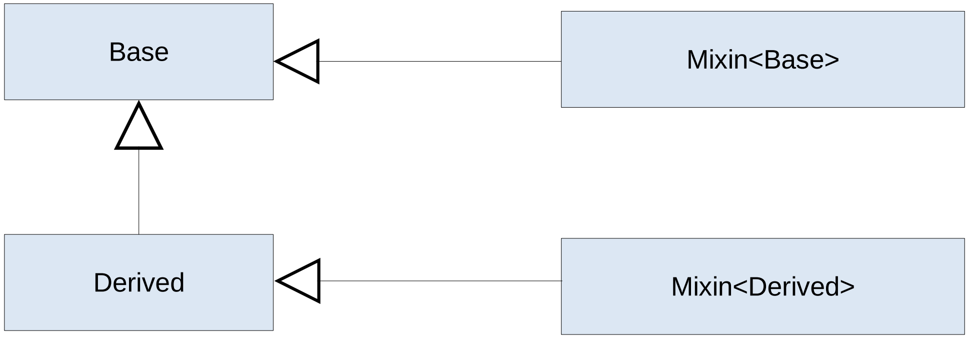
- Barbara Liskov 1977: object of subtype can replace object of base
- Mixin<Derived> is not inherited from Mixin<Base>

```
class Base { ... };
class Derived : public base { ... };

template <class T> class Mixin : public T { ... };

Base        b;
Derived     d;

Mixin<Base>     mb;
Mixin<Derived>  md;

b  = d      // OK
mb = md;    // Error!
```

Liskov substitutional principle

- Barbara Liskov 1977: object of subtype can replace object of base
- Mixin<Derived> is not inherited from Mixin<Base>

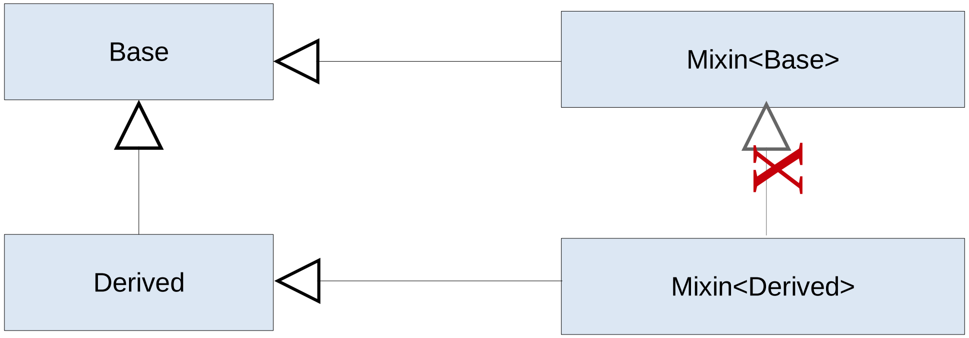
```
class Base { ... };
class Derived : public Base { ... };

template <class T> class Mixin : public T { ... };

Base         b;
Derived      d;

Mixin<Base>     mb;
Mixin<Derived>  md;

b  = d       // OK
mb = md;     // Error!
```



Liskov substitutional principle

- Barbara Liskov 1977: object of subtype can replace object of base
- Mixin<Derived> is not inherited from Mixin<Base>

```
class Base { ... };
class Derived : public Base { ... };

template <class T> class Mixin : public T { ... };

Base        b;
Derived     d;

Mixin<Base>     mb;
Mixin<Derived>  md;

b  = d       // OK
mb = md;     // Error!
```



Constraints

- No concept checking in C++ (yet)

```
class Sortable { void sort(); };

//  be careful with lazy instantiation
template <class Sortable>
class WontWork : public Sortable
{
public:
    void sort()
    {
        Sortable::srot();  // !!misspelled
    }
};
void client()
{
  WontWork<HasSortNotSrot> w;  // still compiles
  w.sort()   // syntax error only here!
}
```

Variadic templates (C++11)

