LEESA: Toward Native XML Processing Using Multi-paradigm Design in C++

BOOST LIBRARIES CONFERENCE 2011

May 16, 2011

Dr. Sumant Tambe
Software Engineer
Real-Time Innovations

Dr. Aniruddha Gokhale
Associate Professor of EECS Dept.
Vanderbilt University

www.dre.vanderbilt.edu/LEESA

Outline

- XML Programming in C++. Specifically, data binding
- What XML data binding stole from us!
- Restoring order: LEESA
- LEESA by examples
- LEESA in detail
  - Architecture of LEESA
  - Type-driven data access
  - XML schema representation using Boost.MPL
  - LEESA descendant axis and strategic programming
  - Compile-time schema conformance checking
  - LEESA expression templates
- Evaluation: productivity, performance, compilers
- C++0x and LEESA
- LEESA in future

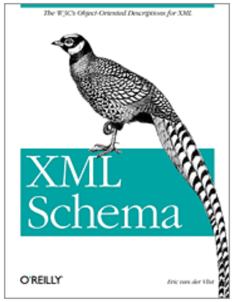
XML Programming: A Paradigm

XML Programming Paradigm

- Type system
  - Regular types
  - Anonymous complex elements
  - Repeating subsequence

<?xml?>

- XML data model
  - XML information set (infoset)
  - E.g., Elements, attributes, text, comments, processing instructions, namespaces, etc. etc.

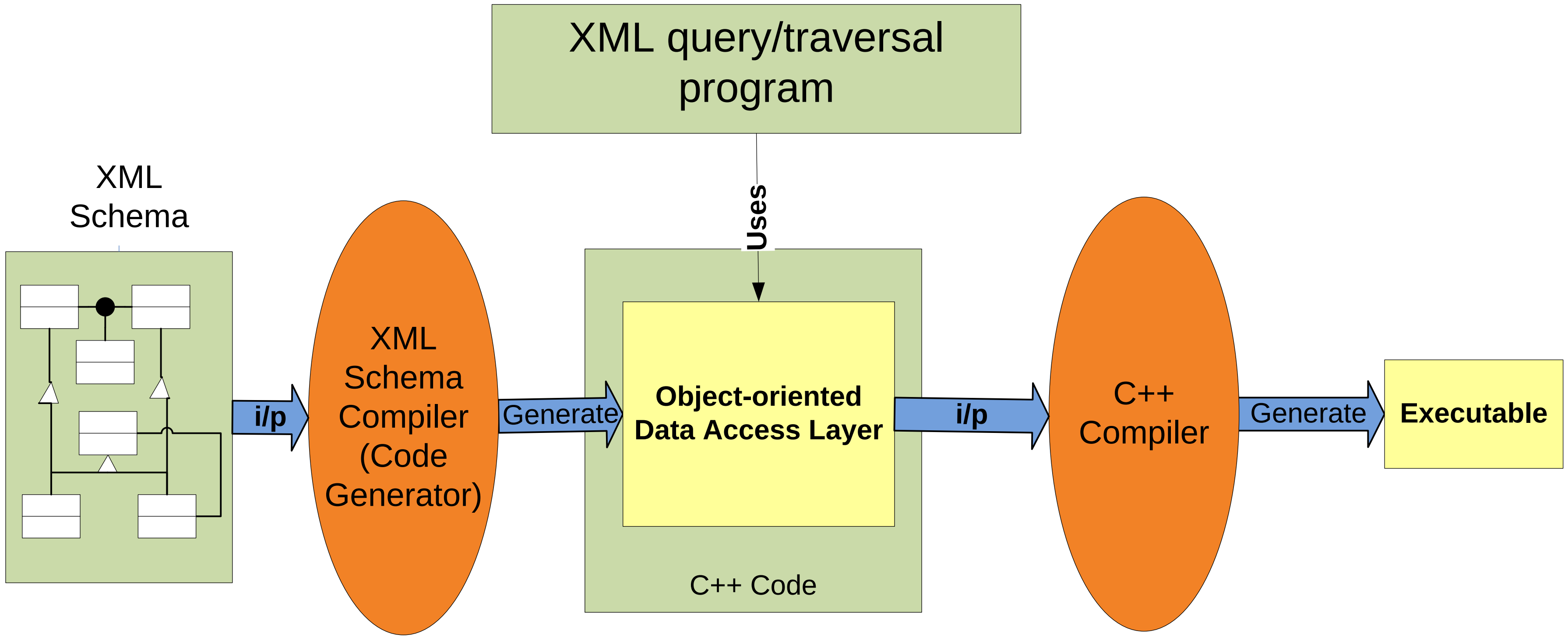


- Schema languages
  - XSD, DTD, RELAX NG
- Programming Languages
  - XPath, XQuery, XSLT
- Idioms and best practices
  - XPath: Child, parent, sibling, descendant axes; wildcards

XML Processing in C++

- Predominant categories & examples (non-exhaustive)
- DOM API
  - Apache Xerces-C++, RapidXML, Tinyxml, Libxml2, PugiXML, lxml, Arabica, MSXML, and many more …
- Event-driven APIs (SAX and SAX-like)
  - Apache SAX API for C++, Expat, Arabica, MSXML, CodeSynthesis XSD/e, and many more …
- XML data binding
  - Liquid XML Studio, Code Synthesis XSD, Codalogic LMX, xmlplus, OSS XSD, XBinder, and many more …
- Boost XML??
  - No XML library in Boost (as of May 16, 2011)
  - Issues: very broad requirements, large XML specifications, good XML libraries exist already, encoding issues, round tripping issues, and more …

XML Data Binding



- Process
  - Automatically generate vocabulary-specific classes from the schema
  - Develop application code using generated classes
  - Parse an XML into an object model at run-time
  - Manipulate the objects directly (CRUD)
  - Serialize the objects back to XML

XML Data Binding

- Example: Book catalog xml and xsd

```
<catalog>
  <book>
    <name>The C++ Programming Language</name>
    <price>71.94</price>
    <author>
      <name>Bjarne Stroustrup</name>
      <country>USA</country>
    </author>
  </book>
  <book>
    <name>C++ Coding Standards</name>
    <price>36.41</price>
    <author>
      <name>Herb Sutter</name>
      <country>USA</country>
    </author>
    <author>
      <name>Andrei Alexandrescu</name>
      <country>USA</country>
    </author>
  </book>
</catalog>
```

```
<xs:complexType name=“book”>
 <xs:sequence>
   <xs:element name="name" type="xs:string" />
   <xs:element name="price" type="xs:double" />
   <xs:element name="author" maxOccurs="unbounded">
    <xs:complexType>
     <xs:sequence>
      <xs:element name="name" type="xs:string" />
      <xs:element name="country" type="xs:string" />
     </xs:sequence>
    </xs:complexType>
  </xs:element>
 </xs:sequence>
</xs:complexType>

<xs:element name="catalog">
 <xs:complexType>
  <xs:sequence>
    <xs:element name=“book”
                type=“lib:book"
                maxOccurs="unbounded">
    </xs:element>
  </xs:sequence>
 </xs:complexType>
</xs:element>
```

XML Data Binding

- Example: Book catalog xsd and generated C++ code

```xml
<xs:complexType name="book">
 <xs:sequence>
   <xs:element name="name"
               type="xs:string" />
   <xs:element name="price"
               type="xs:double" />
   <xs:element name="author"
               maxOccurs="unbounded">
    <xs:complexType>
     <xs:sequence>
      <xs:element name="name"
                  type="xs:string" />
      <xs:element name="country"
                  type="xs:string" />
     </xs:sequence>
    </xs:complexType>
  </xs:element>
 </xs:sequence>
</xs:complexType>

<xs:element name="catalog">
 <xs:complexType>
  <xs:sequence>
    <xs:element name="book"
                type="lib:book"
                maxOccurs="unbounded">
    </xs:element>
  </xs:sequence>
 </xs:complexType>
</xs:element>
```

```cpp
class author {
  private:
          std::string name_;
          std::string country_;
  public:
          std::string get_name() const;
          void set_name(std::string const &);
          std::string get_country() const;
          void set_country(std::string const &);
};

class book {
  private: std::string name_;
          double price_;
          std::vector<author> author_sequence_;
  public:  std::string get_name() const;
          void set_name(std::string const &);
          double get_price() const;
          void set_price(double);
          std::vector<author> get_author() const;
          void set_author(vector<author> const &);
};
class catalog {
  private:
          std::vector<book> book_sequence_;
  public:
          std::vector<book> get_book() const;
          void set_book(std::vector<book> const &);
};
```

XML Data Binding

- Book catalog application program
  - Example: Find all author names

```
std::vector<std::string>
get_author_names (const catalog & root)
{
  std::vector<std::string> name_seq;
  for (catalog::book_const_iterator bi (root.get_book().begin ());
       bi != root.get_book().end ();
       ++bi)
  {
    for (book::author_const_iterator ai (bi->get_author().begin ());
         ai != bi->get_author().end ();
         ++ai)
    {
      name_seq.push_back(ai->name());
    }
  }
  return name_seq;
}
```

- Advantages of XML data binding
  - Easy to use
  - Vocabulary-specific API
  - Type safety
  - C++ programming style and idioms
  - Efficient

But, where is “XML programming” ?

- We lost something along the way. A lot actually!
- Loss of succinctness
  - XML child axis replaced by nested for loops
  - Example: Find all author names

Using XPath (1 line)

```
/book/author/name/text()
```

Using XML data binding (20 lines)

```
std::vector<std::string>
get_author_names (const catalog & root)
{
  std::vector<std::string> name_seq;
  for (catalog::book_const_iterator bi =
         root.get_book().begin ();
       bi != root.get_book().end ();
       ++bi)
  {
    for (book::author_const_iterator ai =
           bi->get_author().begin ());
         ai != bi->get_author().end ();
         ++ai)
    {
      name_seq.push_back(ai->name());
    }
  }
  return name_seq;
}
```

But, where is “XML programming” ?

- Loss of expressive power
  - Example: “Find all names recursively”
  - What if catalogs are recursive too!
  - Descendant axis replaced by manual recursion. Hard to maintain.

Using XPath (1 line)

```
//name/text()
```

```
<catalog>
  <catalog>
    <catalog>
      <catalog>
        <book><name>...</name></book>
        <book><name>...</name></book>
      </catalog>
      <book>...</book>
      <book>...</book>
    </catalog>
    <book>
      <name>...</name>
      <price>...</price>
      <author>
        <name>...</name>
        <country>...</country>
      </author>
    </book>
    <book>...</book>
    <book>...</book>
  </catalog>
</catalog>
```

Using XML data binding using BOOST_FOREACH (20+ lines)

```
std::vector<std::string> get_author_names (const catalog & c)
{
  std::vector<std::string> name_seq;
  BOOST_FOREACH(const book &b, c.get_book())
  {
    BOOST_FOREACH(const author &a, b.get_author())
    {
      name_seq.push_back(a.name());
    }
  }
  return name_seq;
}

std::vector<std::string> get_all_names (const catalog & root)
{
  std::vector<std::string> name_seq(get_author_names(root));
  BOOST_FOREACH (const catalog &c, root.get_catalog())
  {
    std::vector<std::string> names = get_all_names(c);
    name_seq.insert(names.begin(), names.end());
  }
  return name_seq;
}
```

But, where is “XML programming” ?

- Loss of XML programming idioms
  - Cannot use “wildcard” types
  - Example: Without spelling “Catalog” and “Book”, find names that are exactly at the third level.

Using XPath (1 line)

```
/*/*/name/text()
```

Using XML data binding

```
std::vector<std::string>
get_author_names (const catalog & root)
{
  std::vector<std::string> name_seq;
  . . .
  . . .

  return name_seq;
}
```

- Also known as structure-shyness
  - Descendant axis and wildcards don’t spell out every detail of the structure
- Casting `Catalog` to `Object` class isn’t good enough
  - `object.get_book()` → compiler error!
  - `object.get_children()` → Inevitable casting!

