

Introduction to programming with templates

Procedural
(C programming)

C++

Generic
programming

Object Oriented
Programming

Refresher of basic concepts
Policy based class design
Suggestions for further learning

Maria Grazia Pia

INFN Genova, Italy

Maria.Grazia.Pia@cern.ch

<http://www.ge.infn.it/geant4/training/APC2017/>

What is what

Generic programming

Finding the most abstract expression of a computation without losing its essence and efficiency (*A. Alexandrescu*)

Generalizing software components so that they can be easily reused in a wide variety of situations (*Boost*)

Sometimes defined as ***programming with templates***

Object-oriented programming is thought of as *programming with virtual functions*

Generative programming

Writing code that generates code

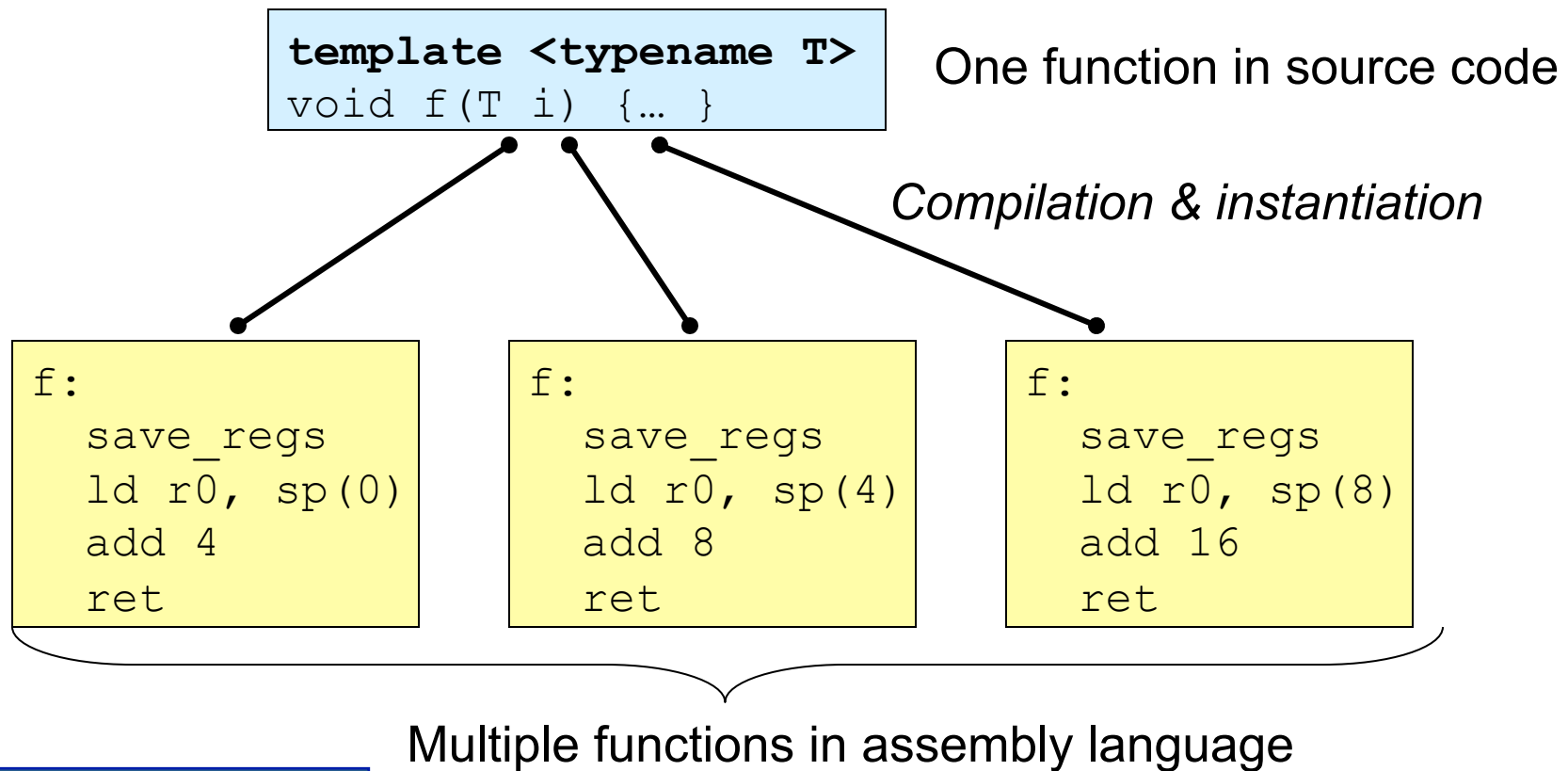
C++ templates are intrinsically generative

Template meta-programming

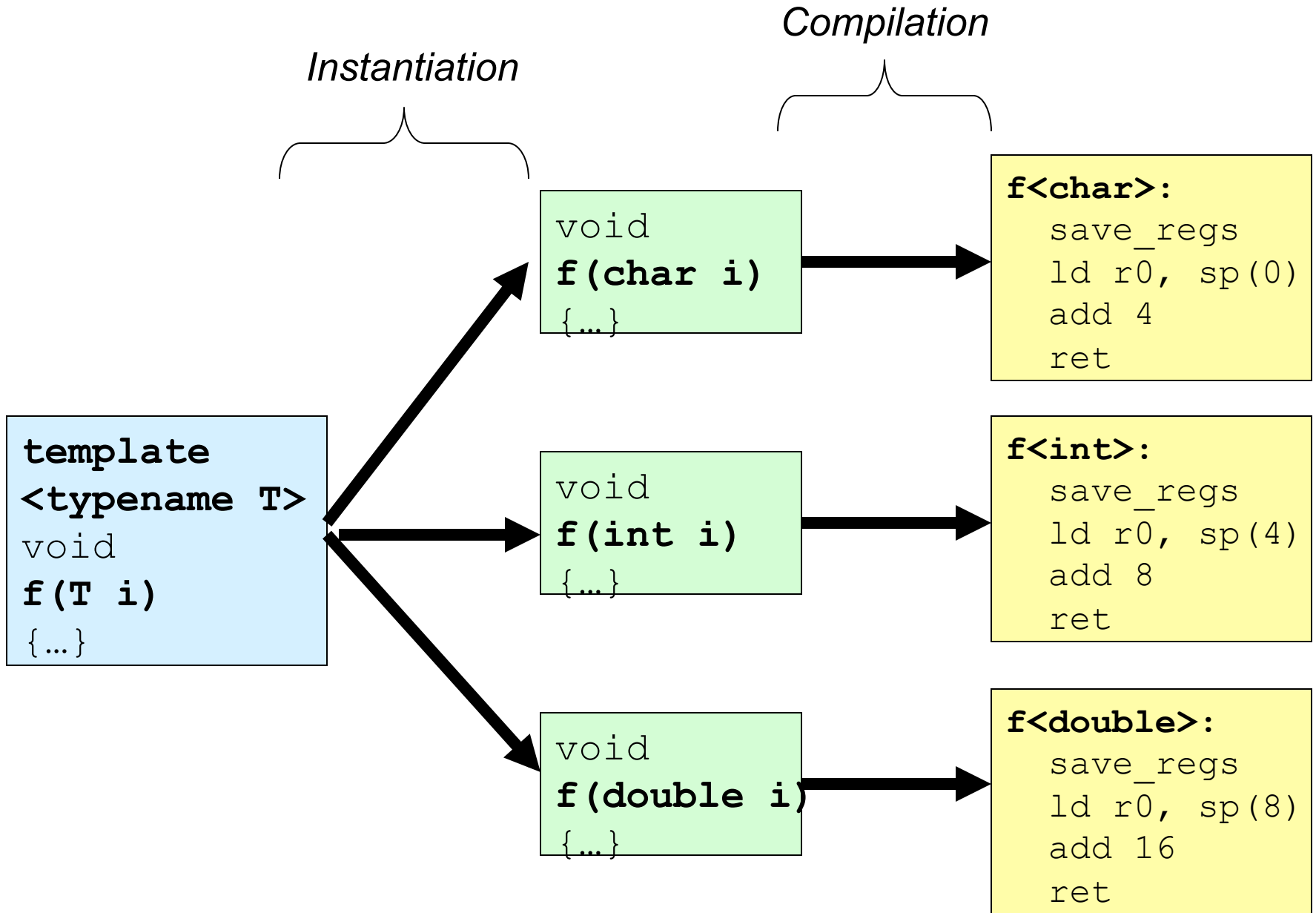
A metaprogram is a program that generates or manipulates program code

