Design Decisions for a Library

Bjarne Stroustrup

AT&T Labs – Research

http://www.research.att.com/~bs

Abstract

I will examine the decisions made for a library I wrote and show how the constraints on the problem to be solved are reflected in the code. The discussion will touch upon separation of interfaces and implementations, type safety, generality of class hierarchies,non-intrusive addition of functionality, external and internal representations, tradeoffs between generic and object-oriented programming, use of exceptions, and more. The discussion will focus on the fundamental and on concrete code, rather than on language subtleties or abstract arguments.

75 mins plus Q&A

Aims an Assumptions

- Aims
  - To show the effect of design decisions on code
  - To discuss design decisions and tradeoffs
  - To give examples of effective
    - design techniques
    - Programming techniques
  - To give me some feedback on XTI
- Assumptions
  - You are experienced C++ programmers
  - You would like better structured, more maintainable code
- Original idea: dialog with experienced designers

Work in Progress

- I’m still modifying the code in significant ways
  - So far, only two (experimenting) users
  - Not every part of the library has been completed
- That means
  - Lack of use
  - Lack of experience
  - Design errors
  - Bugs
- Opportunities for improvements

Overview

- The Project
- General ideal and ideas
- External polymorphism
- Memory management
- Hierarchy design
  - Abstract and concrete classes
- Scope: The central abstraction
- Classes, interfaces, operations
- Final thoughts

The Project

- Communication with remote mobile device
  - Calling interface
    - CORBA, DCOM, Java RMI, …, homebrew interface
  - Transport
    - TCP/IP, XML, …, homebrew protocol

- Big, Ugly, Slow, Proprietary, …
  - Why can’t I just write ISO Standard C++?

The Project

- The obvious code:

  X x = m.f(y);

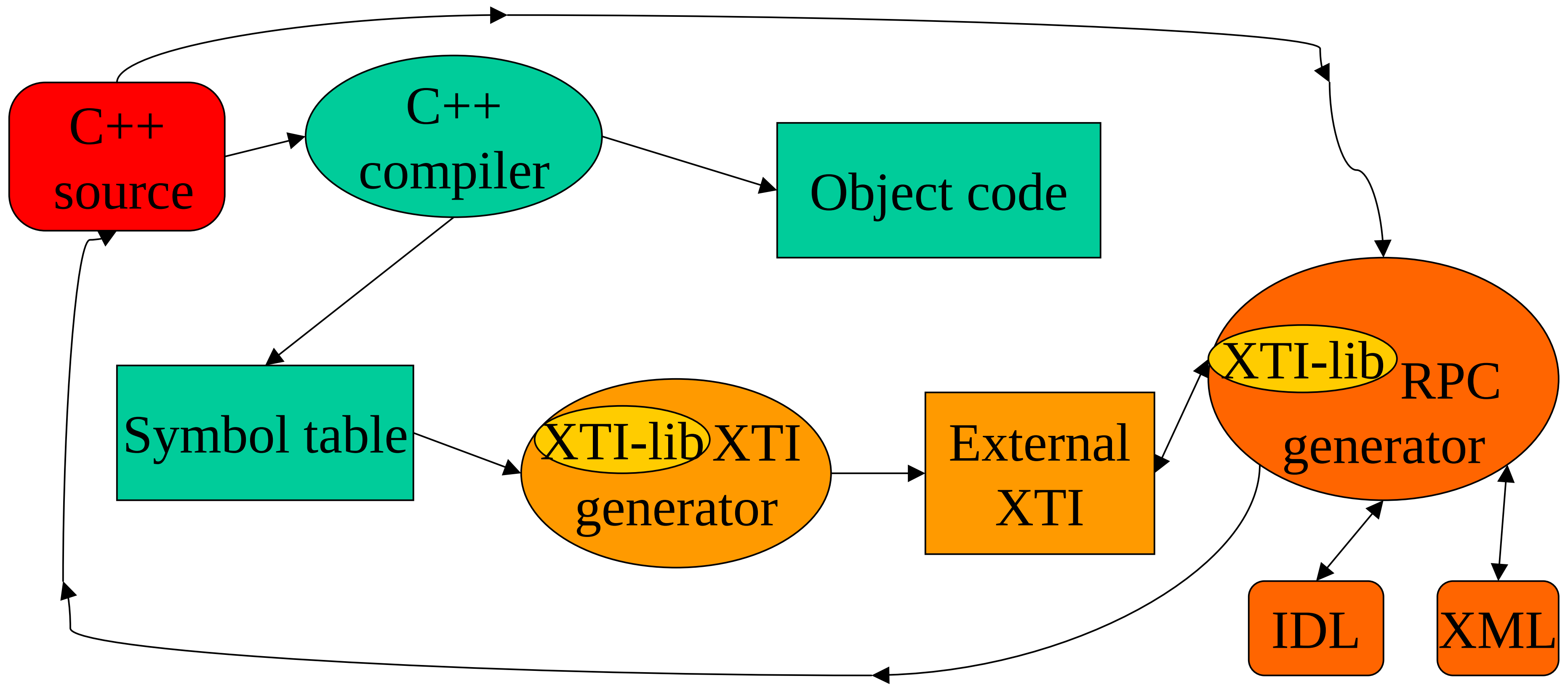
- Given some IDL, this generates something like

  Handle<X> h = send(message(m,&M::f,y));

  X x = unpack_message(h.result());

- If you know X, you can write **message()** and **unpack_message()**
  - In C++, only compilers and humans "know **X**"
  - An old problem with a known solution:
    - Extended type information, reflection, class objects, …
  - This solution is also the solution to many other problems
    - But you have to own a compiler to generate the information

Generation of inter-object communication code



Classical OO approach/solution

- We need to model a system
  - The C++ type system
- In memory and persistent representation
- Open ended problem
  - Ease of programming essential
  - Extensibility essential
- Run-time resolution of operations essential
  - A representation is read from input, possibly incrementally
  - Virtual functions essential for simple programming
- Many implementations likely
  - Data representations will differ
  - interfaces will be common
- Long-lived system
  - Complete information hiding essential

XTI Ideals

- Direct representation of C++’s complete type system as classes
  - built-in types, classes, templates, …
- Learning effort proportional to complexity of task
  - Not just a data structure
- Programming effort proportional to complexity of task
- Hide the representation
  - define the semantics, not the representation
- The set of operations on the XTI must be extensible
- Minimal overhead (run-time and space)
  - XTI is read in only when needed
- No integration with compiler
  - No language feature must depend on XTI

Basic Idea

Classical hierarchy of types, etc.:

$$\begin{tikzcd}
 & \textbf{Scope} & & \textbf{Type} & \\
\textbf{Namespace} \arrow[ur] & & \textbf{Class} \arrow[ul] \arrow[ur] & \textbf{Function} \arrow[u] & \textbf{Pointer} \arrow[ul]
\end{tikzcd}$$

Each node type will provide operations useful for manipulating that kind of C++ construct, e.g., a Function will hold information about number of arguments, type of arguments, etc.

XTI

- A set of classes/objects representing C++ declarations:

```
Program prog(“my_types”);
if (prog.global_scope[“My_vec”].is_class()) { /* … */ }
for (scope::iterator p = prog.begin(); p!=prog.end(); ++p) p->xti_name();
```

- Represent anything that the C++ type system can
  - Classes, enumerations, typedefs, templates, namespaces, functions, non-local variables
  - Not code, local types, local variables

XTI: eXtended Type Information

- Use for run-time resolution
  - Extend **type_info**
  - or key on **typeid**
  - or lookup based on name
- Use for program analysis
  - E.g. ODR verification, version consistency checking
- Use for program transformation
  - E.g. IDL generation

External polymorphism

- A tree representing the C++ type system will have dozens of types of nodes
  - Could have hundreds
- “climbing a tree” with dozens of node types using **if-then-else** and **switch** statements is tedious and error prone
  - That’s why we use virtual functions
- A user cannot add new virtual functions

- So, we need non-member virtual functions!