Wait!! Use XPath API

- Hybrid approach: Pass XPath expression as a string

Using XML data binding + XPath

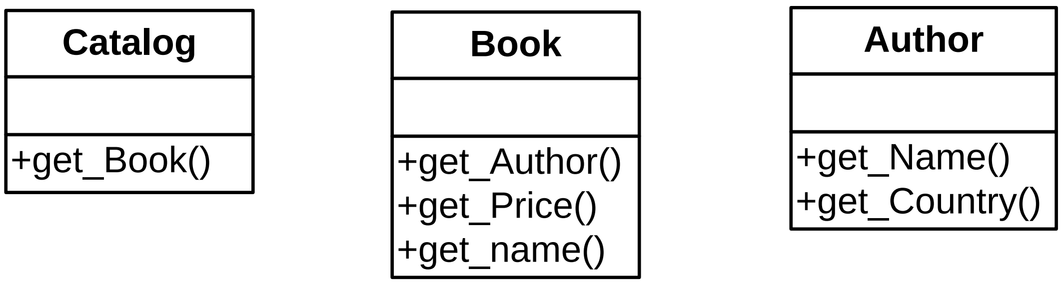
```
DOMElement* root (static_cast<DOMElement*> (c._node ()));
DOMDocument* doc (root->getOwnerDocument ());

dom::auto_ptr<DOMXPathExpression> expr (
  doc->createExpression (
    xml::string ("//author").c_str (),
    resolver.get ()));

dom::auto_ptr<DOMXPathResult> r (
  expr->evaluate (
    doc, DOMXPathResult::ITERATOR_RESULT_TYPE, 0));

while (r->iterateNext ())
{
  DOMNode* n (r->getNodeValue ());

  author* a (
    static_cast<author*> (
      n->getUserData (dom::tree_node_key)));

  cout << "Name  : " << a->get_name () << endl;
}
```

- No universal support
- Boilerplate setup code
  - DOM, XML namespaces, Memory management
- Casting is inevitable
- Look and feel of two APIs is (vastly) different
  - `iterateNext()` Vs. `begin()/end()`
- Can’t use predicates on data outside xml
  - E.g. Find authors of highest selling books

“/book[?condition?]/author/name”

What went wrong?

- Schema-specificity (to much object-oriented bias?)
  - Each class has a different interface (not generic)
  - Naming convention of XML data binding tools vary



- Lost succinctness (axis-oriented expressions)
- Lost structure-shyness (descendant axis, wildcards)
- Can’t use Visitor design pattern (stateful traversal) with XPath

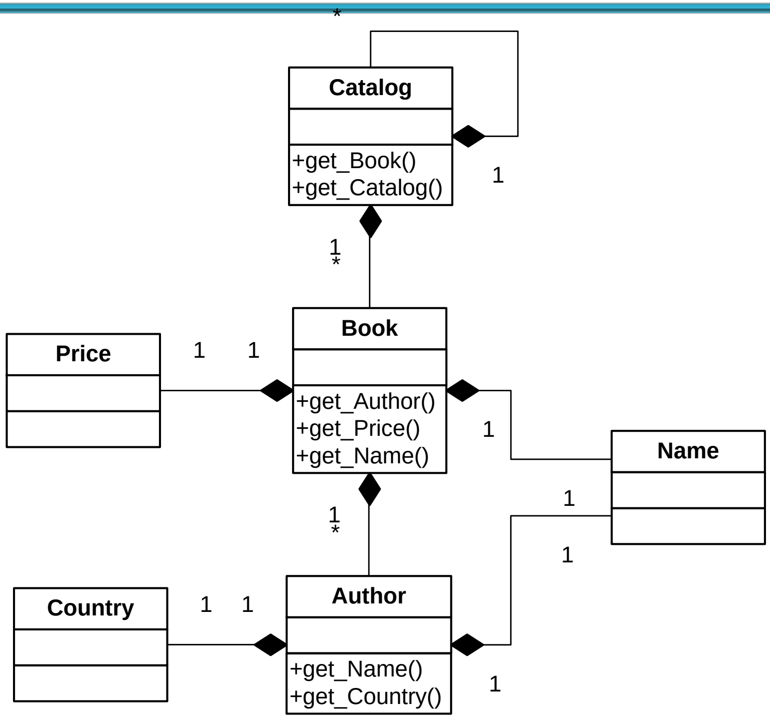
Combine
type-safety of data binding &
power of XPath?

Solution: LEESA

Language for Embedded QuEry and TraverSAl

Multi-paradigm Design in C++

LEESA by Examples




```
<catalog>
  <catalog>
    <catalog>
      <catalog>...</catalog>
    </catalog>
    <book>
      <name>...</name>
      <price>...</price>
      <author>
        <name>...</name>
        <country>...</country>
      </author>
    </book>
  </catalog>
</catalog>
```

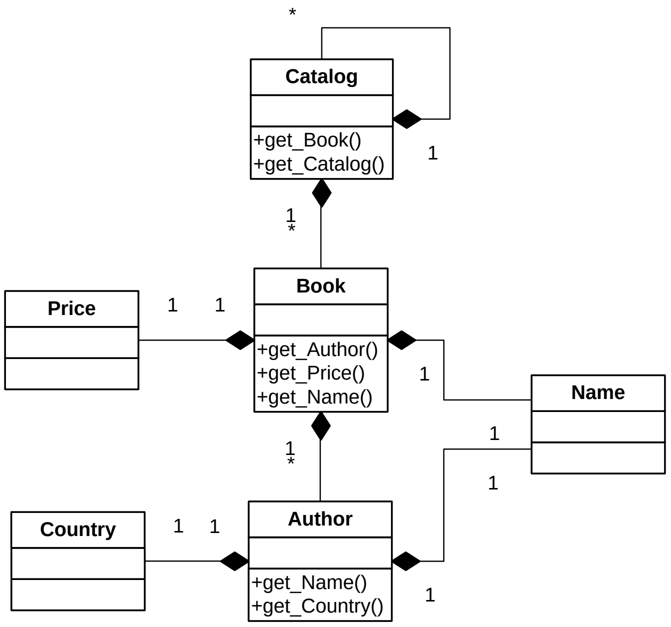
- A book catalog xsd
- Generated six C++ classes
  - Catalog
  - Book
  - Author
  - Price
  - Country
  - Name

  Complex classes (Catalog, Book, Author)

  Simple classes (Price, Country, Name)
- Price, Country, and Name are simple wrappers
- Catalogs are recursive

LEESA by Examples

- Restoring succinctness
  - Example: Find all author names
  - Child axis traversal



Using XPath (1 line)

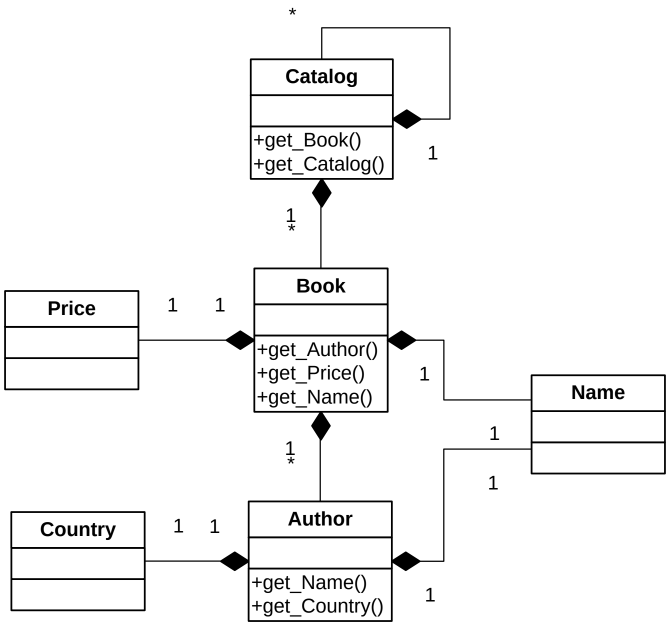
```
/book/author/name/text()
```

Using LEESA (3 lines)

```
Catalog croot = load_catalog(“catalog.xml”);
std::vector<Name> author_names =
evaluate(croot, Catalog() >> Book() >> Author() >> Name());
```

LEESA by Examples

- Restoring expressive power
  - Example: Find all names recursively
  - Descendant axis traversal



Using XPath (1 line)

```
//name/text()
```

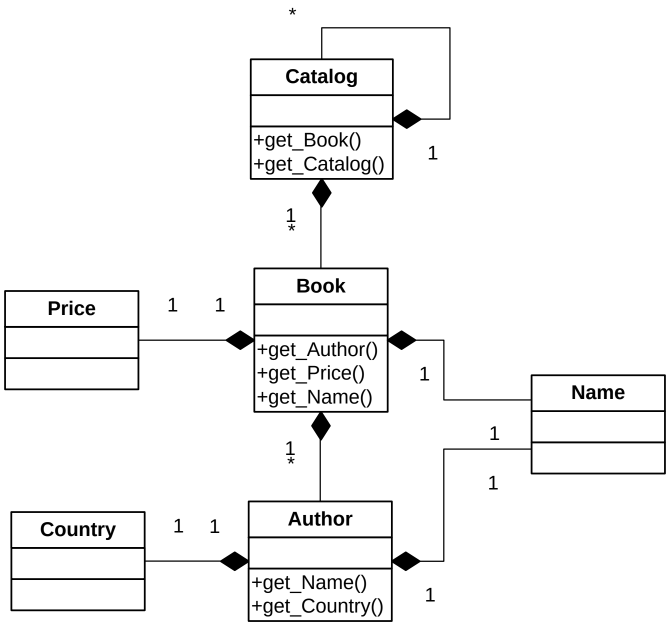
Using LEESA (2 lines)

```
Catalog croot = load_catalog("catalog.xml");
std::vector<Name> names = DescendantsOf(Catalog(), Name())(croot);
```

- Fully statically typed execution
- Efficient: LEESA "knows" where Names are!

LEESA by Examples

- Restoring xml programming idioms (structure-shyness)
  - Example: Without spelling intermediate types, find names that are exactly at the third level.
  - Wildcards in a typed query!



Using XPath (1 line)

```
/*/*/name/text()
```

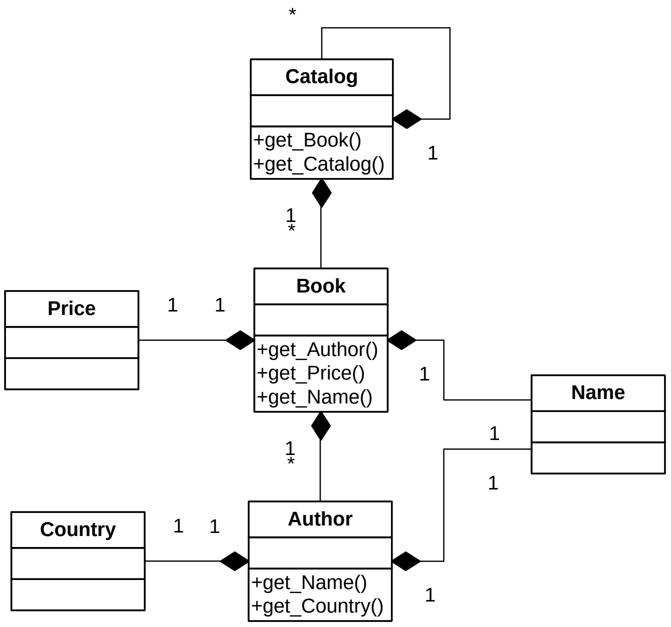
Using LEESA (3 lines)

```
namespace LEESA { struct Underbar {} _; }
Catalog croot = load_catalog("catalog.xml");
std::vector<Name> names =
    LevelDescendantsOf(Catalog(), _, _, Name())(croot);
```

- Fully statically typed execution
- Efficient: LEESA "knows" where Books, Authors, and Names are!