Templates (*refresher*)

- A C++ template is just that, a template
- A single template serves as a pattern, so it can be used multiple times to create multiple instantiations



- Function templates
- Class templates



Multiple functions in assembly language

Function templates

- Provide a functional behavior that can be called for different types
- The representation looks like an ordinary function, but some elements of the function are left undetermined (parameterized)

```
template <typename T> also: template <class T>
inline T const& max (T const& a, T const& b)
{
    // if a < b then use b else use a
    return a<b?b:a;
}
```

defining the template

```
#include <iostream>
#include "max.hh"

int main()
{
    int i = 33;
    std::cout << "max(7,i): " << ::max(7,i) << std::endl;

    double f1 = 2.4;
    double f2 = 7.4;
    std::cout << "max(f1,f2): " << ::max(f1,f2) << std::endl;
}
```

using the template

Template instantiation

The process of replacing template parameters by concrete types is called **instantiation**

It results in an **instance** of a template

`max(7, i)` has the semantics of calling the code:

```
inline int const& max (int const& a, int const& b)
{
    // if a < b then use b else use a
    return a<b?b:a;
}
```

An attempt to instantiate a template for a type that does not support all the operations used within it will result in a compile-time error

```
std::complex<float> c1, c2; // doesn't provide operator <
```

...

```
max(c1,c2);
```

ERROR at compile time

Template parameters

Function templates have two kinds of parameters:

Template parameters

are declared in angle brackets before the function template name

```
template <typename T> // T is template parameter
```

Call parameters

are declared in parentheses after the function template name

```
... max (T const& a, T const& b) // a and b are call parameters
```

One may have as many template parameters as one likes

Overloading

Function templates can be **overloaded** like ordinary functions

```
// maximum of two values of any type
```

```
template <typename T>
```

```
inline T const& max (T const& a, T const& b)
```

```
{
```

```
    return a<b?b:a;
```

```
}
```

T const&

a reference to a const T

```
// maximum of three values of any type
```

```
template <typename T>
```

```
inline T const& max (T const& a, T const& b, T const& c)
```

```
{
```

```
    return max (max(a,b), c);
```

```
}
```


Class templates

```
template <typename T>  
class Stack {  
    ...  
};
```

Declaration of class template

```
template <typename T>  
class Stack {  
    private:  
        std::vector<T> elems;    // elements  
  
    public:  
        Stack();                // constructor  
        void push(T const&);    // push element  
        void pop();             // pop element  
        T top() const;          // return top element  
};
```

For class templates, member functions are instantiated only when they are used

Specializations of Class Templates

One can specialize a class template
for certain template argument

```
template<>
class Stack<std::string> { Stack is specialized for string
    ...
};
```

For specializations, any member function must be defined as an "ordinary" member function, with each occurrence of T being replaced by the specialized type:

```
void Stack<std::string>::push (std::string const& elem)
{
    elems.push_back(elem); // append copy of passed elem
}
```

Partial specialization

```
template <typename T1, typename T2>
class MyClass {
    ...
};
```

// both template parameters have same type

```
template <typename T>
class MyClass<T,T> {
    ...
};
```

// partial specialization: second type is int

```
template <typename T>
class MyClass<T,int> {
    ...
};
```

// both template parameters are pointer types

```
template <typename T1, typename T2>
class MyClass<T1*,T2*> {
    ...
};
```

One can specify
**special
implementations**
for particular
circumstances

```
MyClass<int,float> mif;
```

uses MyClass<T1,T2>

```
MyClass<float,float> mff;
```

uses MyClass<T,T>

```
MyClass<float,int> mfi;
```

uses MyClass<T,int>

```
MyClass<int*,float*> mp;
```

uses MyClass<T1*,T2*>

Default template arguments

One can define default values for template parameters

```
#include <vector>

template <typename T, typename CONT = std::vector<T> >
class Stack {
private:
    CONT elems; // elements

public:
    void push(T const&); // push element
    void pop(); // pop element
    T top() const; // return top element
    bool empty() const { // return whether the stack is empty
        return elems.empty();
    }
};
```

`std::vector<>` is the **default** value for the container that is used to manage the elements

Using templates in practice

Classes and other types



Header file

Non-template code

(non-inline) Functions



- Declaration in a header file
- Definition goes into .cc file

```
#ifndef MYFIRST_HH
#define MYFIRST_HH
#include <iostream>
#include <typeinfo>

// declaration of template
template <typename T>
void print_typeof (T const&);

// implementation/definition of template
template <typename T>
void print_typeof (T const& x)
{
    std::cout << typeid(x).name() << std::endl;
}
#endif // MYFIRST_HH
```

Inclusion model
Declaration and definition in header file

When the compiler sees a template definition, it does not generate code
The compiler produces type-specific instances of the template only when it sees a use of the template

To generate an instantiation, the compiler must have access to the source code that defines the template



Code bloat
Increase compile burden

Debugging

```
std::list<std::string> coll;  
...  
// Find the first element greater than "A"  
std::list<std::string>::iterator pos;  
pos = std::find_if(coll.begin(), coll.end(), // range  
    std::bind2nd(std::greater<int>(), "A")); // criterion
```

```
/local/include/stl/_algo.h: In function 'struct _STL::_List_iterator<_STL::basic_string<char, _STL::char_traits<char>, _STL::allocator<char>>, _STL::Nonconst_traits<_STL::basic_string<char, _STL::char_traits<char>, _STL::allocator<char>>>>_STL::find_if<_STL::_List_iterator<_STL::basic_string<char, _STL::char_traits<char>, _STL::allocator<char>>, _STL::Nonconst_traits<_STL::basic_string<char, _STL::char_traits<char>, _STL::allocator<char>>>>, _STL::binder2nd<_STL::greater<int>>>(_STL::_List_iterator<_STL::basic_string<char, _STL::char_traits<char>, _STL::allocator<char>>, _STL::Nonconst_traits<_STL::basic_string<char, _STL::char_traits<char>, _STL::allocator<char>>>>, _STL::_List_iterator<_STL::basic_string<char, _STL::char_traits<char>, _STL::allocator<char>>, _STL::Nonconst_traits<_STL::basic_string<char, _STL::char_traits<char>, _STL::allocator<char>>>>, _STL::binder2nd<_STL::greater<int>>, _STL::input_iterator_tag)':  
/local/include/stl/_algo.h:115: instantiated from '_STL::find_if<_STL::_List_iterator<_STL::basic_string<char, _STL::char_traits<char>, _STL::allocator<char>>, _STL::Nonconst_traits<_STL::basic_string<char, _STL::char_traits<char>, _STL::allocator<char>>>>, _STL::binder2nd<_STL::greater<int>>>(_STL::_List_iterator<_STL::basic_string<char, _STL::char_traits<char>, _STL::allocator<char>>, _STL::Nonconst_traits<_STL::basic_string<char, _STL::char_traits<char>, _STL::allocator<char>>>>, _STL::_List_iterator<_STL::basic_string<char, _STL::char_traits<char>, _STL::allocator<char>>, _STL::Nonconst_traits<_STL::basic_string<char, _STL::char_traits<char>, _STL::allocator<char>>>>, _STL::binder2nd<_STL::greater<int>>>)'  
testprog.cpp:18: instantiated from here  
/local/include/stl/_algo.h:78: no match for call to '(_STL::binder2nd<_STL::greater<int>>) (_STL::basic_string<char, _STL::char_traits<char>, _STL::allocator<char>>&)'  
/local/include/stl/_function.h:261: candidates are: bool _STL::binder2nd<_STL::greater<int>>::operator()(const int&) const
```