External Polymorphism

```
// The ideal: just define the needed functions and call them

void print(const Function& a) { /* print a function */ }
void print(const Pointer& a) { /* print a pointer */ }

void f(const XTI& obj, const XTI& o2)
{
    print(obj);     // call the right print, e.g., print(const Function&)
    print(o2);      // call the right print, e.g., print(const Pointer&)
}

// with a bit of "scaffolding," this can be done
```

External Polymorphism

```
struct XTI_obj { Kind k; };            // type field for optimizing resolution
void vprint(const XTI_obj& obj)  // “virtual print()”
    // find the exact type and let overload resolution do the rest
{
    switch(obj.k) {
    case FCT:       return print(dynamic_cast<const Function&>(obj));
    case PTR:       return print(dynamic_cast<const Pointer&>(obj));
    // …
    }
}
```

- how can we make “print” a parameter?
  - passing pointer to function won’t give us overload resolution
  - So, pass a function object calling the right function

External Polymorphism

```
template<class F> void vcall(F f, const XTI_obj& obj) // pass a function object
{
    switch(obj.k) {
    case FCT:    return f(dynamic_cast<const Function&>(obj));
    case PTR:    return f(dynamic_cast<const Pointer&>(obj));
    // …
    }
}

struct Fprint {
     template<class T> void operator()(const T& a) const { return print(a); }
};

Fprint Print;      // function object for print()

… vcall(Print,some_obj); …       // use vcall directly or use syntactic sugar

Void vprint(const XTI& obj) { return vcall(Print,obj); } // syntactic sugar
```

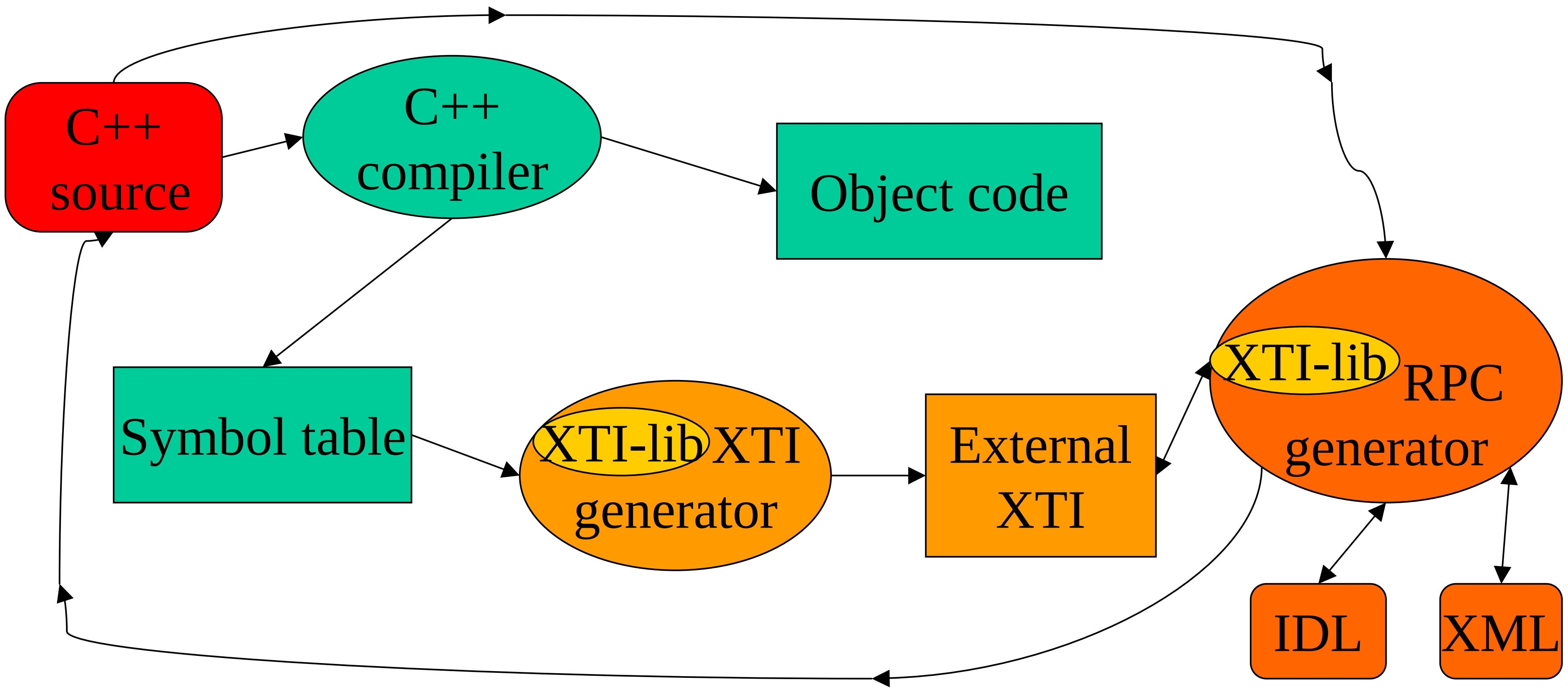
External polymorphism

- Works with a single rarely changed class hierarchy
  - **vcall()** provided by library
- Lots of details left out here
  - Return type, arguments
- Allows user to program as with virtual functions
  - Or better: just define all functions in one place
- No break of encapsulation
- Reasonably efficient
  - Simple calls can be inlined
  - In general, a **dynamic_cast** is needed
    - often, we can get away with a **static_cast**
  - You can eliminate "**Kind**" by using **typeid()** or **if-then-else +dynamic_cast**
    - But not efficiently with current compilers
- Key to easy extensibility
  - No code modification, no exposed data, no user navigation of hierarchy

XTI Design strategy

- Grow the program
  - Or rather grow the set of programs
    - External XTI
    - Compiler to external XTI
    - External XTI to internal XTI reader
    - Basic XTI library
    - Simple applications
  - Design a little, implement a little, test a little, redesign a little, …
    - A compiler is a potent design tool – consistency and completeness
  - Repeatedly complete a system that implement a subset of XTI
  - Feedback and ease of making changes is essential
- As opposed to
  - Design then code
  - Do one part of the system, then the next, then …

Generation of inter-object communication code



Why external XTI?