LEESA by Examples

- User-defined filters
  - Example: Find names of authors from Country == USA
  - Basically unary functors
  - Supports free functions, function objects, boost::bind, C++0x lambda



Using XPath (1 line)

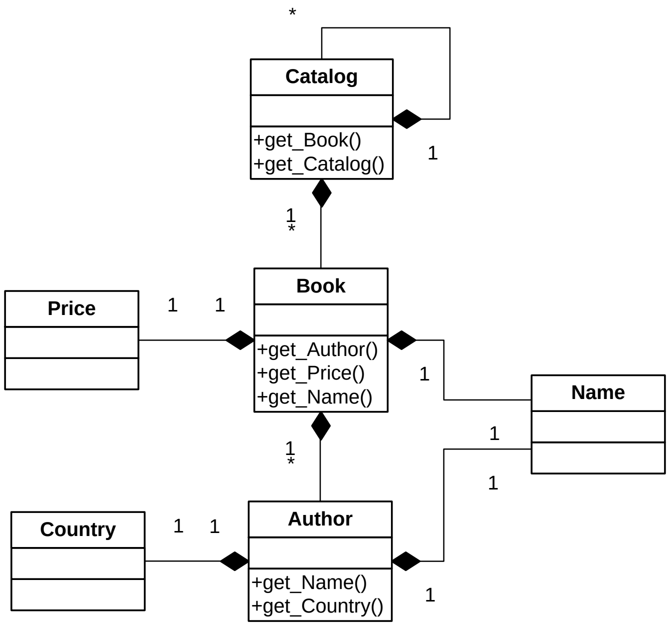
```
//author[country/text() = ‘USA’]/name/text()
```

Using LEESA (6 lines)

```
Catalog croot = load_catalog(“catalog.xml”);
std::vector<Name> author_names = evaluate(croot,
      Catalog()
   >> DescendantsOf(Catalog(), Author())
   >> Select(Author(), [](const Author &a) { return a.get_Country() == “USA"; })
   >> Name());
```

LEESA by Examples (beyond XPath)

- Tuplefication!!
  - Example: Pair the name and country of all the authors
  - `std::vector of boost::tuple<Name *, Country *>`



Using XPath

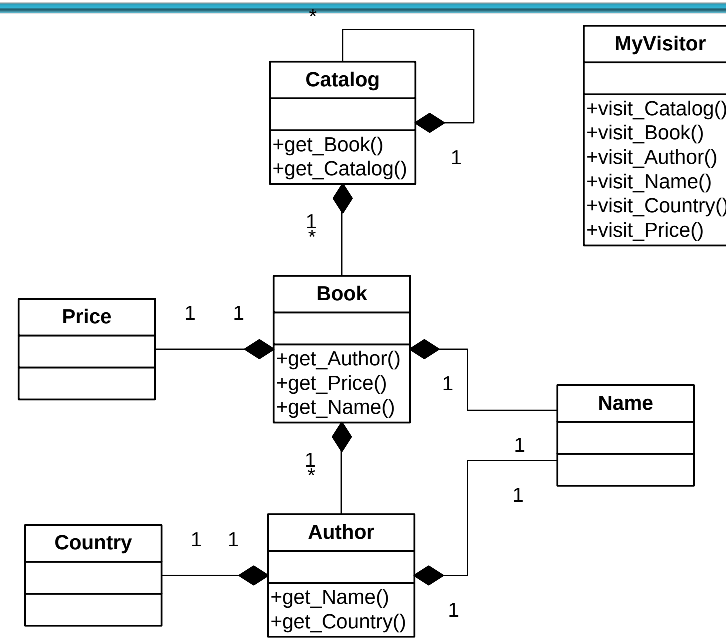
```
???????????????????????????????
```

Using LEESA (5 lines)

```
Catalog croot = load_catalog("catalog.xml");
std::vector<boost::tuple<Name *, Country *> > tuples =
evaluate(croot, Catalog()
              >> DescendantsOf(Catalog(), Author())
              >> MembersAsTupleOf(Author(), make_tuple(Name(), Country())));
```

LEESA by Examples (beyond XPath)

- Using visitors
  - Gang-of-four Visitor design pattern
  - `Visit` methods for all Elements
  - Example: Visit catalog, books, authors, and names in that order
  - Stateful, statically typed traversal
  - fixed depth child axis

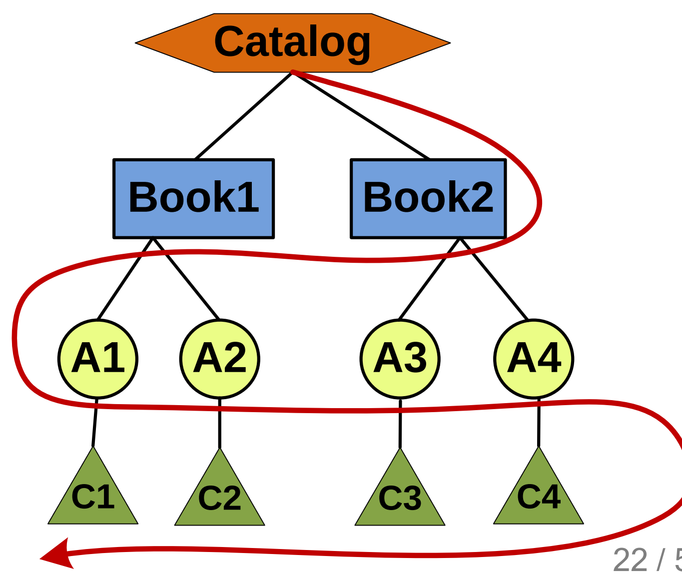


Using XPath

```
????????????????????????????????
```

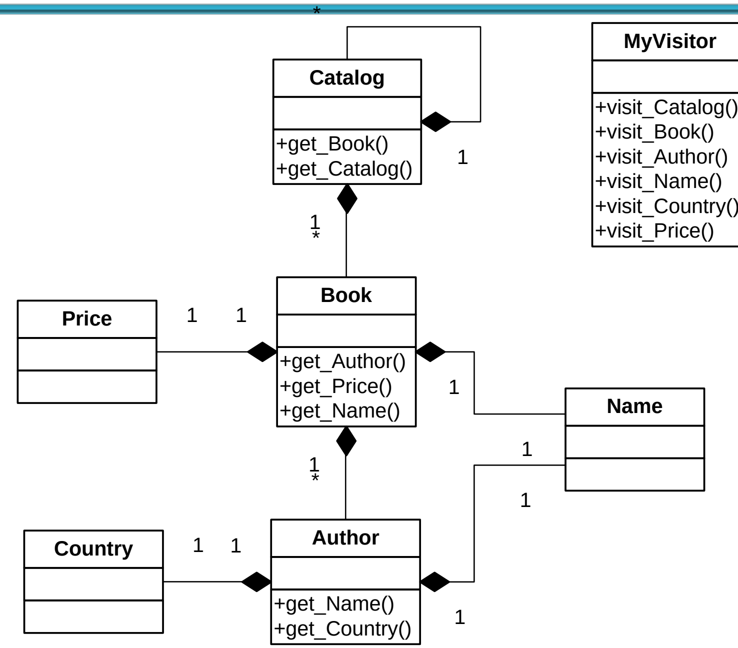
Using LEESA (7 lines)

```
Catalog croot = load_catalog("catalog.xml");
MyVisitor visitor;
std::vector<Country> countries =
evaluate(croot,     Catalog() >>  visitor
                 >> Book()    >>  visitor
                 >> Author()  >>  visitor
                 >> Country() >>  visitor);
```



LEESA by Examples (beyond XPath)

- Using visitors (depth-first)
  - Gang-of-four Visitor design pattern
  - Visit methods for all Elements
  - Example: Visit catalog, books, authors, and names in depth-first manner
  - Stateful, statically typed traversal
  - fixed depth child axis



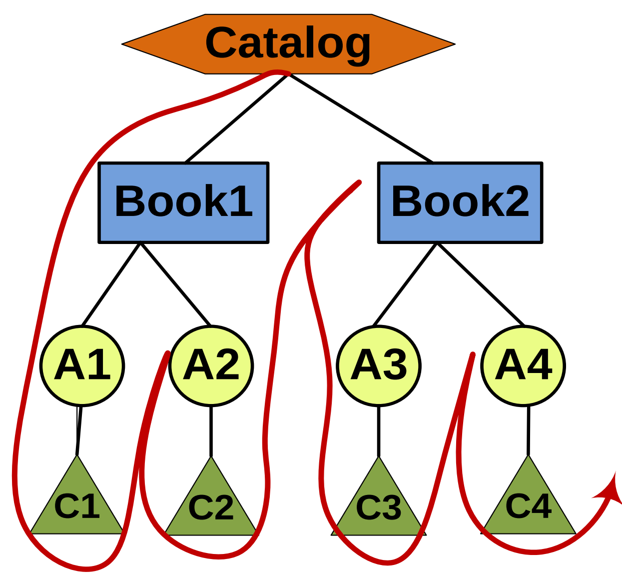
Using XPath

```
????????????????????????????????
```

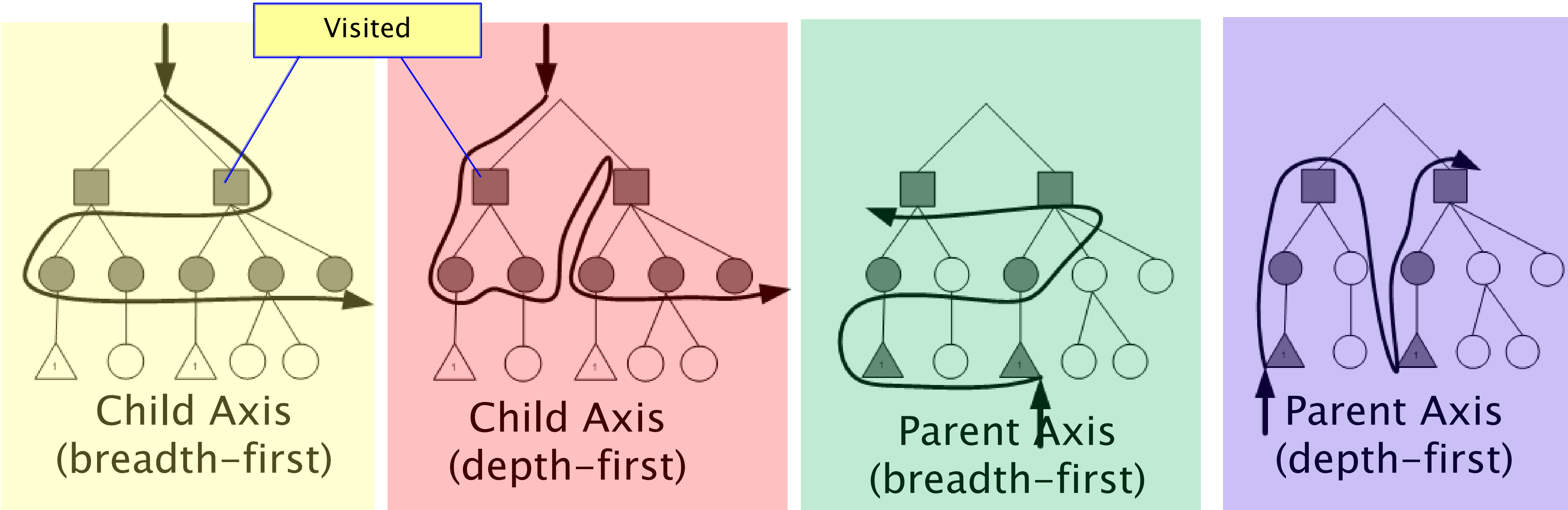
Default precedence.
No parenthesis needed.

Using LEESA (7 lines)

```
Catalog croot = load_catalog("catalog.xml");
MyVisitor visitor;
std::vector<Book> books =
evaluate(croot,      Catalog() >>  visitor
                >>= Book()    >>  visitor
                >>= Author()  >>  visitor
                >>= Country() >>  visitor);
```



LEESA Axis-oriented Expressions




```
Catalog() >> Book() >> v >> Author() >> v
```

```
Catalog() >>= Book() >> v >>= Author() >> v
```

```
Name() << v << Author() << v << Book() << v
```

```
Name() << v <<= Author() << v <<= Book() << v
```

Default precedence. No parenthesis needed.