- Type pack defines sequence of type parameters
- Recursive processing of pack

```
template<typename T>
T sum(T v)
{
  return v;
}
template<typename T, typename... Args>  // template parameter pack
T sum(T first, Args... args)            // function parameter pack
{
  return first + sum(args...);
}

int main()
{
  double lsum = sum(1, 2, 3.14, 8L, 7);

  std::string s1 = "x", s2 = "aa", s3 = "bb", s4 = "yy";
  std::string ssum = sum(s1, s2, s3, s4);
}
```

Variadic templates (C++11)

- Type pack defines sequence of type parameters
- Recursive processing of pack

```
template<typename T>
T sum(T v)
{
  return v;
}
template<typename T, typename... Args>  // template parameter pack
std::common_type<T,Args...>::type sum(T first, Args... args)
{
  return first + sum(args...);
}

int main()
{
  double lsum = sum(1, 2, 3.14, 8L, 7);

  std::string s1 = "x", s2 = "aa", s3 = "bb", s4 = "yy";
  std::string ssum = sum(s1, s2, s3, s4);
}
```

Mixin reloded (C++11)

- Variadic templates make us possible to define variadic set of base

```
struct A {};
struct B {};
struct C {};
struct D {};

template<class... Mixins>
class X : public Mixins...
{
public:
    X(const Mixins&... mixins) : Mixins(mixins)... { }
};

int main()
{
  A a; B b; C c; D d;

  X<A,B,C,D> xx(a,b,c,d);
}
```

Curiously Recurring Template Pattern (CRTP)

- James Coplien 1995
- F-bounded polymorphism 1980's
- Operator generation, Enable_shared_from_this, Static polymorphism

```
template <typename T>
struct Base
{
    // ...
};

struct Derived : Base<Derived>
{
    // ...
};
```

Operator generator

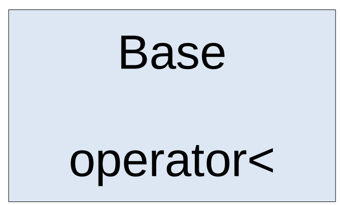
```
class C
{
  // ...
};

bool operator<(const C& l, const C& r)
{
  // ...
};

inline bool operator!=(const C& l, const C& r) { return l<r || r<l; }
inline bool operator==(const C& l, const C& r) { return ! (l!=r); }
inline bool operator>=(const C& l, const C& r) { return r < l; }
// ...
```

Operator generator

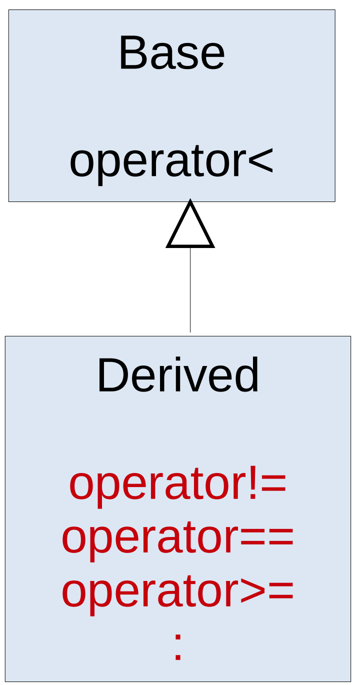
```
class C
{
  // ...
};

bool operator<(const C& l, const C& r)
{
  // ...
};

inline bool operator!=(const C& l, const C& r) { return l<r || r<l; }
inline bool operator==(const C& l, const C& r) { return ! (l!=r); }
inline bool operator>=(const C& l, const C& r) { return r < l; }
// ...
```