 Error message generated by gcc

Generic programming

● Powerful paradigm

- “Bringing aspects of generic programming into the mainstream is most likely C++’s greatest contribution to the software development community during this period.”
B. Stroustrup, Proc. 3rd ACM SIGPLAN Conf. on History of programming languages, 2007

● STL

- The most significant example of generic programming

● Less commonly used than OOP?

● Drawbacks

- code bloat
- compiler support
- poor error messages, cumbersome debugging

Standard Template Library (STL)

Containers

● Sequence

- **vector**: array in contiguous memory
- **list**: doubly-linked list (fast insert/delete)
- **deque**: double-ended queue
- stack, queue, priority queue

● Associative

- **map**: collection of (key,value) pairs
- **set**: map with values ignored
- multimap, multiset (duplicate keys)

● Other

- **string**, basic_string
- **valarray**: for numeric computation
- bitset: set of N bits

Algorithms

● Non-modifying

- find, search, mismatch, count, for_each

● Modifying

- copy, transform/apply, replace, remove

● Others

- unique, reverse, random_shuffle
- sort, merge, partition
- set_union, set_intersection, set_difference
- min, max, min_element, max_element
- next_permutation, prev_permutation

std::vector (more)

Example:

```
#include <vector>
#include <algorithm>

void FunctionExample()
{
    std::vector<int> v(10);
    v[5] = 3; // set fifth element to 3
    std::vector<int>::const_iterator it
        = std::find(v.begin(), v.end(), 3);
    bool found = it != v.end();
    if (found) {
        int three = *it;
    }
}
```

std::vector

Example:

use std::vector,
rather than built-in C-style array,
whenever possible

```
#include <vector>
void FunctionExample()
{
    std::vector<int> v(10);
    int a0 = v[3];           // unchecked access
    int a1 = v.at(3);       // checked access
    v.push_back(2);         // append element to end
    v.pop_back();           // remove last element
    size_t howbig = v.size(); // get # of elements
    v.insert(v.begin()+5, 2); // insert 2 after 5th element
}
```

Polymorphism

- The ability to associate different specific behaviours with a single generic notation
- A cornerstone of the OOP paradigm
 - In C++ supported through **class inheritance** and **virtual functions**
 - Handled at run time: **dynamic polymorphism**
- Templates also allow us to associate different specific behaviours with a single generic notation
 - Handled at compile time: **static polymorphism**
 - **Unbounded**: the interfaces of the types participating in the polymorphic behaviour are not predetermined

Pro and contra

Dynamic polymorphism

The executable code size is potentially **smaller**

Only one polymorphic function is needed, whereas distinct template instances must be generated to handle different types

Static polymorphism

The generated code is potentially **faster**

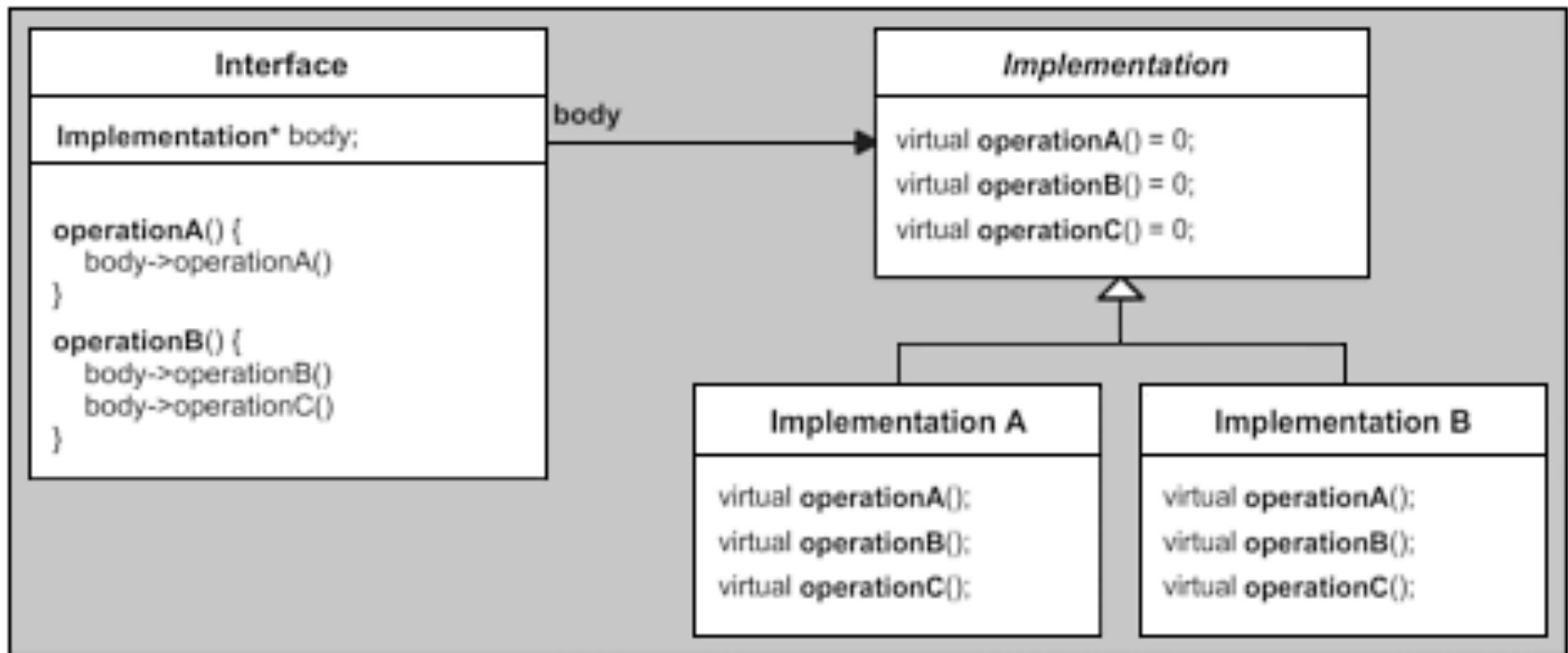
No indirection through pointers is needed a priori and non-virtual functions can be inlined much more often