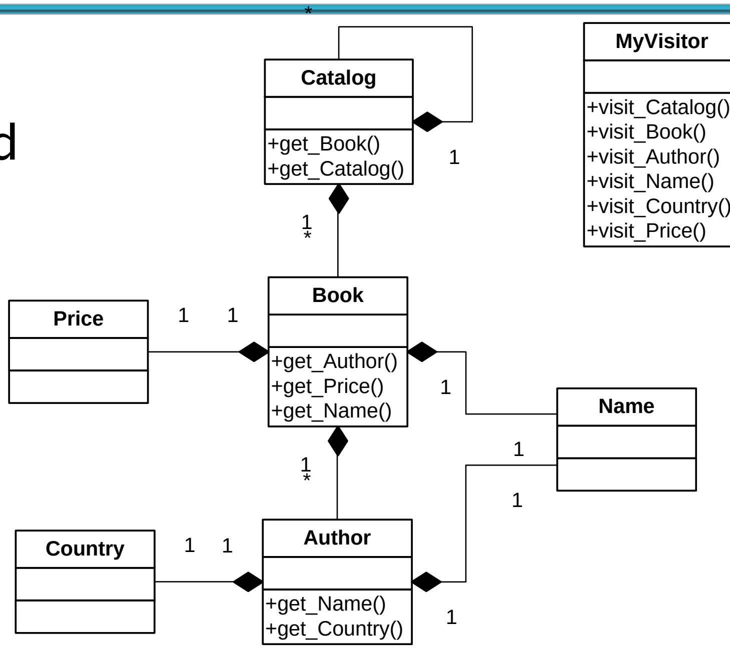
LEESA by Examples (beyond XPath)

- Composing named queries
  - Queries can be named, composed, and passed around as executable expressions
  - Example:

    For each book

    print(country of the author)

    print(price of the book)



Using XPath

```
????????????????????????????????
```

Using LEESA (6 lines)

```
Catalog croot = load_catalog("catalog.xml");
MyVisitor visitor;
BOOST_AUTO(v_country, Author() >> Country() >> visitor);
BOOST_AUTO(v_price,   Price()  >> visitor);
BOOST_AUTO(members, MembersOf(Book(), v_country, v_price));
evaluate(croot, Catalog() >>= Book() >> members);
```

LEESA by Examples (beyond XPath)

- Using visitors (recursively)
  - Hierarchical Visitor design pattern
  - `Visit` and `Leave` methods for all elements
  - Depth awareness
  - Example: Visit everything!!
  - Stateful, statically typed traversal
  - Descendant axis = recursive
  - `AroundFullTD` = AroundFullTopDown

MyHierarchicalVisitor
+visit_Catalog() +visit_Book() +visit_Author() +visit_Name() +visit_Country() +visit_Price() +leave_Catalog() +leave_Book() +leave_Author() +leave_Name() +leave_Country() +leave_price()

Using XPath

```
???????????????????????????????
```

Using LEESA (3 lines!!)

```
Catalog croot = load_catalog(“catalog.xml”);
MyHierarchicalVisitor v;
AroundFullTD(Catalog(), VisitStrategy(v), LeaveStrategy(v)))(croot);
```

What LEESA is and what it is not

- LEESA
  1. Is not an xml parsing library
  2. Does not validate xml files

  XML data binding tool can do both

  3. Does not replace/compete with XPath
  4. Does not resolve X/O impedance mismatch
     - More reading: "Revealing X/O impedance mismatch", Dr. R Lämmel

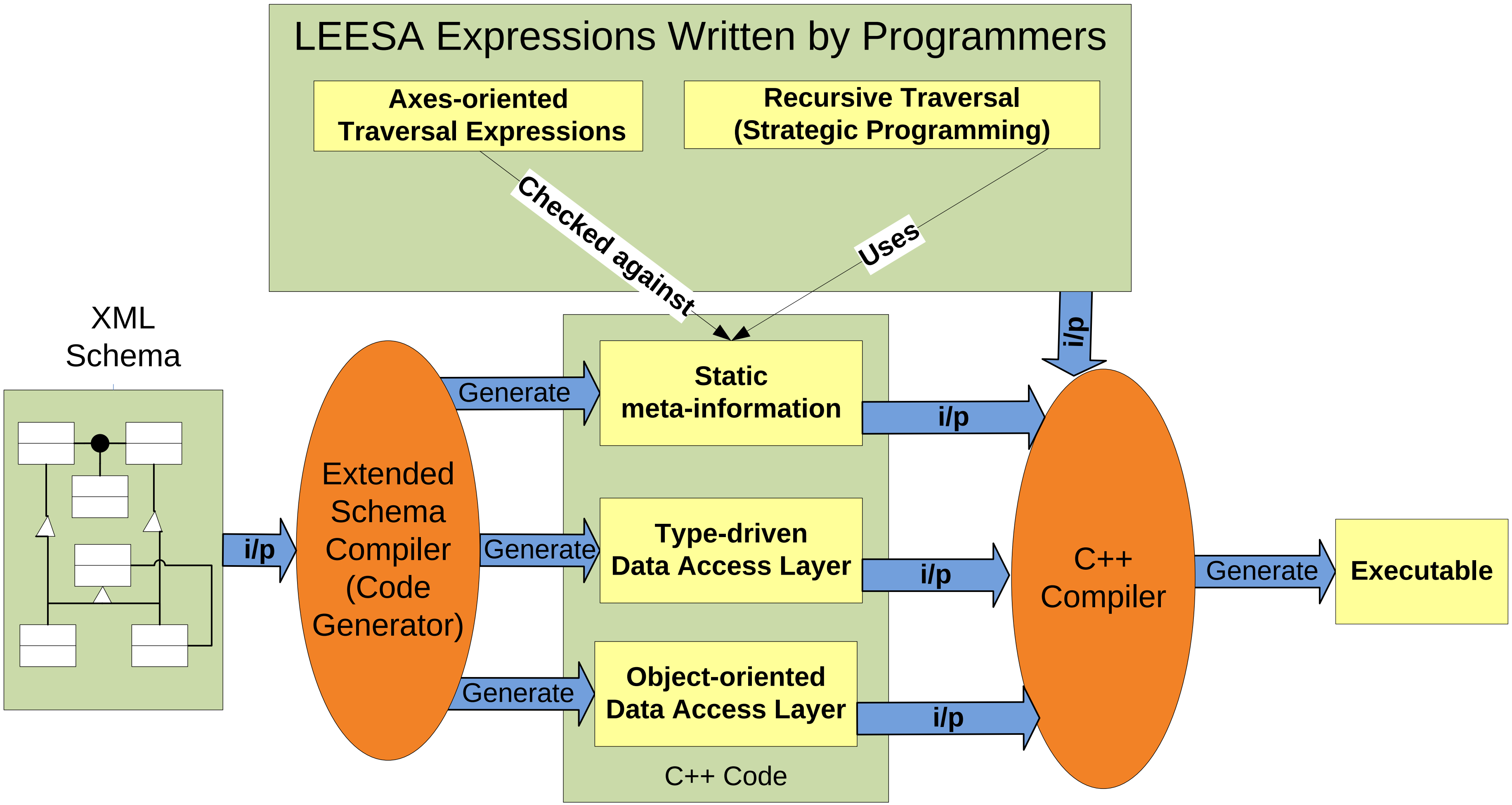
- LEESA
  1. Is a query and traversal library for C++
  2. Validates XPath-like queries at compile-time (schema conformance)
  3. Is motivated by XPath
  4. Goes beyond XPath
  5. Simplifies typed XML programming
  6. Is an embedded DSEL (Domain-specific embedded language)
  7. Is applicable beyond xml (E.g., Google Protocol Buffers, model traversal, hand coded class hierarchies, etc.)

Outline

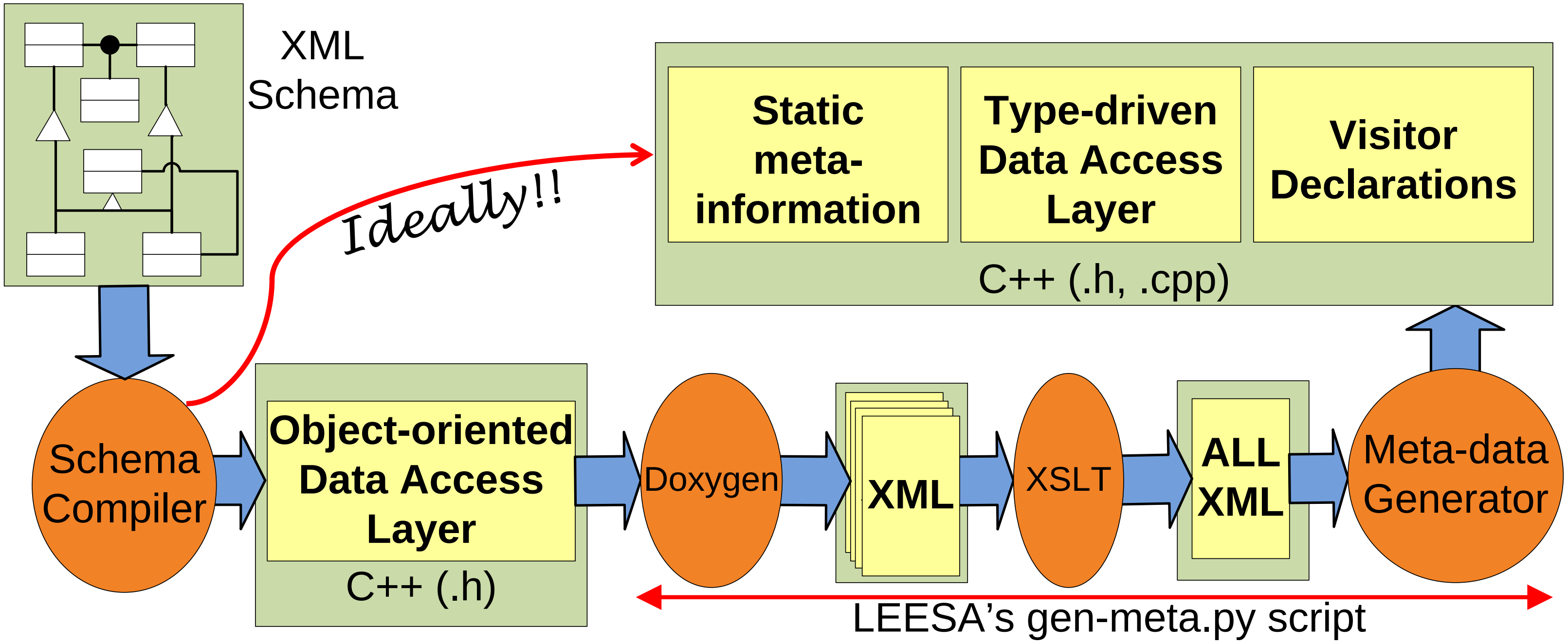
- XML Programming in C++, specifically data-binding
- What XML data binding stole from us!
- Restoring order: LEESA
- LEESA by examples
- LEESA in detail
  - Architecture of LEESA
  - Type-driven data access
  - XML schema representation using Boost.MPL
  - LEESA descendant axis and strategic programming
  - Compile-time schema conformance checking
  - LEESA expression templates
- Evaluation: productivity, performance, compilers
- C++0x and LEESA
- LEESA in future

Architecture of LEESA

- The Process



Extended Schema Compiler

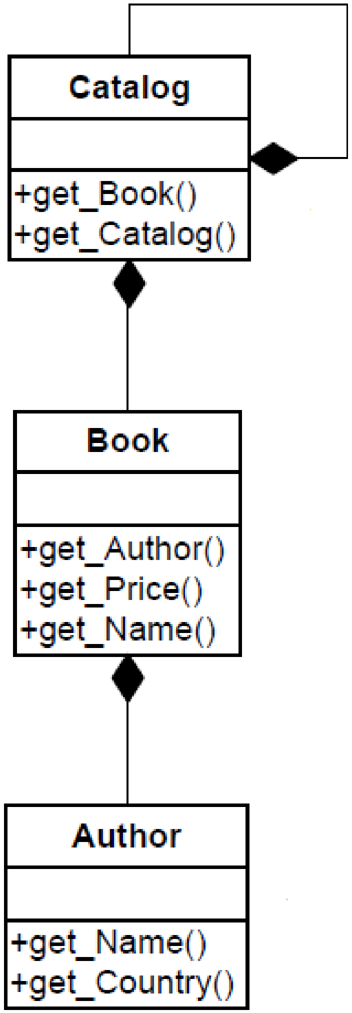


- Extended schema compiler = 4 step process
  - XML schema language (XSD) specification is huge and complex
  - Don't reinvent the wheel: xml data binding tools already process it
  - Naming convention of xml data binding tools vary
  - Applicability beyond xml data binding
    - E.g. Google Protocol Buffers (GPB), hand written class hierarchies
  - Meta-data generator script inserts visitor declaration in the C++ classes

Type-driven Data Access Layer

- To fix → Different interface of each class
- Generic API "children" wrappers to navigate aggregation
- Generated by the Python script
- More amenable to composition

```
std::vector<Book> children (Catalog &c, Book const *) {
  return c.get_Book();
}
std::vector<Catalog> children (Catalog &c, Catalog const *) {
  return c.get_Catalog();
}
std::vector<Author> children (Book &b, Author const *) {
  return b.get_Author();
}
Price children (Book &b, Price const *) {
  return b.get_Price();
}
Name children (Book &b, Name const *) {
  return b.get_Name();
}
Country children (Author  &a, Country const *) {
  return a.get_Country();
}
Name children (Author  &a, Name const *) {
  return a.get_Name();
}
```