- We want automatically generate the operators from operator<

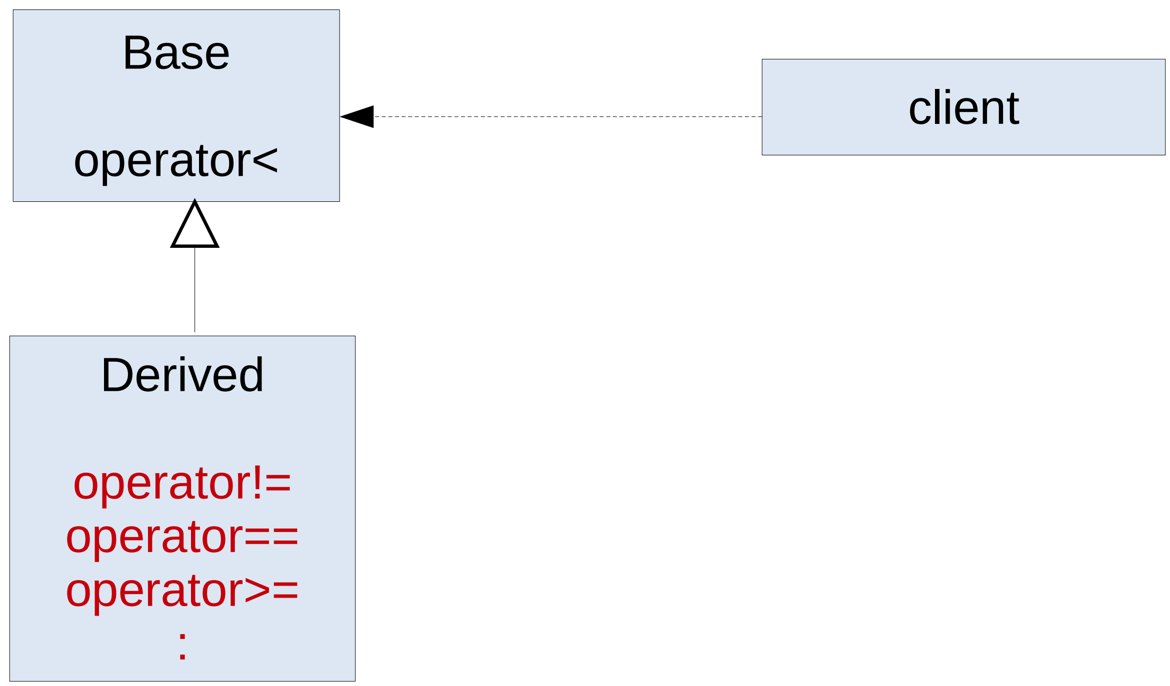
Operator generator in Derived



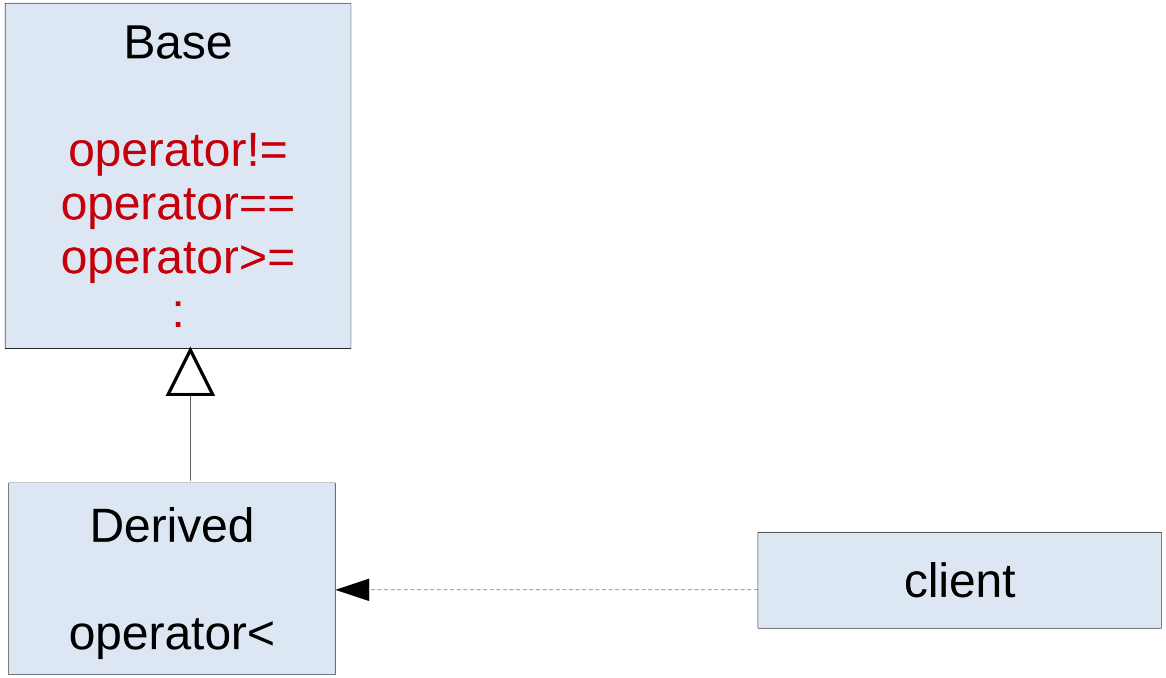
Operator generator in Derived



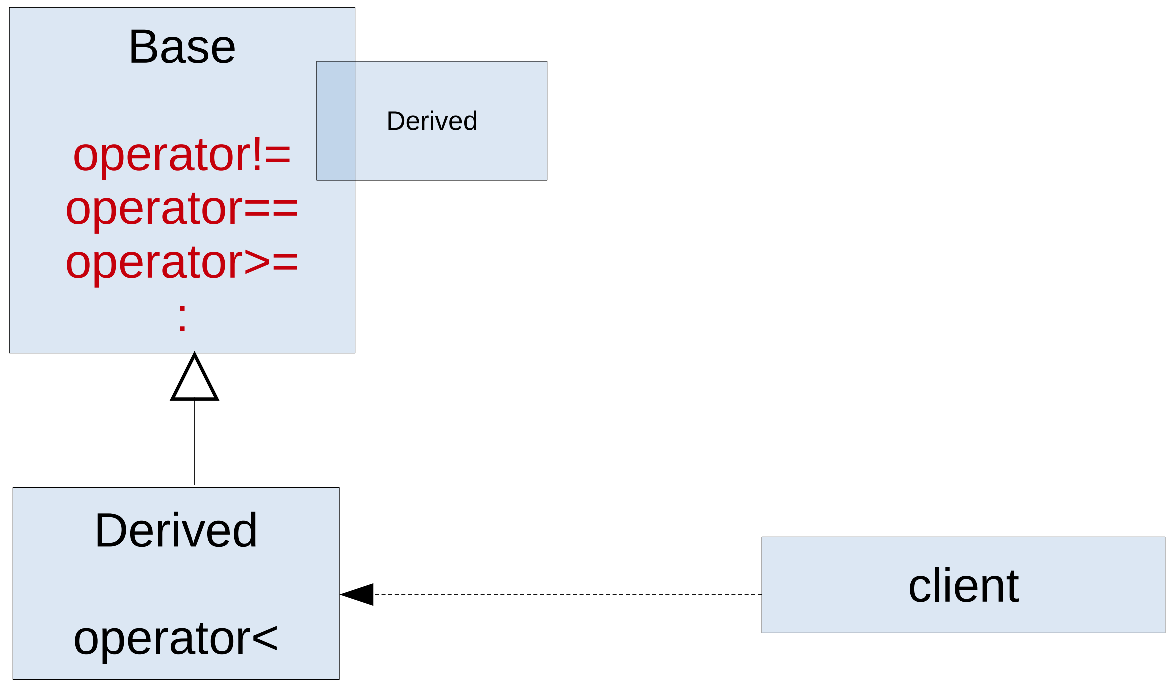
Operator generator in Derived



Operator generator in Base ~~Derived~~



Operator generator in Derived



Enable shared from this

```
#include <memory>
#include <cassert>

class Y : public std::enable_shared_from_this<Y>
{
public:

    std::shared_ptr<Y> f()
    {
        return shared_from_this();
    }
};

int main()
{
    std::shared_ptr<Y> p(new Y);
    std::shared_ptr<Y> q = p->f();
    assert(p == q);
    assert(!(p < q || q < p)); // p and q must share ownership
}
```