- C++ is unnecessarily hard to parse
  - I don’t want to be a compiler owner/maintainer
- As a target for different compilers
  - We initially use GCC debug information
- As a starting point for specifying types
  - XML is too verbose and unfriendly to humans
  - I don’t have a good XML/C++ toolset
    - XTI will help me get one

External XTI

```
i : int                              // int i;
C : class {                          // class C {
    m : const int                    //         const int m;
    mm : * const int                 //         const int* mm;
    f : (int,*char) double           //         double f(int,char*);
    f : (complex) C                  //         C f(complex);
}                                    // };
vector : <T> class {                 // template<class T> class vector {
   p : *T                            //         T* p;
   sz : int                          //         int sz;
}                                    // };
```

Note: strictly prefix, close-to-minimal notation

Human readable, human writable

User-interface principles I

- Type safe
- Abstract
  - No representation dependencies
- No memory management required of users
- Separate interfaces for
  - Plain users: immutable type information
    - Const references
  - Providers of XTI
    - Non-const pointers

Memory management

- Initial idea: reference counted smart pointer
  - Works, but inelegant
    - Why should “ordinary users” see pointers at all?
  - Many pointers are to non-shared objects
    - inefficient
- Second idea: only some pointers counted
  - Too complex to be manageable
- Third idea: let XTI own all pointers
  - Put every pointer to an XTI object in a vector
    - Delete elements of that vector “at end”
  - Don’t give users pointers: const references
  - Not every object needs to be on the free store: member objects

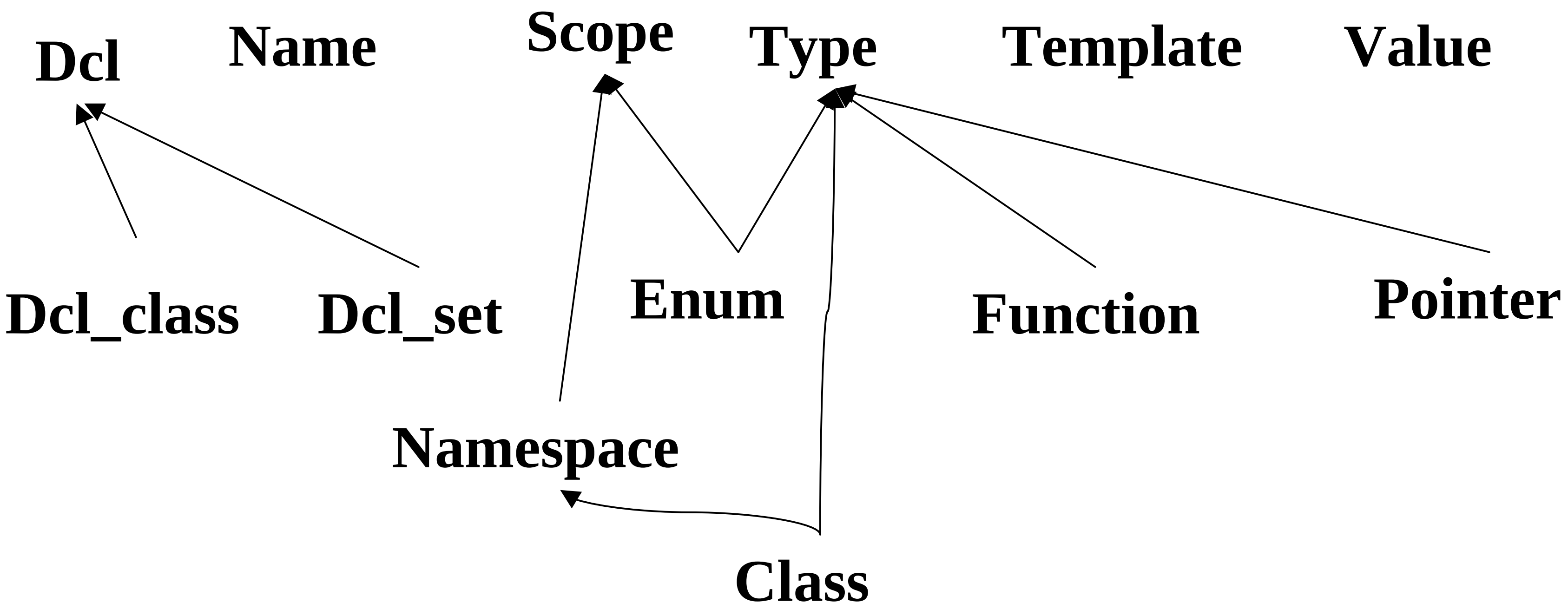
XTI_obj

```
struct XTI_obj {
    virtual  ~XTI_obj() { }             // the virtual destructor is essential
};

class Program_impl : public Program_rep {
    vector<XTI_obj*> owned;        // every XTI_objs is owned by a Program
    // …
    void add_obj(XTI_obj* p) { owned.push_back(p); }
    ~Program_impl() { /* delete all elements of owned */ }
}

template<class T, class A> T* make(A a)   // make all XTI_objs using make()
{
    T* p = new T(a);
    Program_impl::curr_prog->add_obj(p);            // eliminate need for static
    return p;
}
```

User-interface principles II

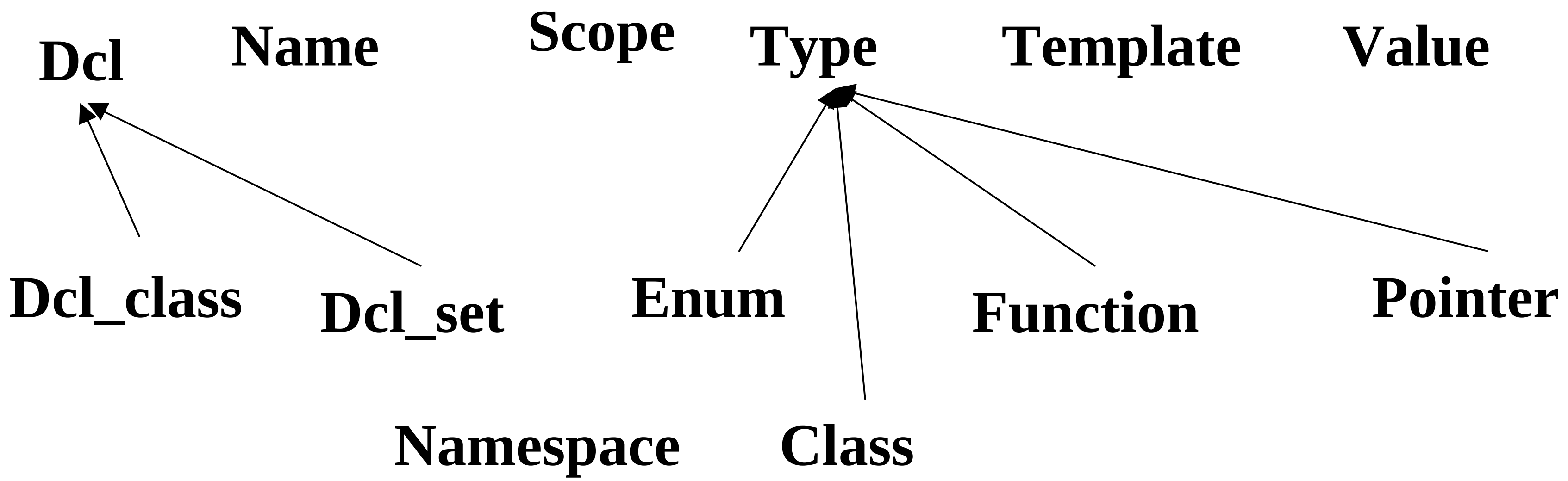
- Complete separation between
  - xti.h
    - Completely abstract
      - imagine multiple implementations, incremental build
    - For users of XTI
    - Scope, Class, Name, Pointer, …
    - References, iterators, const access only
      - But users can add attributes
  - concerte_xti.h
    - Encapsulated data
      - Concrete_xti has users: not just a struct
    - For builders of XTI
    - Scope_impl, Class_impl, Name_impl, Pointer_impl, …
    - Pointers, non-const access