Type-driven Data Access Layer

- Ambiguity!
  - Simple elements and attributes are mapped to built-in types
  - "children" function overloads become ambiguous

```
<xs:complexType name="Author">
  <xs:sequence>
    <xs:element   name="first_name" type="xs:string" />
    <xs:element   name="last_name"  type="xs:string" />
  </xs:sequence>
</xs:complexType>
```

Mapping

Author

-first_name : string
-last_name : string

+get_first_name()
+get_last_name()

gen-meta.py

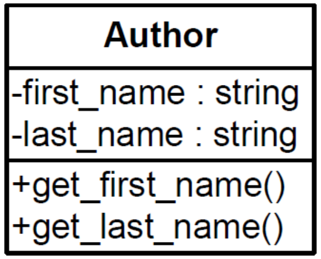

```
std::string children (Author  &a, std::string const *) {
  return a.get_first_name();
}
std::string children (Author  &a, std::string const *) {
  return a.get_last_name();
}
```

Type-driven Data Access Layer

- Solution 1: Automatic schema transformation
  - Force data binding tools to generate unique C++ types
  - gen-meta.py can transforms input xsd while preserving semantics

```
<xs:complexType name="Author">
  <xs:sequence>
    <xs:element   name="first_name" type="xs:string" />
    <xs:element   name="last_name"  type="xs:string" />
  </xs:sequence>
</xs:complexType>
```

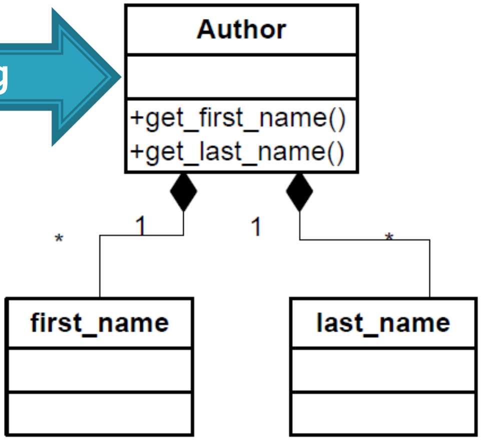
Mapping



Transformation (gen-meta.py)

```
<xs:complexType name="Author">
 <xs:sequence>
  <xsd:element name="first_name">
   <xsd:simpleType>
     <xsd:restriction base="xsd:string" />
   </xsd:simpleType>
  </xsd:element>
  <xsd:element name="last_name">
   <xsd:simpleType>
     <xsd:restriction base="xsd:string" />
   </xsd:simpleType>
  </xsd:element>
 </xs:sequence>
</xs:complexType>
```

Mapping



Type-driven Data Access Layer

- **Solution 1 limitations:** Too many types! Longer compilation times.
- **Solution 2: Generate placeholder types**
  - Create unique type aliases using a template and integer literals
  - Not implemented!

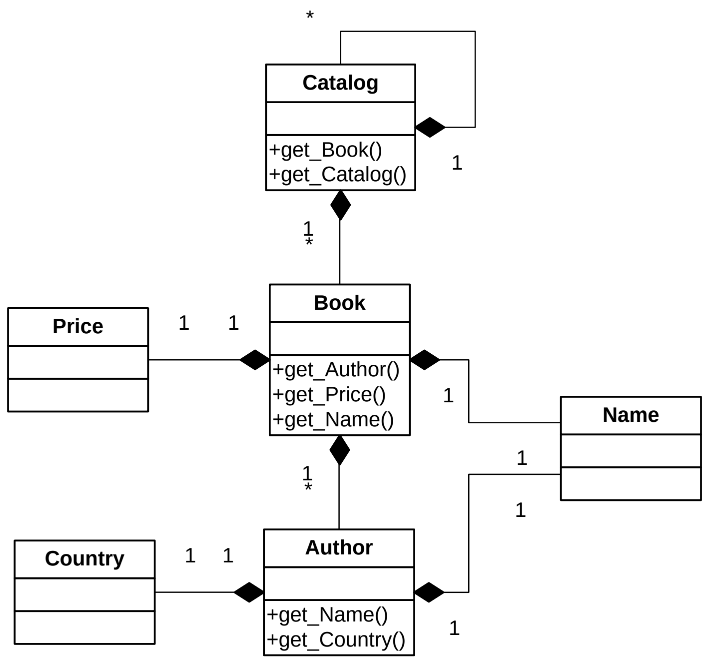
```
<xs:complexType name=“Author”>
  <xs:sequence>
    <xs:element   name=“first_name" type="xs:string" />
    <xs:element   name=“last_name“  type="xs:string" />
  </xs:sequence>
</xs:complexType>
```

Code generation
(gen-meta.py)

```
namespace LEESA {
  template <class T, unsigned int I>
  struct unique_type
  {
    typedef T nested;
  };
}
namespace Library {
  typedef LEESA::unique_type<std::string, 1> first_name;
  typedef LEESA::unique_type<std::string, 2> last_name;
}
```

Schema Representation using Boost.MPL

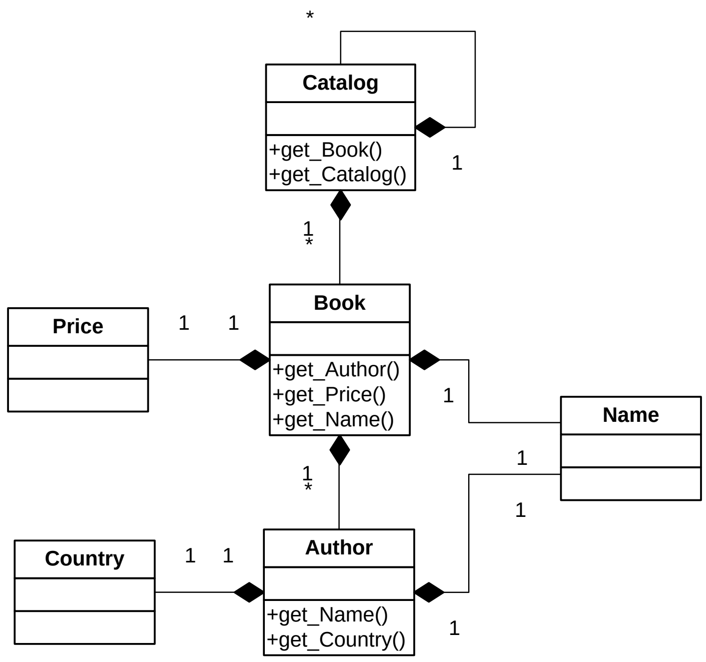
- A key idea in LEESA
  - Externalize structural meta-information using Boost.MPL
  - LEESA's meta-programs traverse the meta-information at compile-time




```
template <class Kind>
struct SchemaTraits
{
  typedef mpl::vector<> Children; // Empty sequence
};

template <>
struct SchemaTraits <Catalog>
{
  typedef mpl::vector<Book, Catalog> Children;
};
template <>
struct SchemaTraits <Book>
{
  typedef mpl::vector<Name, Price, Author> Children;
};
template <>
struct SchemaTraits <Author>
{
  typedef mpl::vector<Name, Country> Children;
};
```

Schema Representation using Boost.MPL

- A key idea in LEESA
  - Externalize structural meta-information using Boost.MPL
  - Descendant meta-information is a transitive closure of `Children`



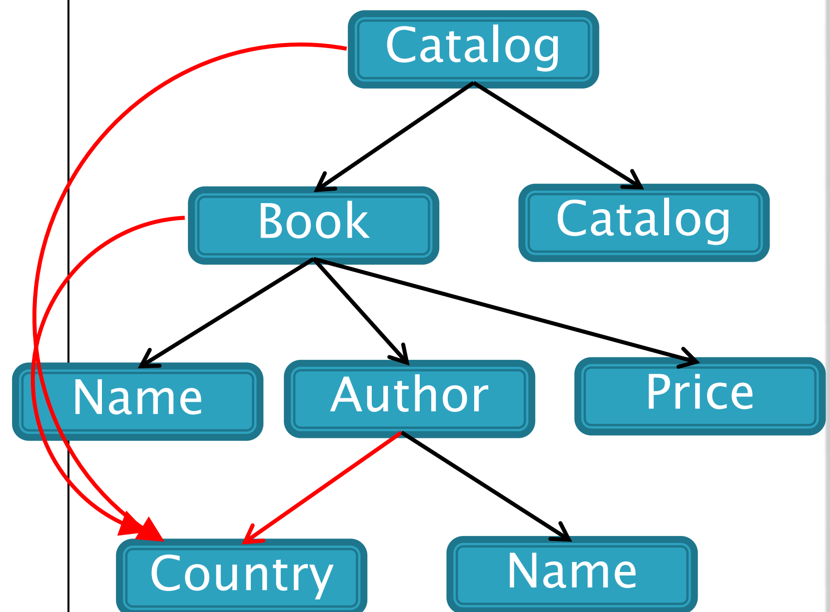

```
template <class Kind> struct SchemaTraits {
  typedef mpl::vector<> Children; // Empty sequence
};
template <> struct SchemaTraits <Catalog> {
  typedef mpl::vector<Book, Catalog> Children;
};
template <> struct SchemaTraits <Book> {
  typedef mpl::vector<Name, Price, Author> Children;
};
template <> struct SchemaTraits <Author> {
  typedef mpl::vector<Name, Country> Children;
};
typedef boost::mpl::true_  True;
typedef boost::mpl::false_ False;
template<class A, class D> struct IsDescendant  : False {};
template<> struct IsDescendant<Catalog, Catalog> : True {};
template<> struct IsDescendant<Catalog, Book>    : True {};
template<> struct IsDescendant<Catalog, Name>    : True {};
template<> struct IsDescendant<Catalog, Price>   : True {};
template<> struct IsDescendant<Catalog, Author>  : True {};
template<> struct IsDescendant<Catalog, Country> : True {};
template<> struct IsDescendant<Book, Name>       : True {};
template<> struct IsDescendant<Book, Price>      : True {};
template<> struct IsDescendant<Book, Author>     : True {};
template<> struct IsDescendant<Book, Country>    : True {};
template<> struct IsDescendant<Author, Name>     : True {};
template<> struct IsDescendant<Author, Country>  : True {};
```

Locating Descendants: Meta-programming

```
std::vector<Country> countries = DescendantsOf(Catalog(), Country())(croot);
```

- Algorithm (conceptual)

1. IsDescendant<Catalog, Country>::value
2. Find all children types of Catalog
   SchemaTraits<Catalog>::Children =
   boost::mpl::vector<Book, Catalog>
3. Iterate over Boost.MPL vector
4. IsDescendant<Book, Country>::value
5. Use type-driven data access on each Catalog
   std::vector<Book>=children(Catalog&, Book*)
   For Catalogs repeat step (1)
6. Find all children types of Book
   SchemaTraits<Book>::Children =
   boost::mpl::vector<Name, Author, Price>
7. Iterate over Boost.MPL vector
8. IsDescendant<Name, Country>::value
9. IsDescendant<Price, Country>::value
10. IsDescendant<Author, Country>::value
11. Use type drive data access on each Book
    std::vector<Author>=children(Book&, Author*)
12. Find all children types of Author
    SchemaTraits<Author>::Children =
    boost::mpl::vector<Country, Name>
13. Repeat until Country objects are found



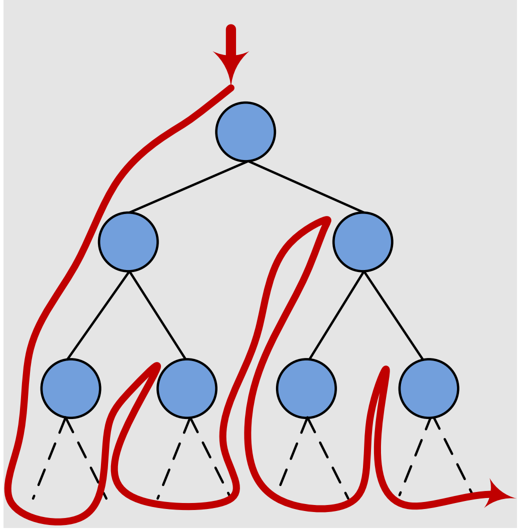
Adopting Strategic Programming (SP)