Object counter

```
template <typename T>
struct counter {
    static int objects_created;
    static int objects_alive;
    counter() {
        ++objects_created;
        ++objects_alive;
    }
    counter(const counter&) {
        ++objects_created;
        ++objects_alive;
    }
protected:
    ~counter() // objects should never be removed through pointers of this type
    {
        --objects_alive;
    }
};
template <typename T> int counter<T>::objects_created( 0 );
template <typename T> int counter<T>::objects_alive( 0 );

class X : counter<X> { ... };
class Y : counter<Y> { ... };
```

Polymorphic chaining

```
class Printer
{
public:
  Printer(ostream& pstream) : m_stream(pstream) {}

  template <typename T>
  Printer& print(T&& t) { m_stream << t; return *this; }

  template <typename T>
  Printer& println(T&& t) { m_stream << t << endl; return *this; }
private:
  ostream& m_stream;
};








void f()
{
  Printer{myStream}.println("hello").println(500);  // works!


}
```

Polymorphic chaining

```
class Printer
{
public:
  Printer(ostream& pstream) : m_stream(pstream) {}

  template <typename T>
  Printer& print(T&& t) { m_stream << t; return *this; }

  template <typename T>
  Printer& println(T&& t) { m_stream << t << endl; return *this; }
private:
  ostream& m_stream;
};
class ColorPrinter : public Printer
{
public:
  ColorPrinter() : Printer(cout) {}
  ColorPrinter& SetConsoleColor(Color c) { /* ... */ return *this; }
};
void f()
{
  Printer{myStream}.println("hello").println(500);  // works!


}
```

Polymorphic chaining

```
class Printer
{
public:
  Printer(ostream& pstream) : m_stream(pstream) {}

  template <typename T>
  Printer& print(T&& t) { m_stream << t; return *this; }

  template <typename T>
  Printer& println(T&& t) { m_stream << t << endl; return *this; }
private:
  ostream& m_stream;
};
class ColorPrinter : public Printer
{
public:
  ColorPrinter() : Printer(cout) {}
  ColorPrinter& SetConsoleColor(Color c) { /* ... */ return *this; }
};
void f()
{
  Printer{myStream}.println("hello").println(500);  // works!
  // compile error             v here we have Printer, not ColorPrinter
  ColorPrinter().print("Hello").SetConsoleColor(Color.red).println("Printer!");
}
```

Polymorphic chaining

```
template <typename ConcretePrinter>
class Printer
{
public:
  Printer(ostream& pstream) : m_stream(pstream) {}

  template <typename T>
  ConcretePrinter& print(T&& t) { m_stream << t;
                                return static_cast<ConcretePrinter&>(*this); }
  template <typename T>
  ConcretePrinter& println(T&& t) { m_stream << t << endl;
                                return static_cast<ConcretePrinter&>(*this); }
 private:
  ostream& m_stream;
};
class ColorPrinter : public Printer<ColorPrinter>
{
public:
  ColorPrinter() : Printer(cout) {}
  ColorPrinter& SetConsoleColor(Color c) { /* ... */ return *this; }
};
void f()
{
  Printer{myStream}.println("hello").println(500);  // works!
  ColorPrinter().print("Hello").SetConsoleColor(Color.red).println("Printer!");
}
```

Static polymorphism

- When we separate interface and implementation
- But no run-time variation between objects

```
template <class Derived>
struct Base
{
  void interface()
  {
    // ...
    static_cast<Derived*>(this)->implementation();
    // ...
  }
  static void static_funcion()
  {
    Derived::static_sub_funcion();
  }
};
struct Derived : Base<Derived>
{
  void implementation();
  static void static_sub_function();
};
```