XTI classes I



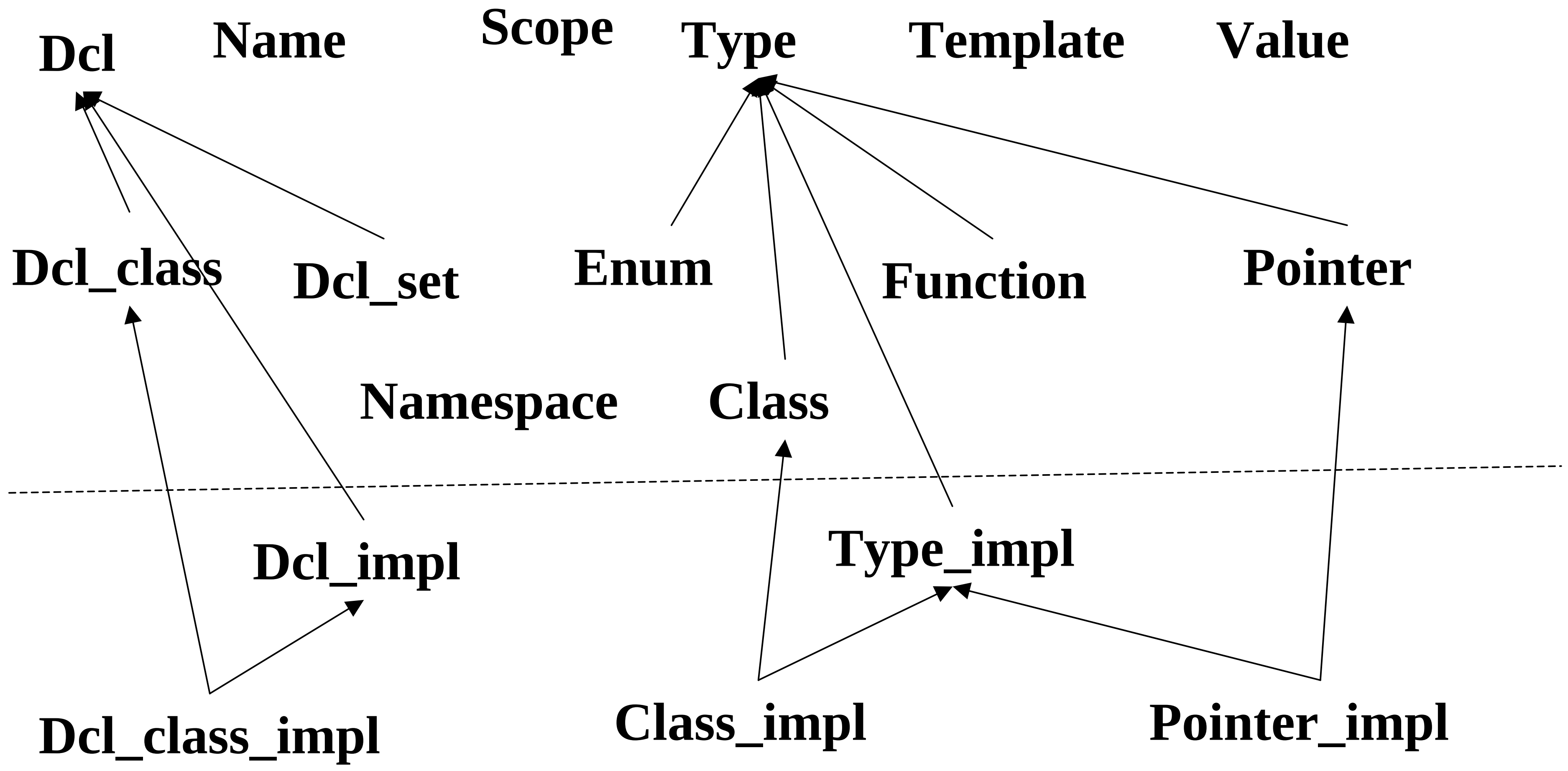
- A class is a namespace and a namespace is a scope
  - Beautiful code re-use
  - Messy class hierarchy, especially when implementations are considered

XTI classes II

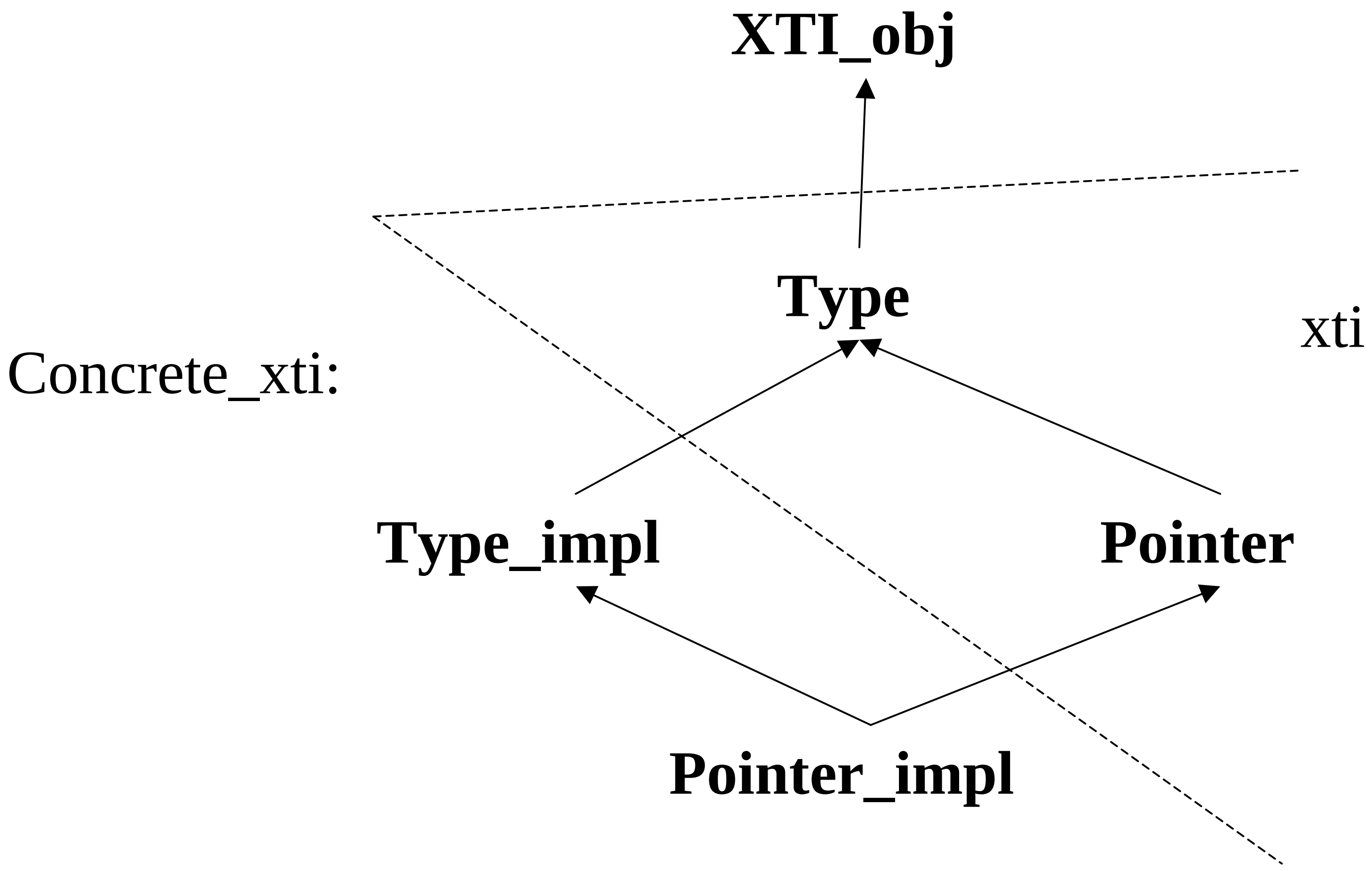


- Simpler structure
  - a namespace has a scope, a class has a scope
- Needs a few forwarding functions
  - Still not quite as elegant as class -> namespace -> scope
- Initial change took 15 minutes, but I keep needing more forwarding functions