- Strategic Programming Paradigm
  - A systematic way of creating recursive tree traversal
  - Developed in 1998 as a term rewriting language: Stratego
- Why LEESA uses strategic programming
  - **Generic**
    - LEESA can be designed without knowing the types in a xml tree
  - **Recursive**
    - LEESA can handles mutually and/or self recursive types
  - **Reusable**
    - LEESA can be reused as a library for any xsd
  - **Composable**
    - LEESA can be extended by its users using policy-based templates
- Basic combinators
  - Identity, Fail, Sequence, Choice, All, and One

Strategic Programming (very) Simplified

```
fullTD(node)
{
 visit(node);
 forall children c of node
      fullTD(c);
}
```

Pre-order traversal pseudo-code (fullTopDown)



```
fullTD(node)
{
 visit(node);
 All(node, fullTD);
}
```

```
All(node, strategy)
{
 forall children c of node
   strategy(c);
}
```

```
fullTD(node)
{
 seq(node, visit, All(fullTD));
}
```

Recursive traversal (1 out of many)

```
seq(node,strategy1,strategy2)
{
  strategy1(node);
  strategy2(node);
}
```

```
All(node, strategy)
{
 forall children c of node
   strategy(c);
}
```

Basic Combinators (2 out of 6)

Strategic Programming in LEESA

```
template <class Strategy1,
          class Strategy2>
class Seq
{
  template <class Data>
  void operator()(Data d)
  {
    Strategy1(d);
    Strategy2(d);
  }
};
```

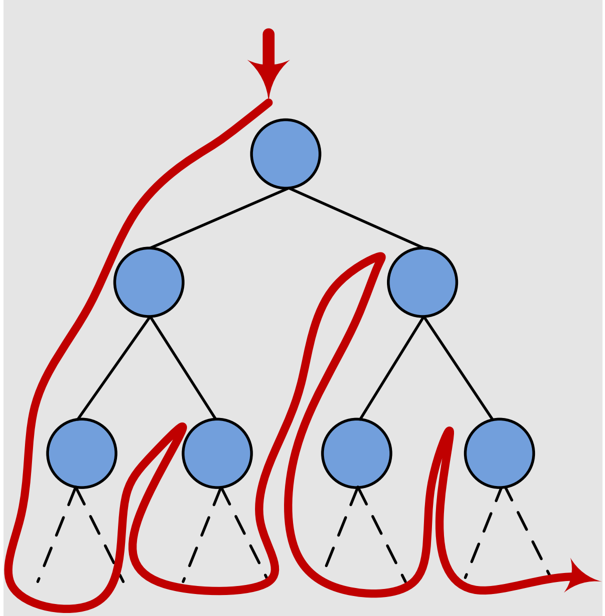
```
template <class Strategy>
class All
{
  template <class Data>
  void operator()(Data d)
  {
    foreach T in SchemaTraits<Data>::Children
       std::vector<T> t = children(d, (T *)0);
       Strategy(t);
  }
};
```

Boost.MPL Meta-information

Type-driven Data Access

Sequence + All = FullTD

```
template <class Strategy>
class FullTD
{
  template <class data>
  void operator()(Data d)
  {
    Seq<Strategy,All<FullTD>>(d);
  }
};
```

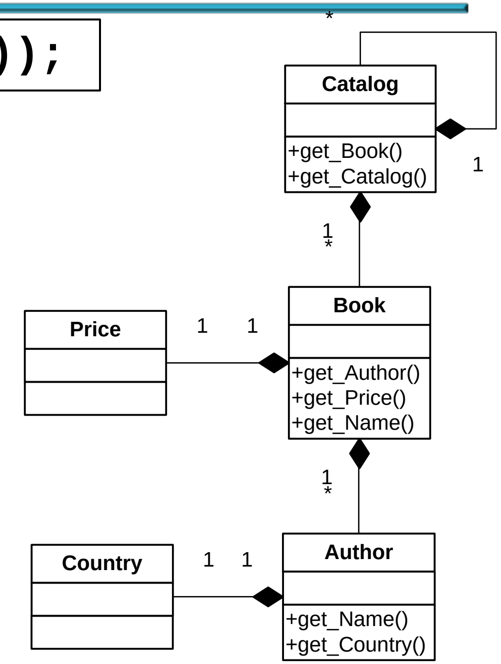


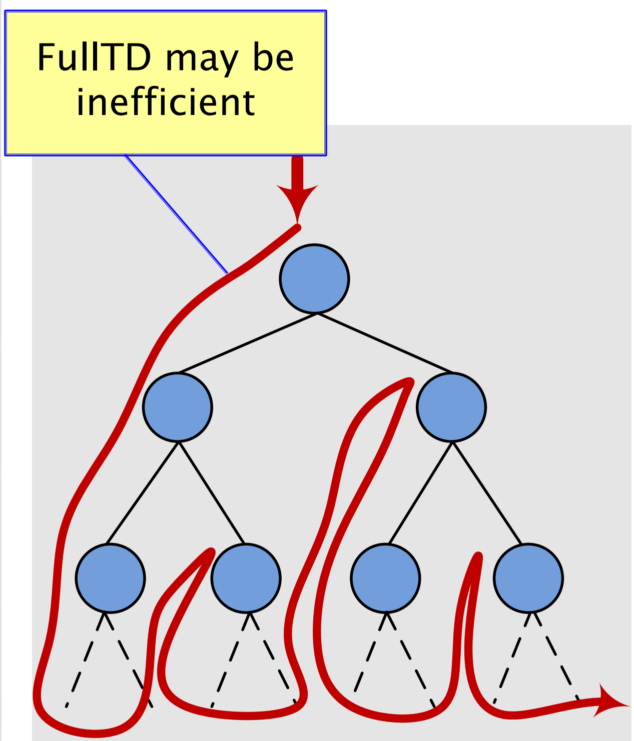
Note: Objects and constructors omitted for brevity

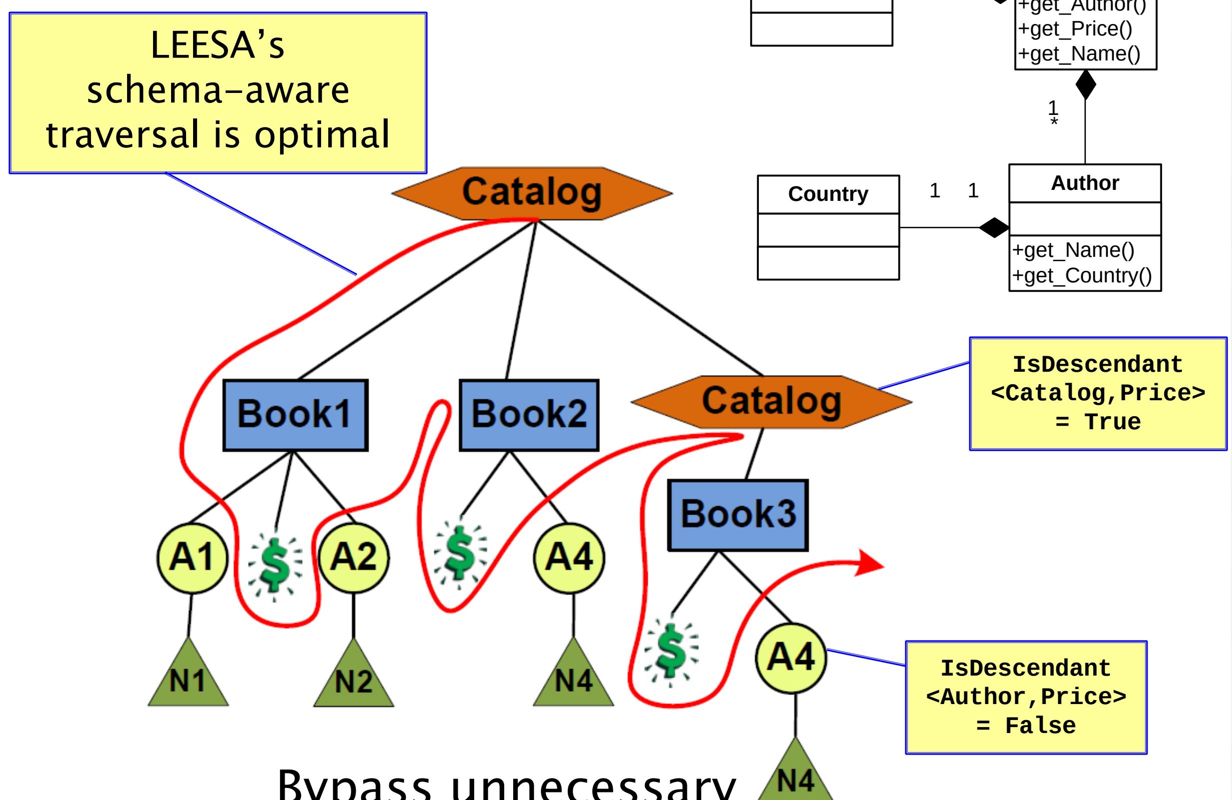
Schema-aware Descendant Axis Traversal

```
BOOST_AUTO(prices, DescendantsOf(Catalog(), Price()));
```

- LEESA uses FullTopDown<Accumulator<Price>>
- But schema unaware recursion in every sub-structure is inefficient
- We know that Authors do not contain Prices







Bypass unnecessary sub-structures (Author) using meta-programming

Compile-time Schema Conformance Checking

- LEESA has compile-time schema conformance checking
  - LEESA queries compile only if they agree with the schema
  - Uses externalized schema and meta-programming
  - Error message using BOOST_MPL_ASSERT
  - Tries to reduce long and incomprehensible error messages
  - Shows assertion failures in terms of concepts
    - `ParentChildConcept, DescendantKindConcept, etc.`
    - Originally developed for C++0x concepts
  - Examples

DescendantKindConcept Failure

ParentChildConcept Failure

```
1. BOOST_AUTO(prices, DescendantsOf(Author(), Price()));

2. BOOST_AUTO(books, Catalog() >> Book() >> Book());

3. BOOST_AUTO(countries, LevelDescendantsOf(Catalog(),_,Country()));
```

LevelDescendantKindConcept Failure

Compile-time Schema Conformance Checking

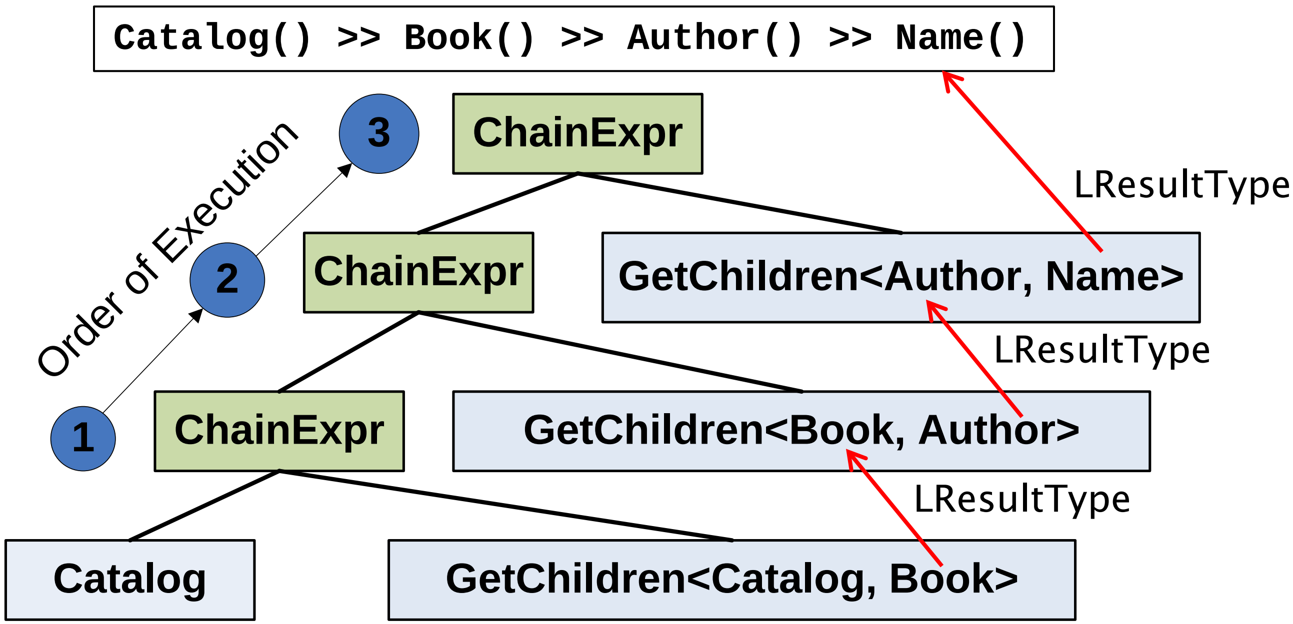
- `Country` is at least 2 “steps” away from a `Catalog`

```
LevelDescendantsOf(Catalog(),_,Country());
```

```
1>------ Build started: Project: library, Configuration: Release Win32 ------
1>  driver.cxx
1>  using native typeof
1>C:\mySVN\LEESA\include\LEESA/SP_Accumulation.cpp(112): error C2664: 'boost::mpl::assertion_failed' : cannot convert
parameter 1 from 'boost::mpl::failed
************LEESA::LevelDescendantKindConcept<ParentKind,DescendantKind,SkipCount,Custom>::* ***********' to
'boost::mpl::assert<false>::type'
1>          with
1>          [
1>              ParentKind=library::Catalog,
1>              DescendantKind=library::Country,
1>              SkipCount=1,
1>              Custom=LEESA::Default
1>          ]
1>          No constructor could take the source type, or constructor overload resolution was ambiguous
1>          driver.cxx(155) : see reference to class template instantiation
'LEESA::LevelDescendantsOp<Ancestor,Descendant,SkipCount,Custom>' being compiled
1>          with
1>          [
1>              Ancestor=LEESA::Carrier<library::Catalog>,
1>              Descendant=LEESA::Carrier<library::Country>,
1>              SkipCount=1,
1>              Custom=LEESA::Default
1>          ]
1>C:\mySVN\LEESA\include\LEESA/SP_Accumulation.cpp(112): error C2866:
'LEESA::LevelDescendantsOp<Ancestor,Descendant,SkipCount,Custom>::mpl_assertion_in_line_130' : a const static data member
of a managed type must be initialized at the point of declaration
1>          with
1>          [
1>              Ancestor=LEESA::Carrier<library::Catalog>,
1>              Descendant=LEESA::Carrier<library::Country>,
1>              SkipCount=1,
1>              Custom=LEESA::Default
1>          ]
1>  Generating Code...
========== Build: 0 succeeded, 1 failed, 0 up-to-date, 0 skipped ==========
```