Dynamic and static polymorphism
can be combined in the same software design



Bridge design pattern

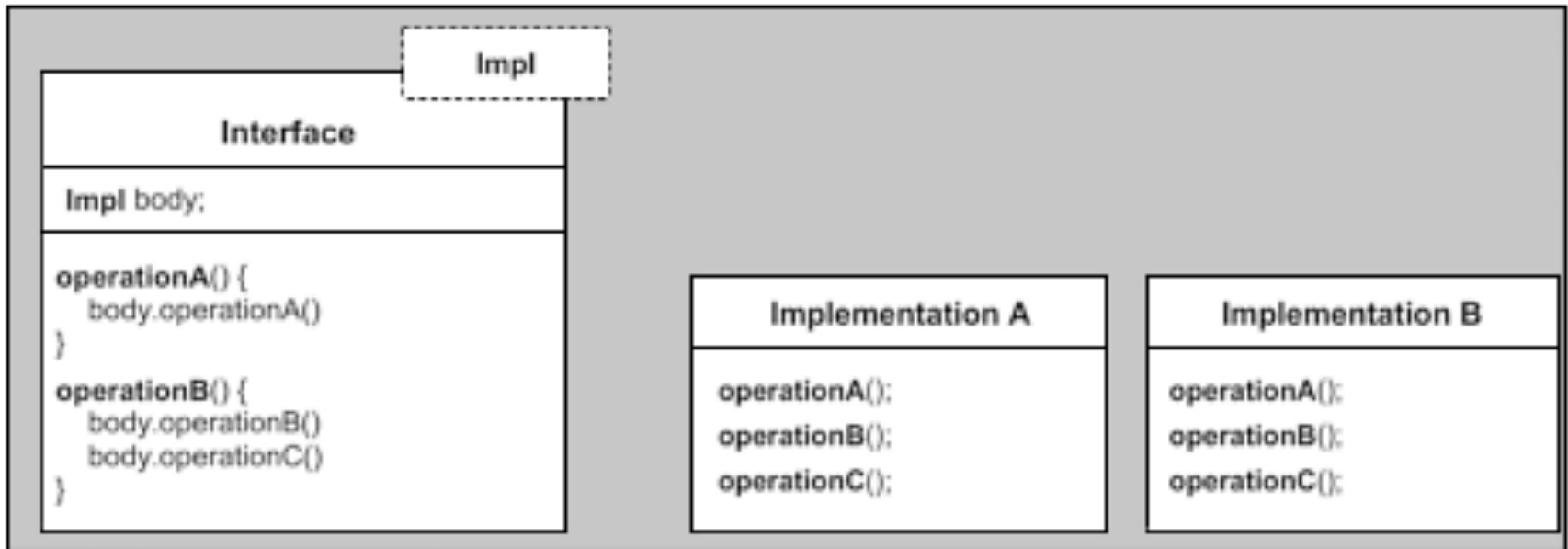
Decouple an abstraction from its implementation so that the two can vary independently



According to Design Patterns, this is achieved by using a pointer to refer to the actual implementation and delegating all calls to this class

Bridge using templates

If the type of the implementation is known at compile time, one could use the approach via templates



This software design avoids pointers and should be faster

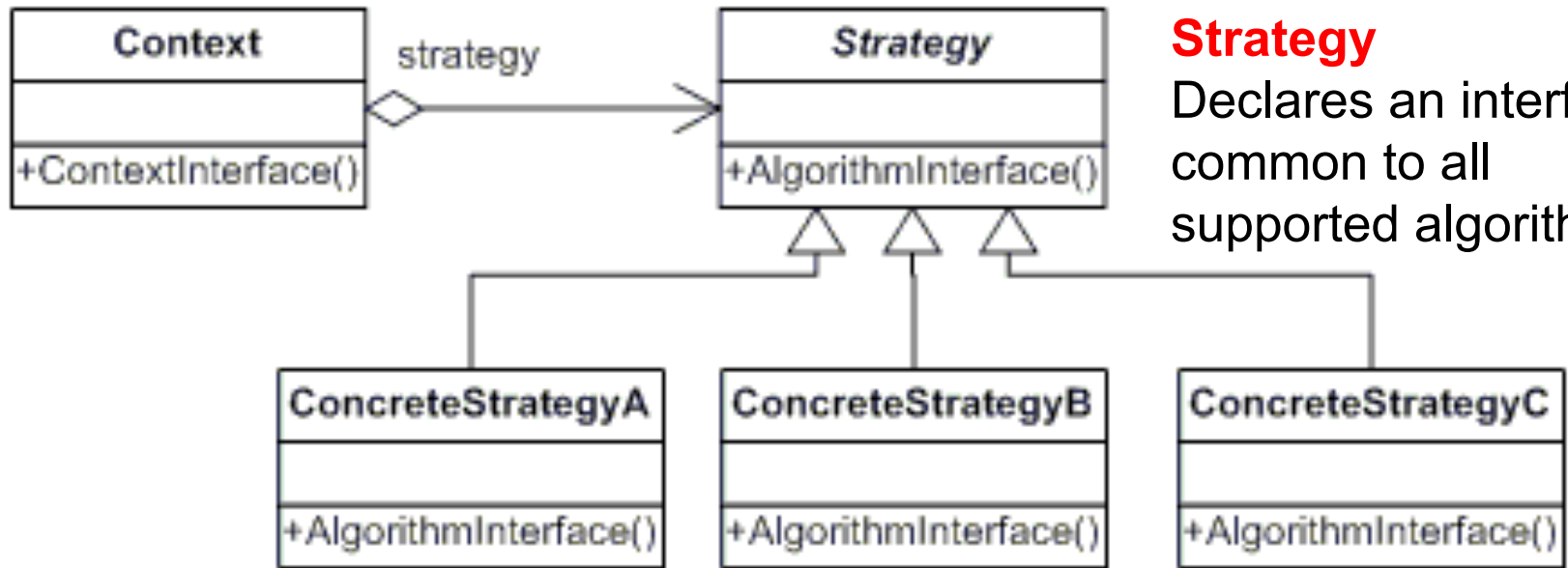
Strategy design pattern

Define a **family of algorithms**, **encapsulate** each one,
 and make them **interchangeable**