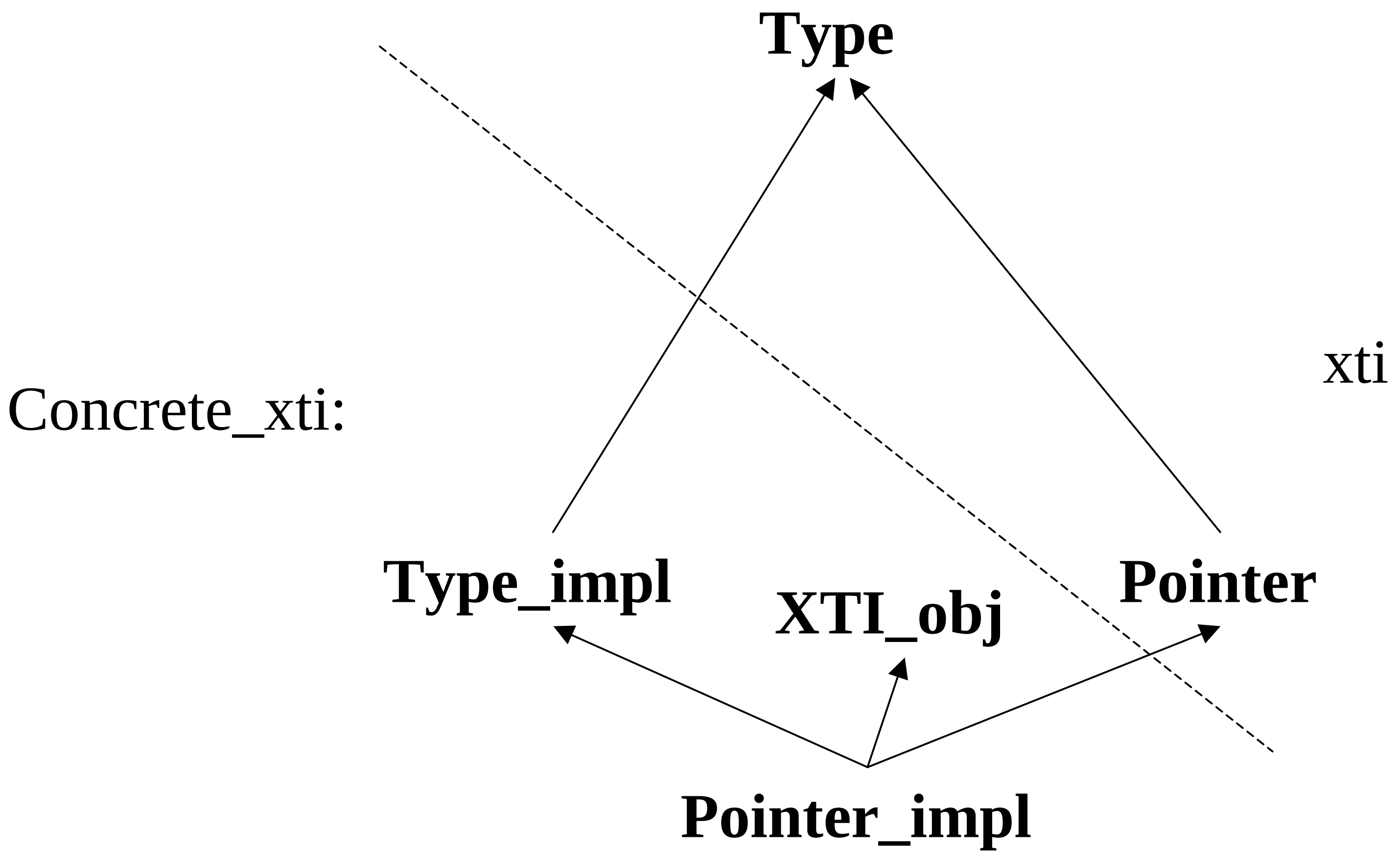
Concrete XTI classes



Hierarchy design



Hierarchy design



XTI classes

```
struct Type {      // common base class for all Types
    virtual Bits bits() const = 0;     // too low level; find something better

    virtual bool is_const() const = 0;
    virtual bool is_volatile() const = 0;

    virtual string type_name() const = 0;    // string representation
    virtual string xti_name() const = 0;     // string representation

//  virtual bool is_function() = 0;          // this is the beginning of a real mess
//  virtual const Function& get_function() = 0;      // find something better

    template<class T> bool is() const
            { return dynamic_cast<const T*>(this)!=0; }
    template<class T> const T& get() const
            { return dynamic_cast<const T&>(*this); }
};
```