LEESA Expression Templates

- (Nearly) all LEESA queries are expression templates
  - Hand rolled. Not using Boost.Proto

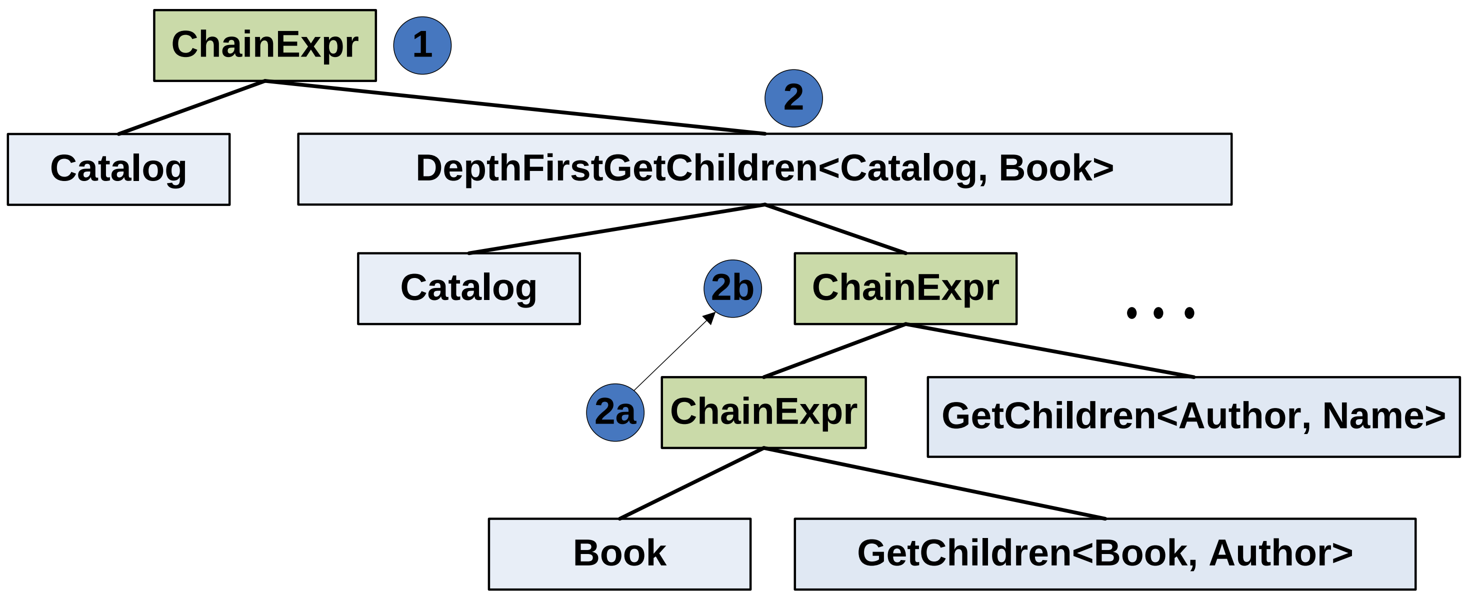



```
template <class L, class H>
ChainExpr<L, GetChildren<typename ExpressionTraits<L>::result_type, H> >
operator >> (L l, H h)
{
  typedef typename ExpressionTraits<L>::result_type LResultType;
  typedef GetChildren<LResultType, H> GC;
  return ChainExpr<L, GC>(l, h);
}
```

LEESA Expression Templates

- (Nearly) all LEESA queries are expression templates
  - Hand rolled. Not using Boost.Proto
  - Every LEESA expression becomes a unary function object
  - LEESA query → Systematically composed unary function objects

```
Catalog() >>= Book() >> Author() >> Name()
```



Outline

- XML Programming in C++, specifically data-binding
- What XML data binding stole from us!
- Restoring order: LEESA
- LEESA by examples
- LEESA in detail
  - Architecture of LEESA
  - Type-driven data access
  - XML schema representation using Boost.MPL
  - LEESA descendant axis and strategic programming
  - Compile-time schema conformance checking
  - LEESA expression templates
- Evaluation: productivity, performance, compilers
- C++0x and LEESA
- LEESA in future

Evaluation: Productivity

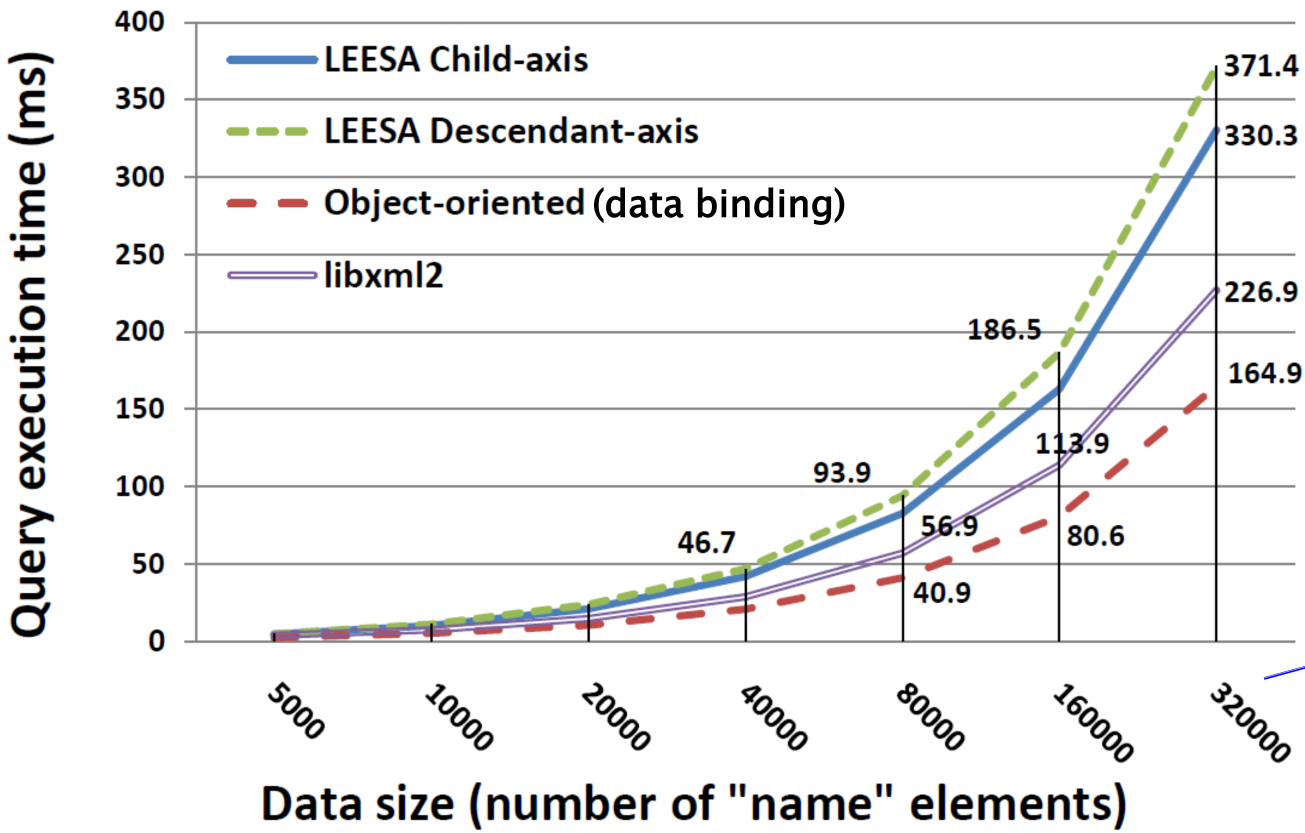
- Reduction in boilerplate traversal code
  - Results from the 2009 paper in the Working Conference on Domain-Specific Languages, Oxford, UK

Traversal Pattern	Axis	Occurrences	Original #lines (average)	#Lines using LEESA (average)
A single loop iterating over a list of objects	Child	11	8.45	1.45
	Association	6	7.50	1.33
5 sequential loops iterating over siblings	Sibling	3	41.33	6
2 Nested loops	Child	2	16	1
Traversal-only visit functions	Child	3	11	0
Leaf-node accumulation using depth-first	Descendant	2	43.5	4.5
Total traversal code	-	All	414 (absolute)	53 (absolute)

87% reduction in traversal code

Evaluation: Performance (run-time)