Strategy lets the algorithm vary independently from clients that use it

Context

is configured with a ConcreteStrategy object

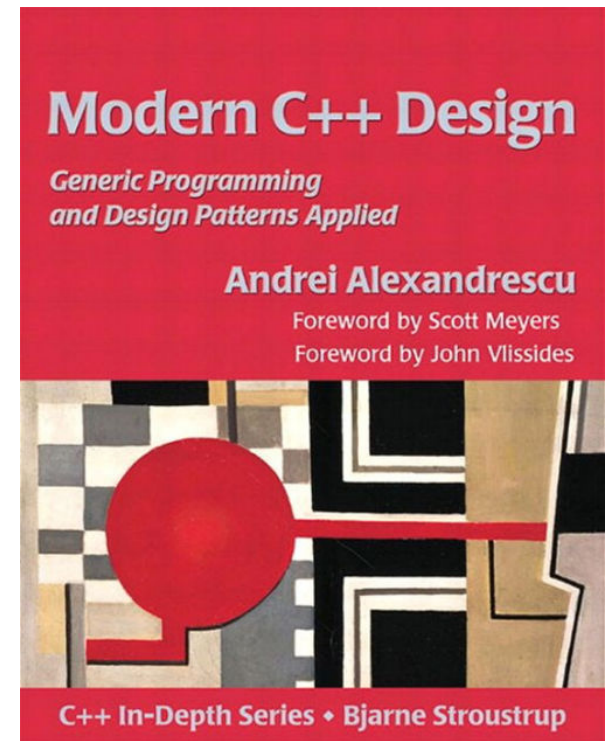


Strategy

Declares an interface
 common to all
 supported algorithms

ConcreteStrategy

implements the algorithm using the Strategy interface



Example of a design technique

POLICY BASED CLASS DESIGN

Policy-based design

- **Policy-based class design** fosters assembling a class with complex behaviour (**host**) out of many little classes (**policies**), each of which takes care of a single behavioural or structural aspect
- A **policy** defines a **class** or **class template interface**
 - Policy classes are not intended for stand-alone use
They are **inherited** by, or **contained** within, other classes
 - Policy classes capture and encapsulate reusable behaviour
- **Policy host** classes are parameterised classes
- Policies are reminiscent of the Strategy pattern
 - Policies are **compile-time bound**
 - Policies are not required to inherit from a base class
 - No need of virtual methods, faster execution

Example: policy to create objects

The Creator policy prescribes a **class template of type T**
This class template must expose a **member function *Create***,
which takes no arguments and returns a pointer to T

```
template <class T>
struct OpNewCreator
{
    static T* Create()
    {
        return new T;
    }
};
```

Loosely defined interface

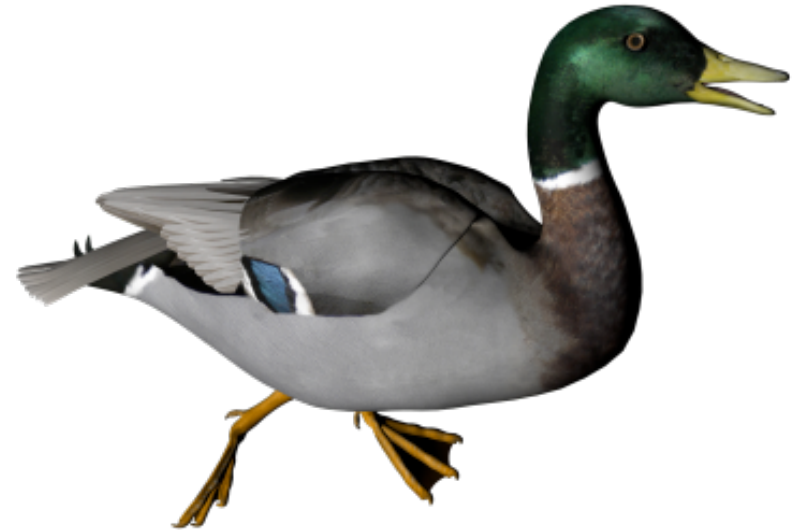
**Multiple
implementations**

```
template <class T>
struct MallocCreator
{
    static T* Create()
    {
        void* buf = std::malloc(sizeof(T));
        if (!buf) return 0;
        return new(buf) T;
    }
};
```

```
template <class T>
struct PrototypeCreator
{
    PrototypeCreator(T* pObj = 0)
        :pPrototype_(pObj)
    {}
    T* Create()
    {
        return pPrototype_ ? pPrototype_->Clone() : 0;
    }
    T* GetPrototype() { return pPrototype_; }
    void SetPrototype(T* pObj) { pPrototype_ = pObj; }
private:
    T* pPrototype_;
};
```

Duck typing

If it looks like a duck,
walks like a duck
and quacks like a duck,
then it is a duck



The **policy interface** does not have a direct,
explicit representation in code
It is **defined implicitly** via duck typing

It must be documented separately and manually
(*e.g. in comments or software documentation*)

Example: host class

Library code

```
template <class CreationPolicy>
class WidgetManager : public CreationPolicy
{
    ...
};
```

Application code

```
typedef WidgetManager< OpNewCreator<Widget> > MyWidgetMgr;
```

WidgetManager allows the user
to **configure a specific aspect of its functionality**
It acts as a code generation engine

Implementing Policy Classes with Template Template Parameters

Library code

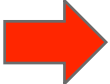
```
template <template <class Created> class CreationPolicy>  
class WidgetManager : public CreationPolicy<Widget>  
{  
    ...  
};
```

*formal argument for CreationPolicy
(not WidgetManager)*



Application code


```
typedef WidgetManager<OpNewCreator> MyWidgetMgr;
```

The host can
instantiate it
with a 
different type

Library code

```
template <template <class> class CreationPolicy>  
class WidgetManager : public CreationPolicy<Widget>  
{  
    ...  
    void DoSomething()  
    {  
        Gadget* pW = CreationPolicy<Gadget>().Create();  
        ...  
    }  
};
```

*WidgetManager creates objects of type Gadget
using the same creation policy*



Implementing Policy Classes with Template Member Functions

- Use template member functions in conjunction with simple classes
- The policy implementation is a simple class
 - as opposed to a template class
- which has one or more templated members

```
struct OpNewCreator
{
    template <class T>
    static T* Create()
    {
        return new T;
    }
};
```

Enriched policies

Creator policy

prescribes only one
member function

Create

PrototypeCreator

Defines two more
member functions

GetPrototype

SetPrototype

WidgetManager inherits its policy class



The two additional public functions propagate through
WidgetManager and are directly accessible to clients

*WidgetManager asks only for the Create member function
Users, however, can exploit the enriched interface*

Design features

Rule of thumb

Anything that can be done in more than one way should be migrated from the class and isolated as a policy

The hardest task of creating policy-based class design is to correctly **decompose** the functionality of a class

Orthogonality

Well designed policies encode an action that is largely **orthogonal** with respect to any other



No dependencies
across policies

Is it worth the effort?

- Programming with templates is cumbersome
- Error messages are often difficult to interpret