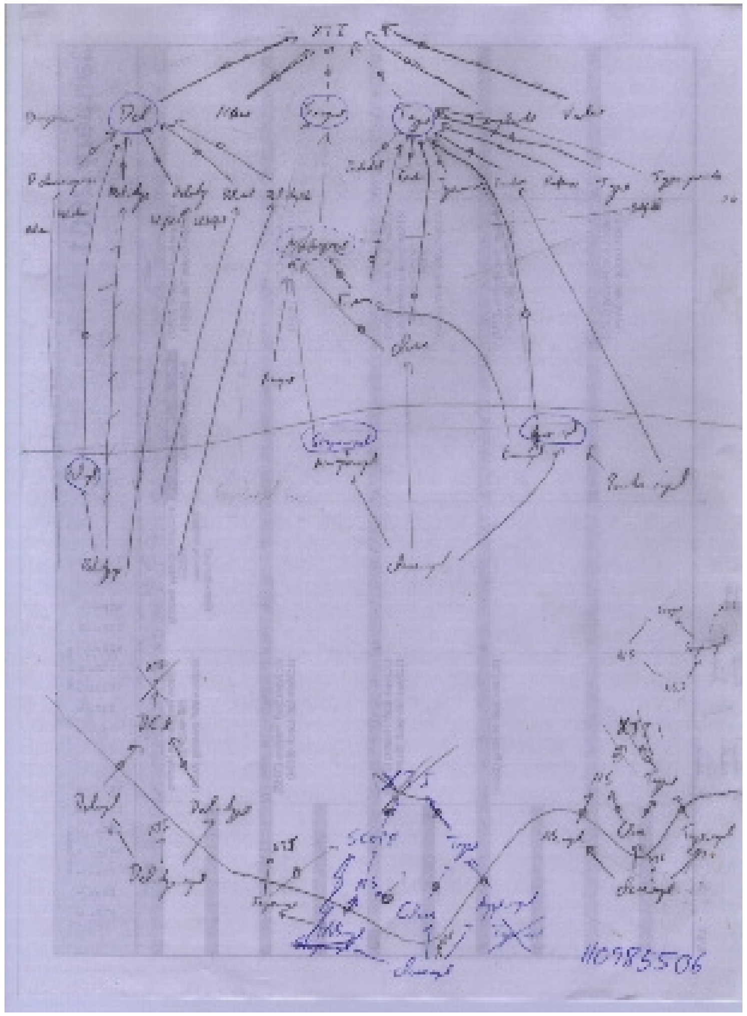
XTI classes

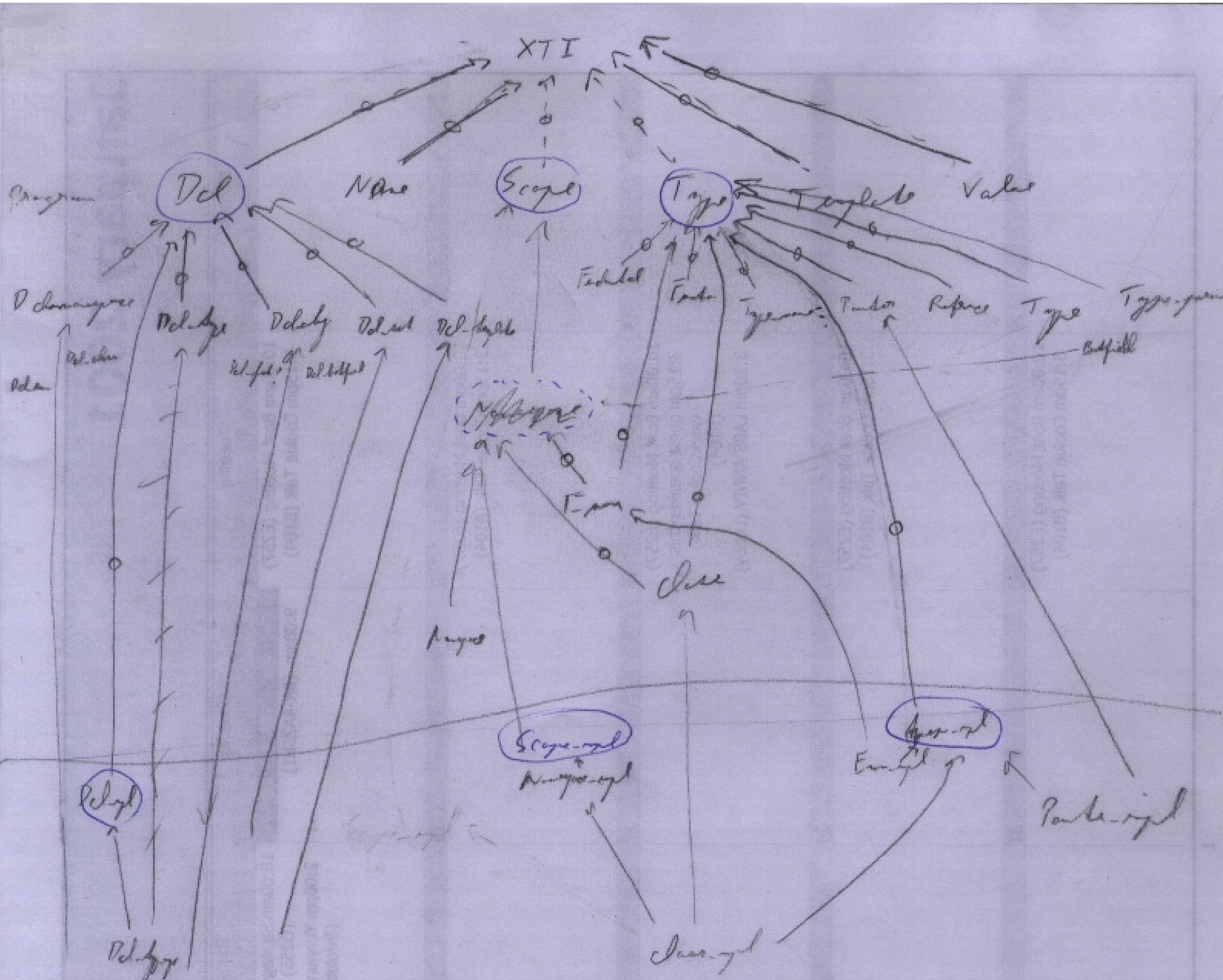
```
class Class : public virtual Type {
public:
    virtual Scope& members() = 0;
    virtual Scope& bases() = 0;

    virtual const string& name() const = 0;
    virtual const Name& get_name() const = 0;       // a Name knows its
                                                     // enclosing Scope
};
```

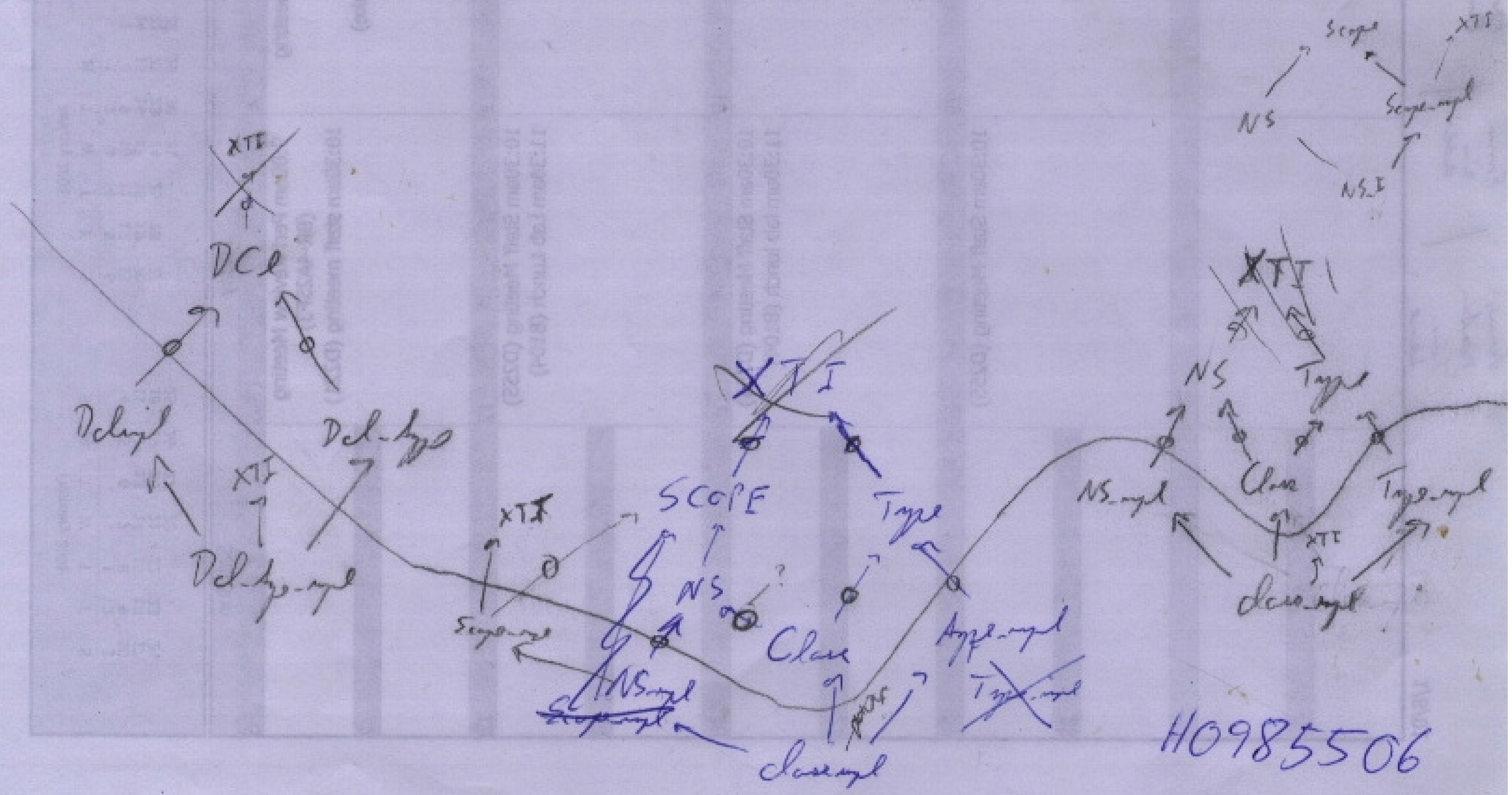
Concrete XTI classes

```
class Class_impl : public Class, public Type_impl, public XTI_obj {
    Scope_impl b;           // Scope members, not pointers to XTI_objs
    Scope_impl m;
public:
    Class_impl(Name_impl* n);    // constructor

    const Scope& members() { return m; } //  xti access (problem with attributes)
    const Scope& bases() { return b; }
    const string& name() const { return m.name(); }
    const Name& get_name() const { return m.get_name(); }

    string type_name() const { return cv_string()+m.name(); } // etc.
    string xti_name() const { return cv_string()+m.name(); }

    Scope_impl* base_impl() { return &b; }          // concrete_xti access
    Scope_impl* member_impl() { return &m; }
};
```