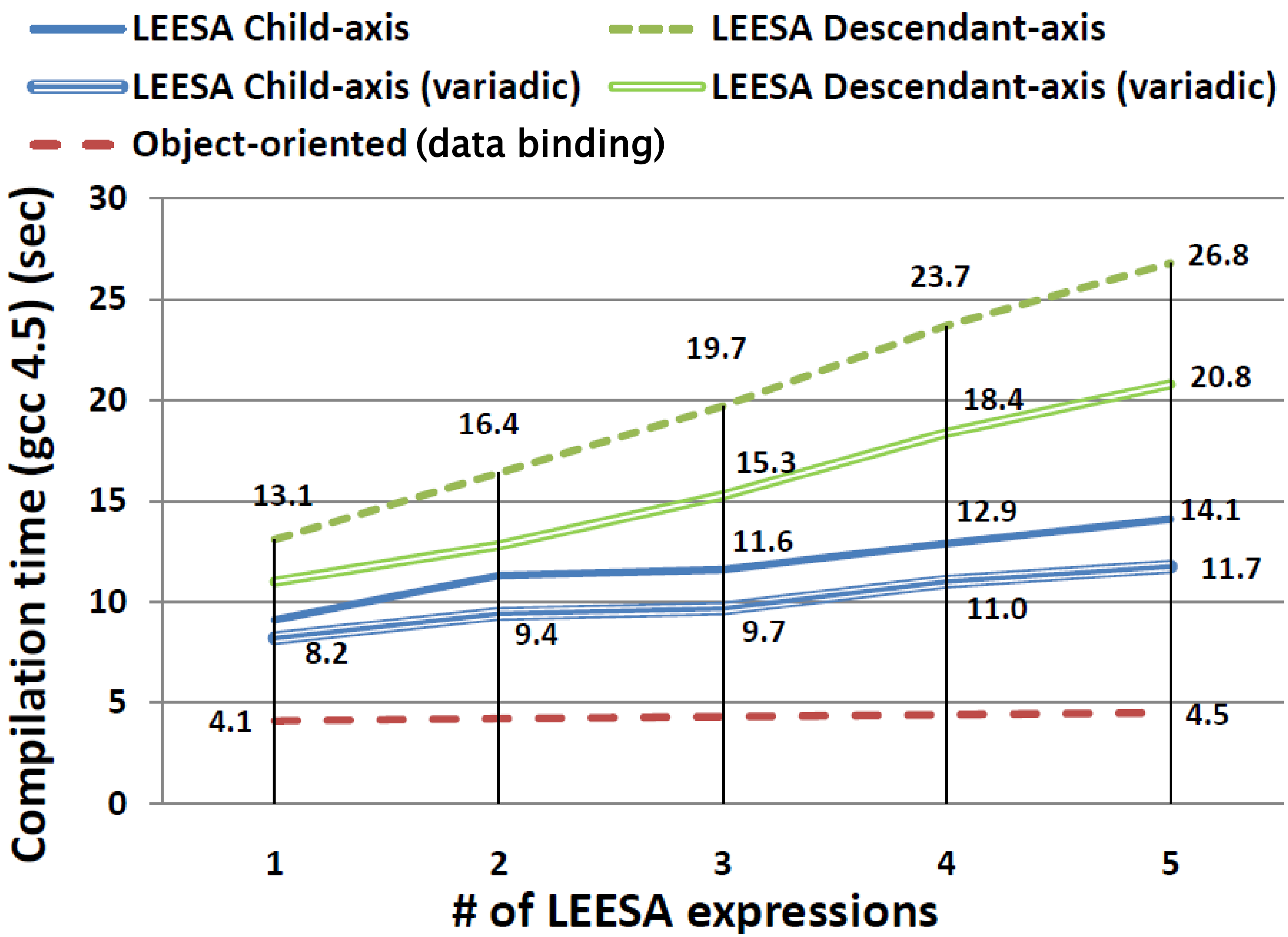
- CodeSynthesis xsd data binding tool on the catalog xsd
- Abstraction penalty from construction, copying, and destruction of internal containers (`std::vector<T>` and `LEESA::Carrier<T>`)
- GNU Profiler: Highest time spent in `std::vector<T>::insert` and iterator dereference functions



33 seconds for parsing, validating, and object model construction

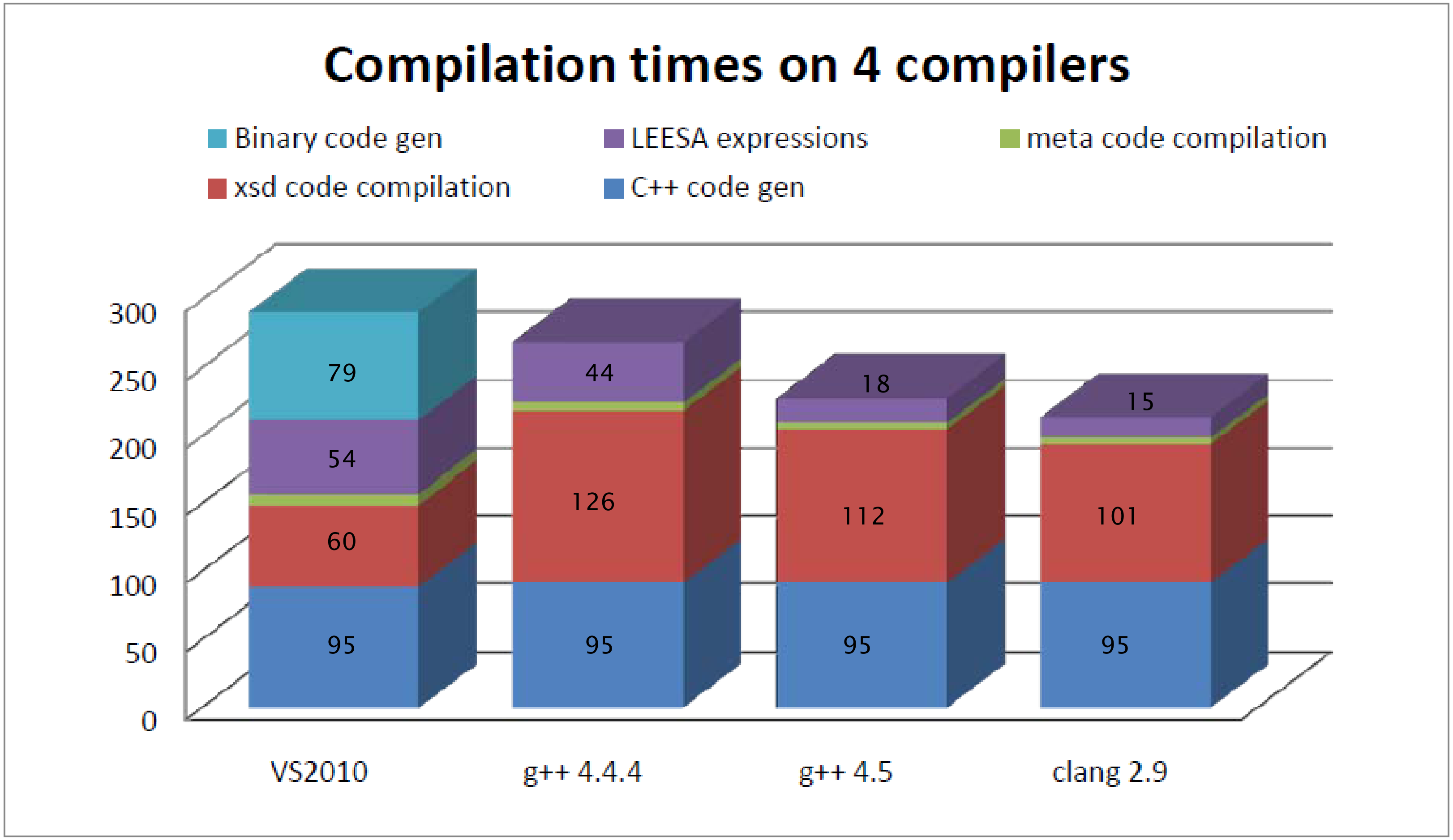
Evaluation: Performance (compile-time)

- Compilation time affects programmer productivity
- Experiment
  - An XML schema containing 300 types (4 recursive)
  - gcc 4.5 (with and without variadic templates)

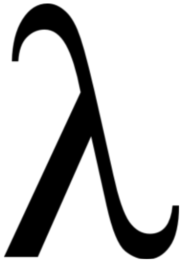


Evaluation: Performance (compile-time)

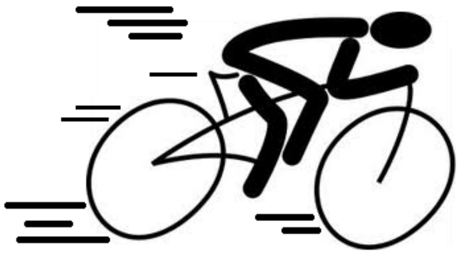
- Experiment: Total time to build an executable from an xsd on 4 compilers
  - XML schema containing 300 types (4 recursive)
  - 5 LEESA expressions (all using descendant axis)
  - Tested on Intel Core 2 Duo 2.67 GHz, 4 GB laptop



C++0x and LEESA



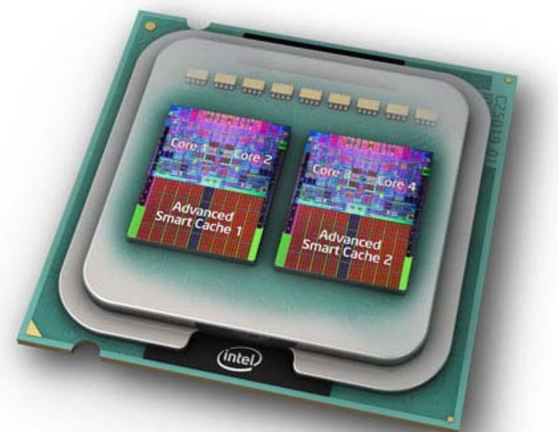
- Readability improvements
  - Lambdas!
  - LEESA actions (e.g., Select, Sort) can use C++0x lambdas
  - `static_assert` for improved error reporting
  - `auto` for naming LEESA expressions



- Performance improvements (run-time)
  - Rvalue references and move semantics
  - Optimize away internal copies of large containers
- Performance improvements (Compile-time)
  - Variadic templates → Faster schema conformance checking
  - No need to use BOOST_MPL_LIMIT_VECTOR_SIZE and Boost.Preprocessor tricks
- Simplifying LEESA's implementation
  - Trailing return-type syntax and `decltype`
  - Right angle bracket syntax

LEESA in Future

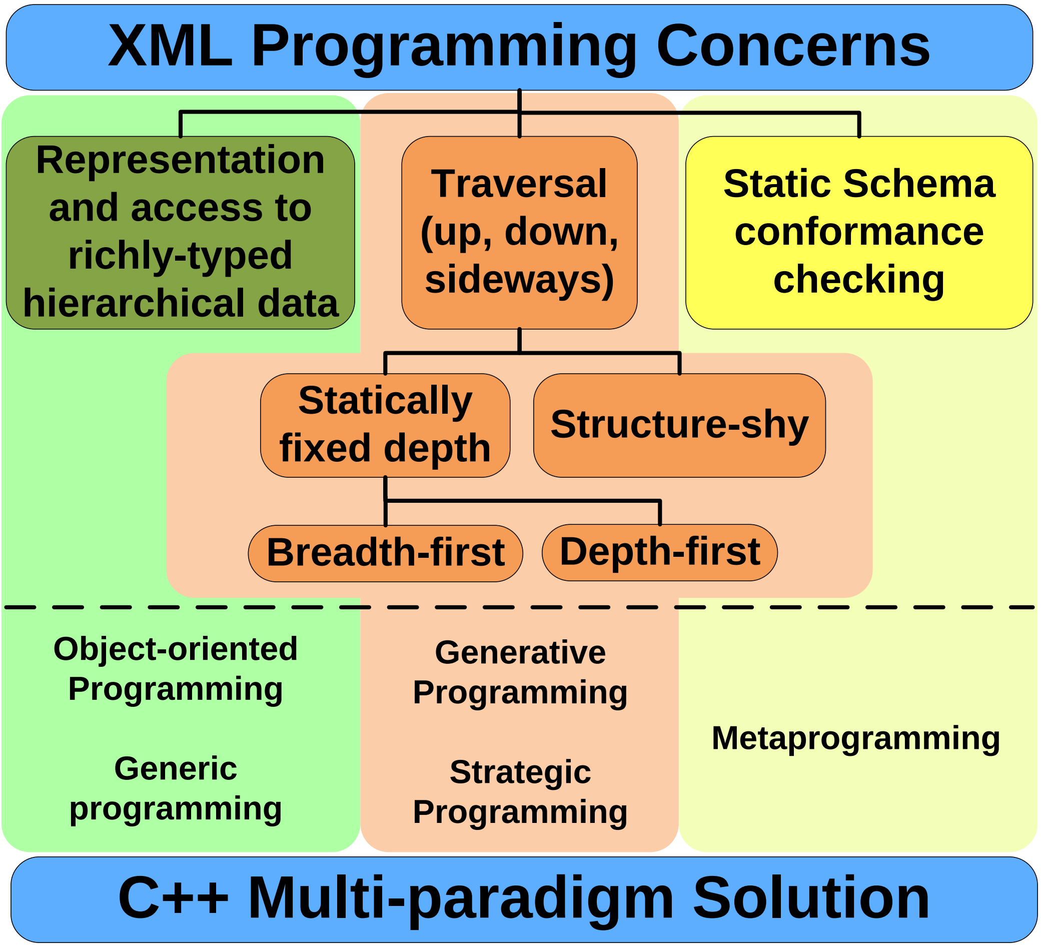
- Become a part of the Boost libraries!?
- Extend LEESA to support
  - Google Protocol Buffers (GPB)
  - Apache Thrift
  - Or any “schema-first” data binding in C++
- Better support from data binding tools?
- Parallelization on multiple cores
  - Parallelize query execution on multiple cores behind LEESA’s high-level declarative programming API
- Co-routine style programming model
  - LEESA expressions return containers
  - Expression to container → expensive!
  - Expression to iterator → cheap!
  - Compute result only when needed (lazy)
- XML literal construction
  - Checked against schema at compile-time



<?xml?>

Concluding Remarks

LEESA → Native XML Processing Using Multi-paradigm Design in C++



Thank You!!