Benefits

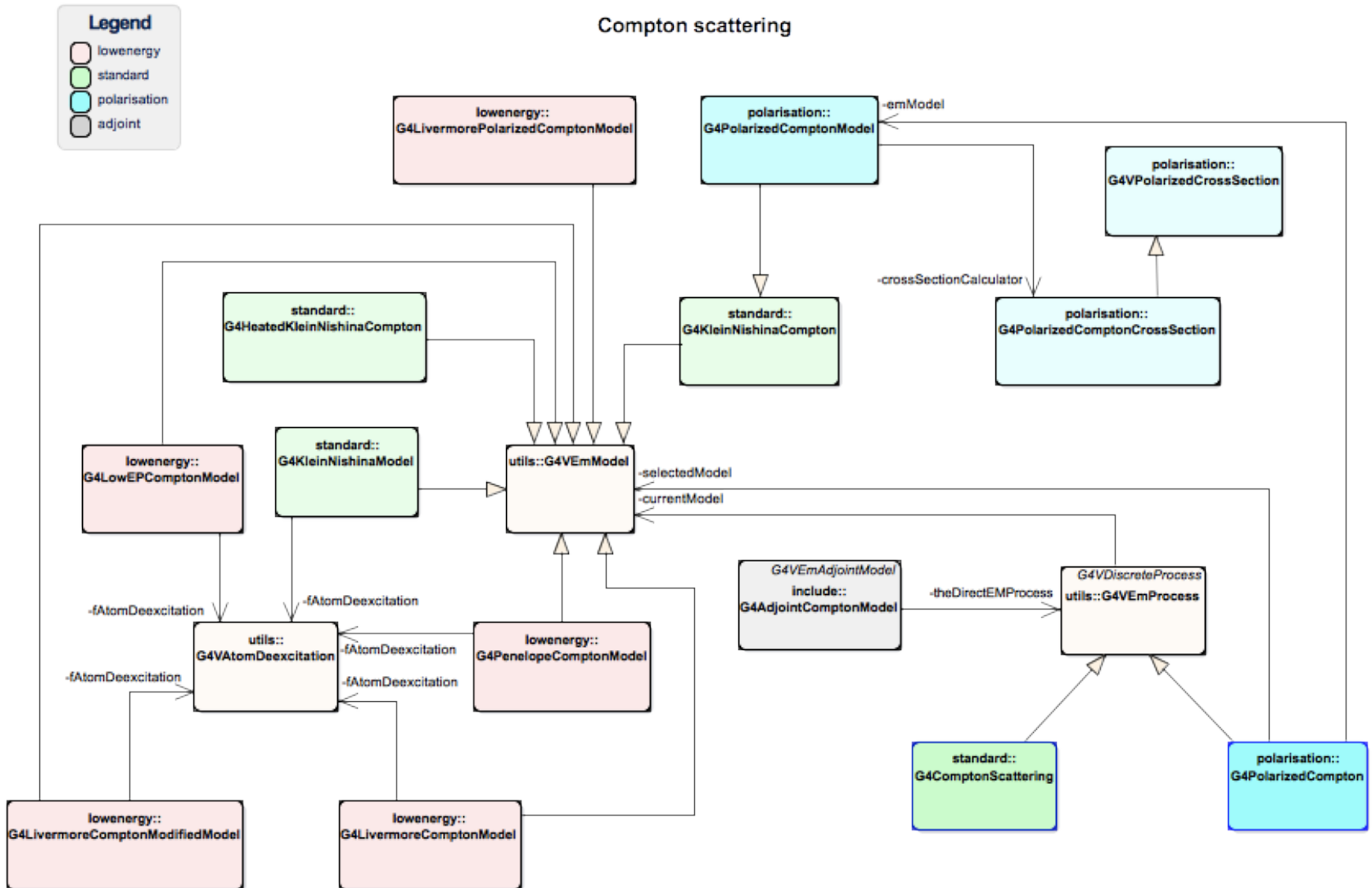
of policy-based
design

Computational performance

Testing

- Unit testing of policy classes is easy
 - Test small classes, with well defined responsibilities
- Unit testing in general is greatly simplified
 - Ability to workaround complexities in testing
 - ▷ e.g. swap in mock policies

Compton scattering

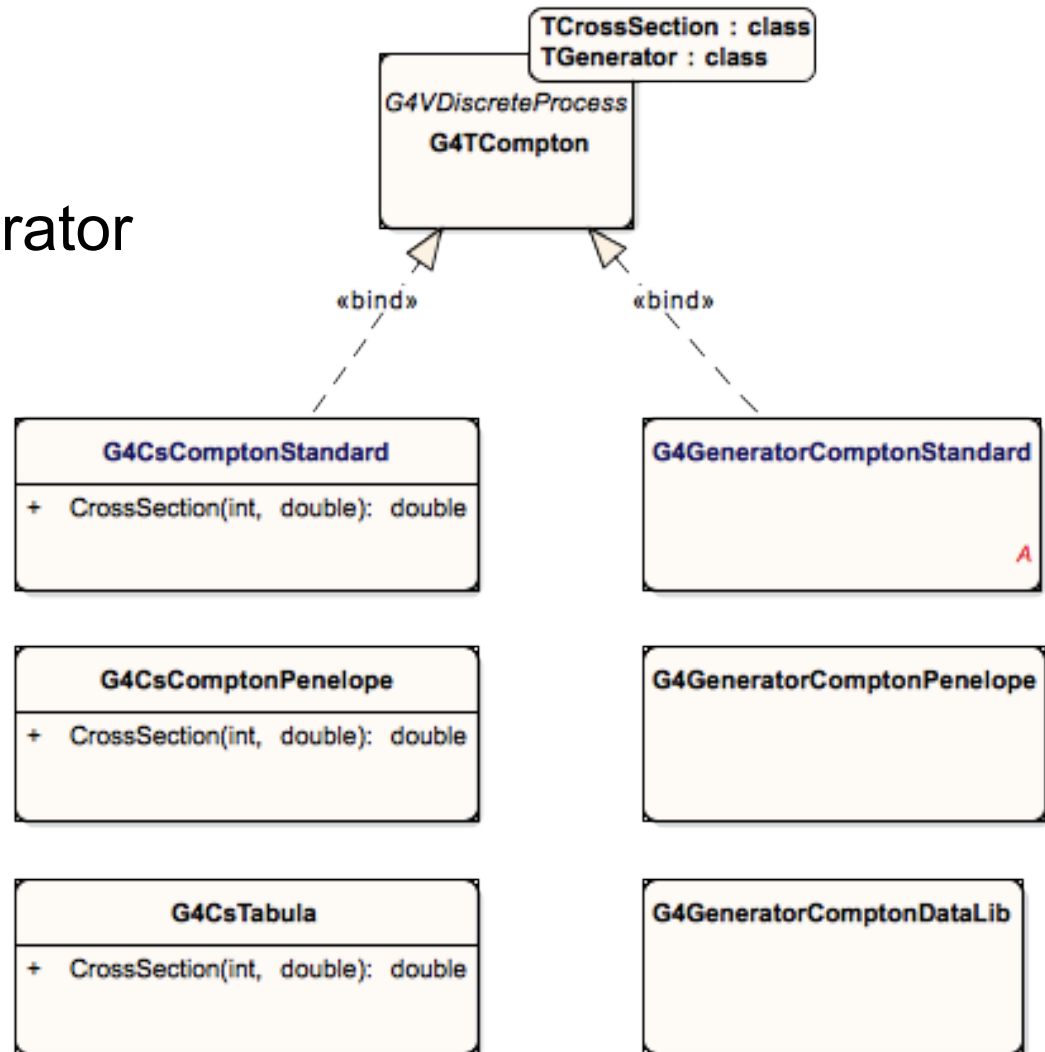


Compton scattering

Two policies:

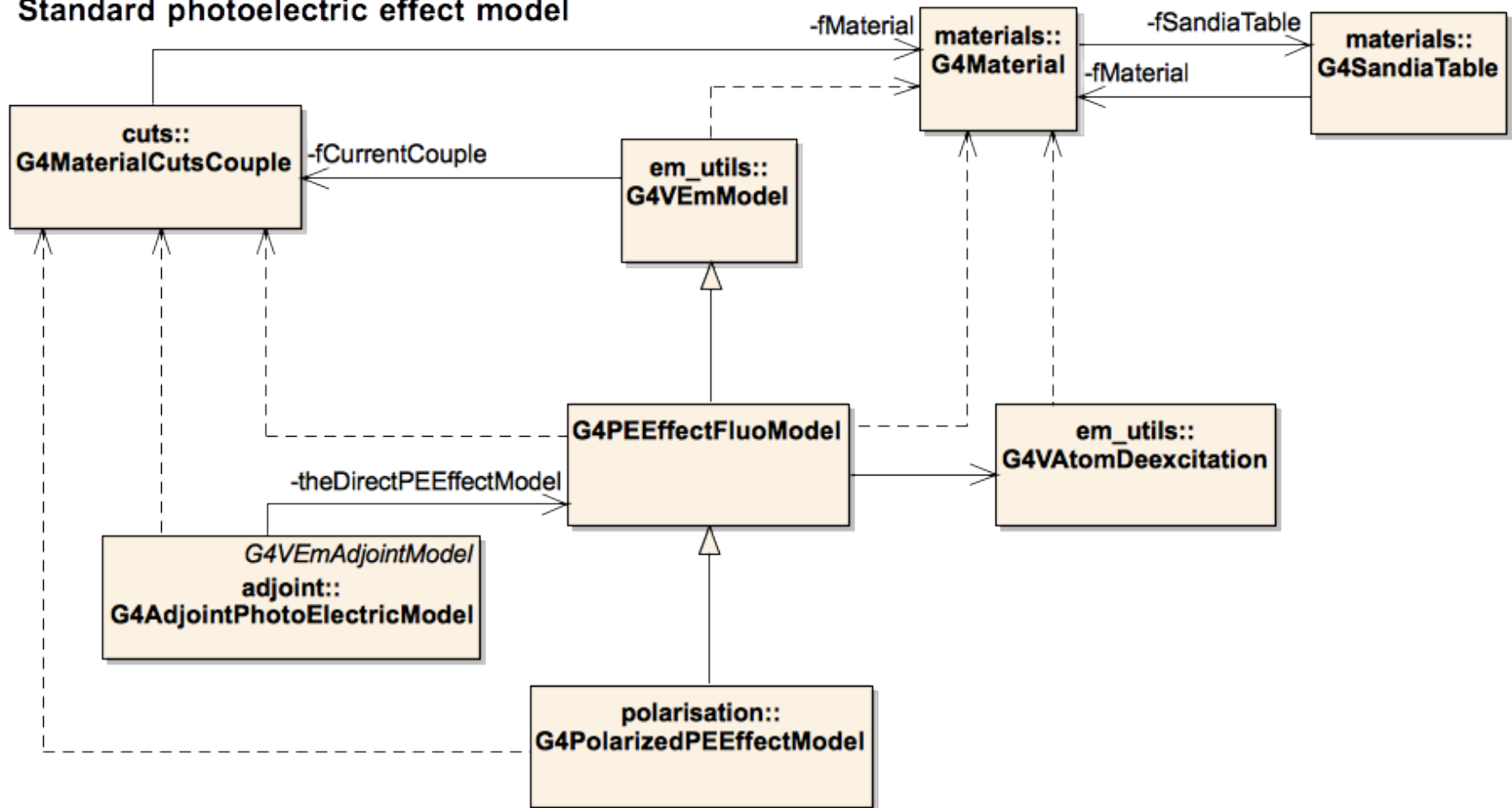
1. Cross section
2. Final state generator

Unit testing!



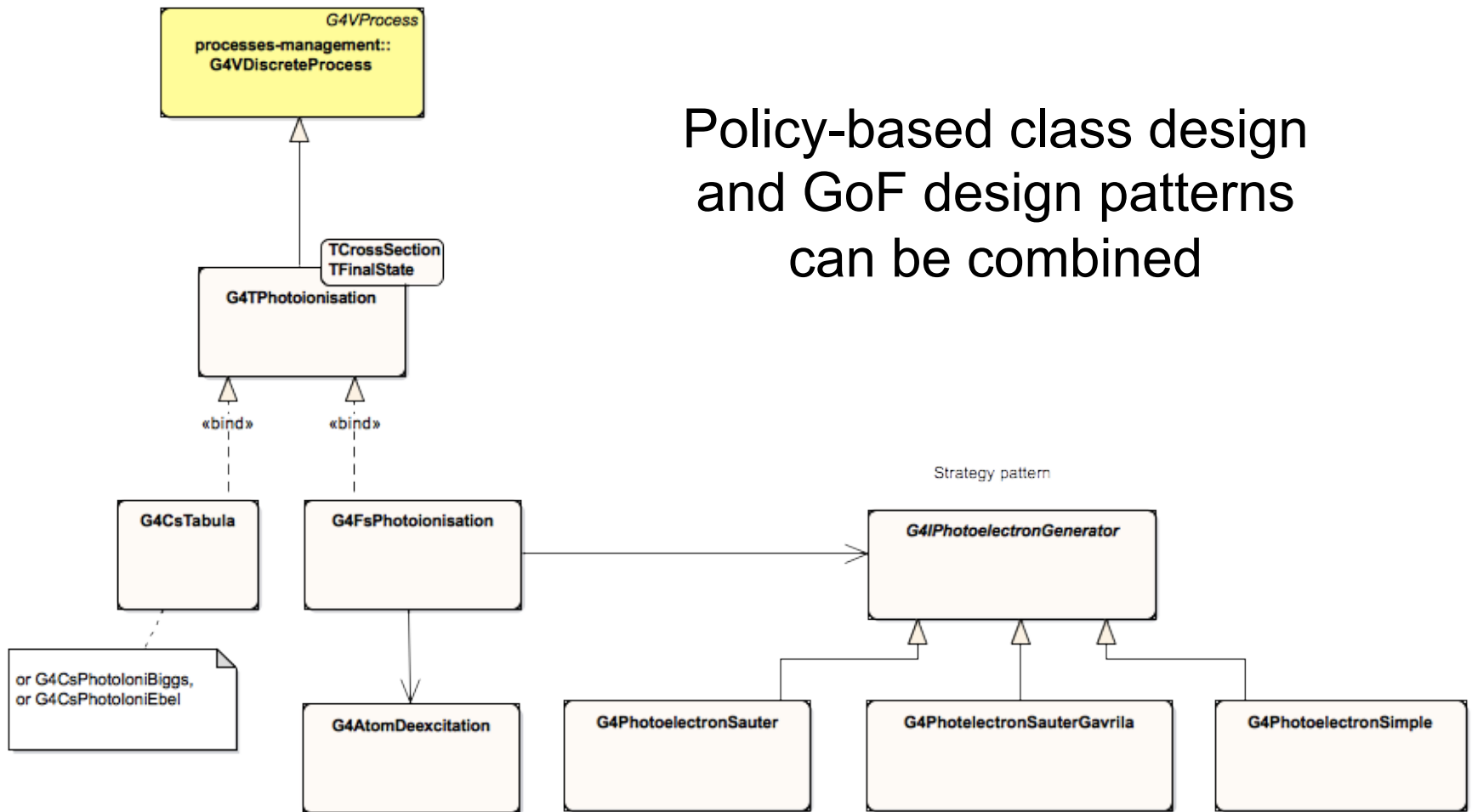
Photoelectric effect

Standard photoelectric effect model



Photoelectric effect

Photoionisation



Policy-based class design
and GoF design patterns
can be combined

Physics validation = unit tests

IEEE TRANSACTIONS ON NUCLEAR SCIENCE, VOL. 63, NO. 2, APRIL 2016 1117

Validation of Cross Sections for Monte Carlo Simulation of the Photoelectric Effect

Min Cheol Han, Han Sung Kim, Maria Grazia Pia, Tullio Basaglia, Matej Batič, Gabriela Hoff, Chan Hyeong Kim, and Paolo Saracco