Real-world design tool ☺




XTI
Decl
Name
Scope
Type
Template
Value
Program
D namespace
Decl_type
Decl_obj
Decl_var
Decl_template
Decl_alias
Decl_func
Decl_bitfield
Decl_enum
Fundamental
Function
Typename
Pointer
Reference
Type
Type_parameter
Bitfield
Namespace
Enum
Class
Argument
Scope_impl
Namespace_impl
Decl_impl
Decl_type
Class_impl
Type_impl
Enum_impl
Pointer_impl


XTI
DCl
Declimpl
Del-Impl
XTI
Del-Impl-impl
XTI
Scopeimpl
SCOPE
NS
~~Scopeimpl~~
NSimpl
XTI
Class
closeimpl
Type
Implimpl
~~Typeimpl~~
NS_impl
NS
Type
Class
XTI
closeimpl
Typeimpl
XTI
Scope
XTI
NS
Scopeimpl
NSI

H0985506

Scope

- The central abstraction / simplification
  - Used for all sequences of (name,value) pairs
    - Global scope
    - Namespace
    - Class
    - Enumeration
    - Argument list
    - Template argument list
    - Attribute list
  - Access: Indexed, iterator, and name
  - Order preserving

Iterators

- First idea
  - Maintain user/implementer distinction
    - Abstract class **Iter** (pure interface, holding no data**)**
    - Concrete class **Iter_impl** (carrying data, different representations can be used without affecting users)
- But that's impossible
  - Users need to create and copy **Iter**s
- And it's wrong (over-abstraction)
  - An random-access iterator is an abstraction for something holding 0,1,2,3,…
  - We don't need an abstraction for an abstraction
- Obvious solution
  - **Iter** is a type holding values in the range [0,n)
    - Range checked for type safety

Class Scope

```
struct Scope {
	// define iterator type: basically range checked long
public:
	virtual const Dcl& operator[](const string&) const = 0;		// map style access
	virtual const Dcl& operator[](const char*) const = 0;		// v[0] ambiguous?

	virtual const Dcl& operator[](int) const = 0;			// vector style access
	virtual size_t size() const = 0;

	virtual bool has_member(const string& s) const = 0; // convenience only
	virtual bool has_member(const char* s) const = 0;


	virtual Iter begin() const = 0;				// STL container style access
	virtual Iter end() const = 0
	virtual Iter position(const string& s) = 0;
	virtual Iter position(const char* s) = 0;
	virtual Iter position(int i) = 0;

	virtual const string& name() const = 0;
	virtual const Name& get_name() const = 0;	// Name contains enclosing Scope
};
```

Class Scope_impl

```
class Scope_impl : public virtual Scope, public XTI_obj {
    // a scope maps strings (not Names) to Dcls: you can look up by variable name (string)
    // or by index (declarations are numbered [0,n), and you can iterate through a scope

    vector<Dcl_impl*> v;                // dclclarations in declaration order
    mutable map<string,int> m;          // map into v index
    Scope_impl* enclosing;
    Name* n;                            // optional name

    void build_map() const; // the m is built only when needed
    Dcl& at(int i) const;   // range checked access
    // …
};
```

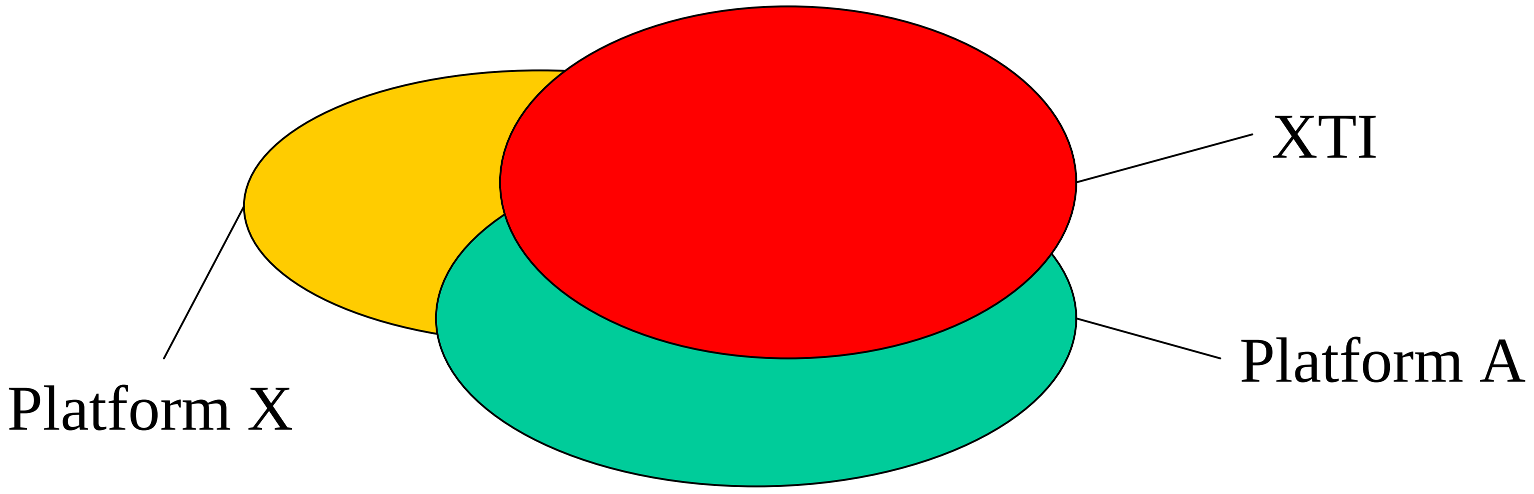
What should be represented?

- Answer any question that can be asked about declarations in source code after preprocessing
  - Typedefs, constants, templates, template specializations, access control, declaration order, variable names, …
  - Probably not: line numbers, file names, comments
- **Program** roughly equivalent to translation unit
  - But a user can write program to merge external XTI files
- Allow addition of “attributes”
  - Sizeof, offset
  - Line numbers, file names
  - By compiler, by XTI writer, and by programmer

What can/cannot be done with XTI?

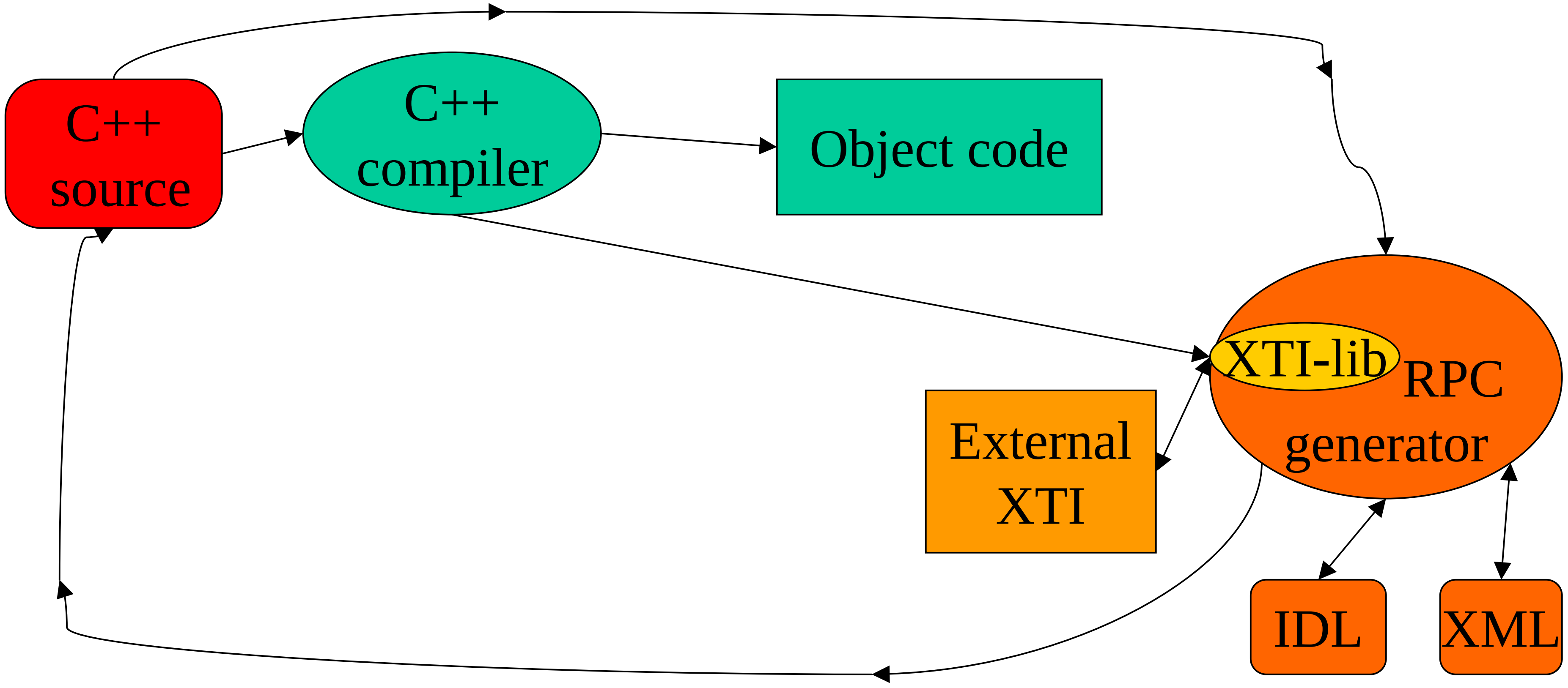
- Read in XTI for the running program itself
  - Find XTI for an object (using RTTI) or for a class (by name)
- Read in XTI
  - make a modified copy
  - write XTI, C++, IDL, XML, …
- Read XTI from two sources
  - Compare, merge, …
- Add new class or operation to XTI
  - but not use it in the current program
  - Output code, compile, link, and use
- Look at XTI, see what functions are available
  - But not call them
  - Build function table, link it, then call

Relationship with platform services



- XTI can
  - be common interface to common services
    - Minimizing a program’s platform dependencies
  - extend platform services to cover Standard C++
    - Platforms often support “common language facilities” only
  - support platform-specific facilities through optional extensions to XTI
    - potential for thin layer common interfaces to non-universal services
    - Hard to do

Generation of inter-object communication code



Simple code example

```
Program prog;
Prog.add(cin);    // read some XTI
Scope& p = prog.global_scope();

for (Scope::iterator pp=p.begin(); pp!=p.end(); ++pp)
            cout << pp->name() << ‘\n’;        // name of global

for (Scope::iterator pp=p.begin(); pp!=p.end(); ++pp)          // tree climbing
    if (pp->has_type())
            cout<< pp->name() << " : " << pp->get_type().xti_name() << '\n';
    else
            cout<< pp->name() << "\n";

for (Scope::iterator pp=p.begin(); pp!=p.end(); ++pp)   // use virtual function
    pp->print_xti(cout);   // xti declarations
```

Attributes

```
class Dcl_impl : public virtual Dcl {
    /*      base for Dcl_*_impl classes
            never instantiated by itself, type_impl() is virtual to ensure that
            (even though not every Dcl_*_impl class can meaningfully define it)
            not a XTI_obj
    */
    Name_impl* n;           // a declaration knows its name
    Bits mod;               // access, etc.
    Scope_impl* attr;       // #(name,value) attributes // use member object
public:
    // …
};
```

Overloading

- 1st idea, fct_type_list
  - A function can have several types
  - Doesn’t work
    - type + non-type overload
- 2nd idea, type_list
  - Doesn’t work
    - access control
    - Inline, mutable, static, etc. apply to declaration, not type
- 3rd idea, Dcl_set
  - Makes it necessary to change of type after declaration
    - E.g., Dcl_fct to Dcl_set
    - Later useful for dealing with undeclared names

How many classes?

- How to represent declarations
  - Alternatives
    - Dcl_ordinary
    - Dcl_type, Dcl_object, Dcl_fct, … (not distinguished in C++ grammar)
  - Most application code wants to distinguish: use several classes
    - User writes N virtual functions
- How to represent a fundamental type
  - Alternatives
    - Fundamental
    - Int, Double, Char, …
  - Most application code isn’t interested in which fundamental type is presented: use a single “Fundamental” class
    - User writes one virtual function containing tests
    - Const/volatile affected that decision

Concrete XTI interface

- How to allow updates
  - Get/set
    - ugly, inherently unstructured, and error-prone
  - Get + whole-object assignment
    - Doesn’t work with derived classes – slicing.
  - Return pointers to interesting members + a few set functions

  - Co-variant return defeated by const

Program

- A **Program** object
  - is the root of an XTI data structure
  - is responsible for deleting XTI objects that it owns
- How does a user get a first Program object
  - Cannot be abstract
    - Factory object with default (general, ugly, works)
    - C-style implementation supplied new_program()
  - Needs to be initialized from somewhere
- A program can contain several **Program** objects
  - So you can write programs comparing types from different sources, e.g. do global consistency checks

Query interface

- XTI has a minimal interface
  - Logically complete
  - Needs support library for convenience
- Programs can be written as sets of overloaded functions plus a simple loop
  - External polymorphism essential for extensibility
- Define composable function objects for use with algorithms

```
is_public { /* …*/ };
is_virtual { /* … */ };
find(s.begin(),s.end(), compose2(logical_and<bool>,Is_public,Is_virtual));
```

- Use lambdas to simplify notation?

```
Lambda x;
find(s.begin(),s.end(),is_public(x)&&is_virtual(x));
```

Random observations

- Strive for simplicity and symmetry
  - You can’t optimize every dimension at once
    - Settle for balance
- Design tools
  - Colleagues
  - simple diagrams
  - Compiler
  - parser generator (for checking only)

Current status

- Specification
  - Still mutating
  - What information should be available as XTI? Why? How?
- XTI class hierarchy
  - Still mutating (385+366 lines, about 60 classes)
- GCC produces hard-to-read type information
  - Support patch in 3.0
- GCC output to external XTI
  - incomplete (approximately 800 lines)
- External XTI reader
  - Incomplete (1066 lines)
- Query interface
  - Back of the envelope draft