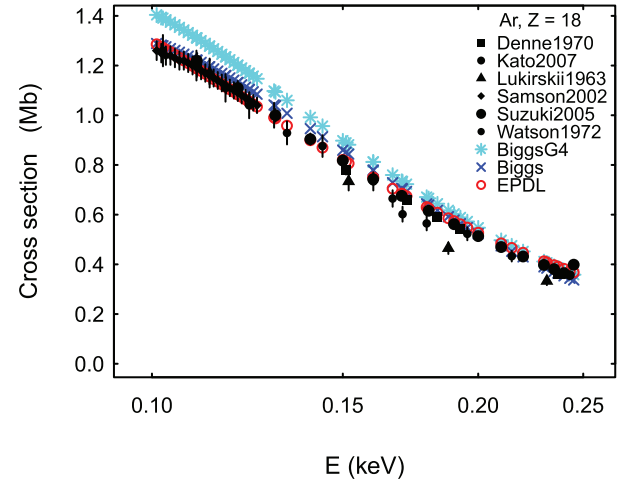


Fig. 28. Total photoionization cross section for argon as a function of photon energy, above 100 eV: original and modified Biggs-Lighthill parameterizations exhibit different behavior with respect to experimental data.

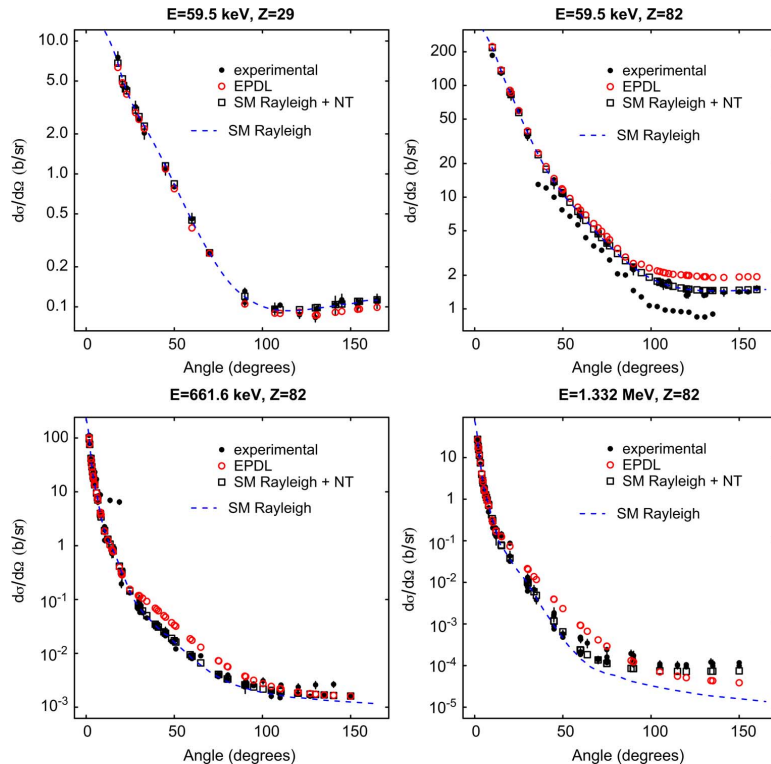


Fig. 2. Differential cross section as a function of scattering angle for representative energies and target elements: experimental measurements (black circles), calculations based on S-matrix (SM, black empty squares) and on EPDL (red circles). The S-matrix calculations account for Rayleigh scattering and nuclear Thomson scattering; S-matrix calculations limited to the Rayleigh scattering amplitude are shown as a blue dashed line. The sources of experimental data are documented in Tables III and IV.

1636 IEEE TRANSACTIONS ON NUCLEAR SCIENCE, VOL. 59, NO. 4, AUGUST 2012


Photon Elastic Scattering Simulation: Validation and Improvements to Geant4

Matej Batič, Gabriela Hoff, Maria Grazia Pia, and Paolo Saracco

Benefit from policy-based class design

Caveat


Designing with policy classes is inherently **compile-time bound**



Policy classes should be used for those aspects of a design that are **fixed at runtime**

Overusing policies might lead to excessive recompilation, code bloating and rigid architectures

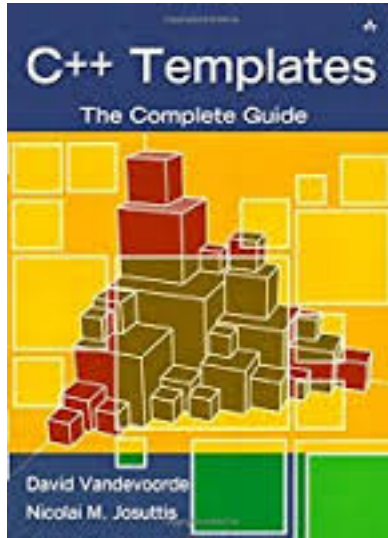
Decomposing classes and choosing the right interface for each policy is **critical**



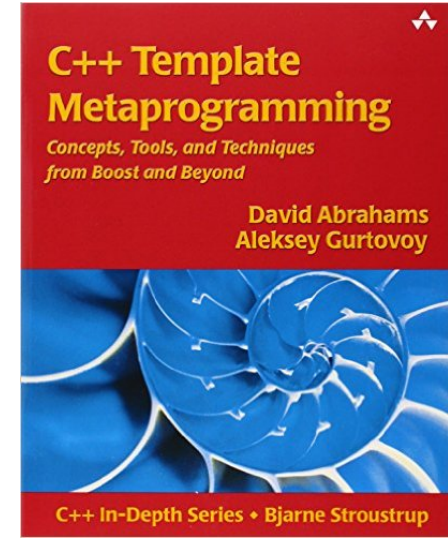
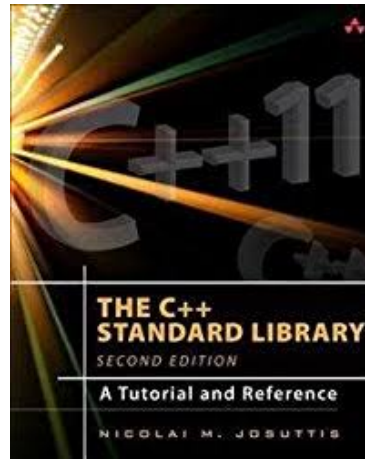
The developer is responsible for identifying **orthogonal policies**

Choosing a non-orthogonal decomposition leads to policies that **depend** on each other

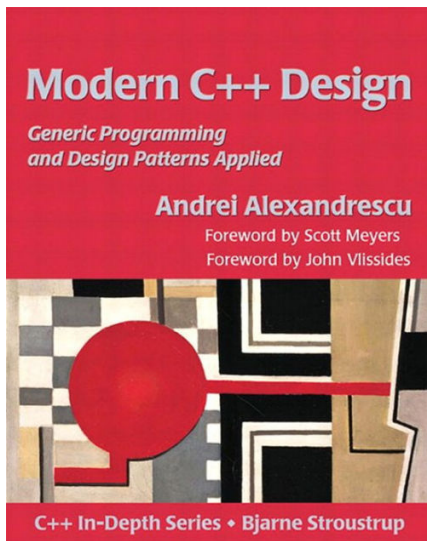
Suggestions for reading



Basic reading



Techniques and tools



Advanced reading
Focus on design

Homework with your own code

1. Encapsulate algorithms in a Strategy pattern
2. Redesign the code using policy classes
3. Evaluate maintainability, testability
4. Measure computational performance