

Generation of Language Bindings for the Document Object Model

Luca Padovani^{ab} Claudio Sacerdoti Coen^a Stefano Zacchiroli^a

{lpadovan, sacerdot, zacchiro}@cs.unibo.it

Department of Computer Science, University of Bologna

Supported by

^a European Project IST-2001-33562 MoWGLI

^b Ontario Research Centre for Computer Algebra

Presentation outline

- C implementation of the Document Object Model
- from the DOM specification to a C++ binding
- code generation: architecture overview
- extending the architecture to a different language
- final considerations

The Document Object Model

“The Document Object Model (DOM) is an **application programming interface (API)** for valid HTML and well-formed XML documents. It defines the **logical structure of documents** and **the way a document is accessed and manipulated.**”

```
interface Element : Node {
  readonly attribute DOMString tagName;
  DOMString getAttribute(in DOMString name);
  void setAttribute(in DOMString name, in DOMString value)
    raises(DOMException);
  ...
  NodeList getElementsByTagName(in DOMString name);
  ...
};
```

C implementation

Not OO:

```
GdomeException exc;
GdomeNode *p = gdome_el_firstChild (el, &exc);
while (p != NULL) {
    GdomeNode *next = gdome_n_nextSibling (p, &exc);
    if (gdome_n_nodeType (p, &exc) == GDOM_ELEMENT_NODE) {
        GdomeElement *pel = gdome_el_cast (p);
        /* do something with the element */
    }
    gdome_n_unref (p, &exc);
    p = next;
}
```

C implementation

Explicit and invasive handling of **exceptions**:

```
GdomeException exc;
GdomeNode *p = gdome_el_firstChild (el, &exc);
while (p != NULL) {
    GdomeNode *next = gdome_n_nextSibling (p, &exc);
    if (gdome_n_nodeType (p, &exc) == GDOM_ELEMENT_NODE) {
        GdomeElement *pel = gdome_el_cast (p);
        /* do something with the element */
    }
    gdome_n_unref (p, &exc);
    p = next;
}
```

C implementation

Explicit and tempting handling of **casts**:

```
GdomeException exc;
GdomeNode *p = gdome_el_firstChild (el, &exc);
while (p != NULL) {
    GdomeNode *next = gdome_n_nextSibling (p, &exc);
    if (gdome_n_nodeType (p, &exc) == GDOM_ELEMENT_NODE) {
        GdomeElement *pel = gdome_el_cast (p);
        /* do something with the element */
    }
    gdome_n_unref (p, &exc);
    p = next;
}
```

C implementation

Explicit and error-prone handling of **memory with reference counters**:

```
GdomeException exc;
GdomeNode *p = gdome_el_firstChild (el, &exc);
while (p != NULL) {
    GdomeNode *next = gdome_n_nextSibling (p, &exc);
    if (gdome_n_nodeType (p, &exc) == GDOMElement_NODE) {
        GdomeElement *pel = gdome_el_cast (p);
        /* do something with the element */
    }
    gdome_n_unref (p, &exc);
    p = next;
}
```

Wish list for an object-oriented API

Close the gap between the API specification and the C implementation:

- no apparent code for exception handling
- native subtyping rules
- automatic dynamic casts
- automatic memory management

The code we would like to write:

```
for (DOM::Node p = el.get_firstChild(); p;  
    p = p.get_nextSibling())  
  if (DOM::Element pe1 = p) {  
    // do something with the element  
  }
```

Recipe for a binding

Ingredients

- a C implementation is already **available**
- it is easy to **call C code from C++**
- the DOM API is **uniform** (name conventions and types)
- we would like to **experiment** without writing tons of code

Outcome

- create a **C++ binding** for the C implementation
- **generate the code** of the binding automatically

And, by the way...

There exists an **informative XML encoding** of the DOM API used for

- generating the W3C recommendation
- generating Java/ECMAScript interfaces/documentation
- generating test suites

which means

- no need to write a **parser for the specification**
- no need to design/use a dedicated transformation language, **XSLT** can generate code

DOM API specification in XML

```
<interface name="Element" inherits="Node">
  <descr>...</descr>
  <attribute type="DOMString" name="tagName" readonly="yes"/>
  <method name="setAttribute">
    <parameters>
      <param name="name" type="DOMString" attr="in"/>
      <param name="value" type="DOMString" attr="in"/>
    </parameters>
    <returns type="void"/>
    <raises>
      <exception name="DOMException">...</exception>
    </raises>
  </method>
</interface>
```

DOM API specification in XML

```
<interface name="Element" inherits="Node">
  <descr>...</descr>
  <attribute type="DOMString" name="tagName" readonly="yes"/>
  <method name="setAttribute">
    <parameters>
      <param name="name" type="DOMString" attr="in"/>
      <param name="value" type="DOMString" attr="in"/>
    </parameters>
    <returns type="void"/>
    <raises>
      <exception name="DOMException">...</exception>
    </raises>
  </method>
</interface>
```

DOM API specification in XML

```
<interface name="Element" inherits="Node">
  <descr>...</descr>
  <attribute type="DOMString" name="tagName" readonly="yes"/>
  <method name="setAttribute">
    <parameters>
      <param name="name" type="DOMString" attr="in"/>
      <param name="value" type="DOMString" attr="in"/>
    </parameters>
    <returns type="void"/>
    <raises>
      <exception name="DOMException">...</exception>
    </raises>
  </method>
</interface>
```

DOM API specification in XML

```
<interface name="Element" inherits="Node">
  <descr>...</descr>
  <attribute type="DOMString" name="tagName" readonly="yes"/>
  <method name="setAttribute">
    <parameters>
      <param name="name" type="DOMString" attr="in"/>
      <param name="value" type="DOMString" attr="in"/>
    </parameters>
    <returns type="void"/>
    <raises>
      <exception name="DOMException">...</exception>
    </raises>
  </method>
</interface>
```

DOM API specification in XML

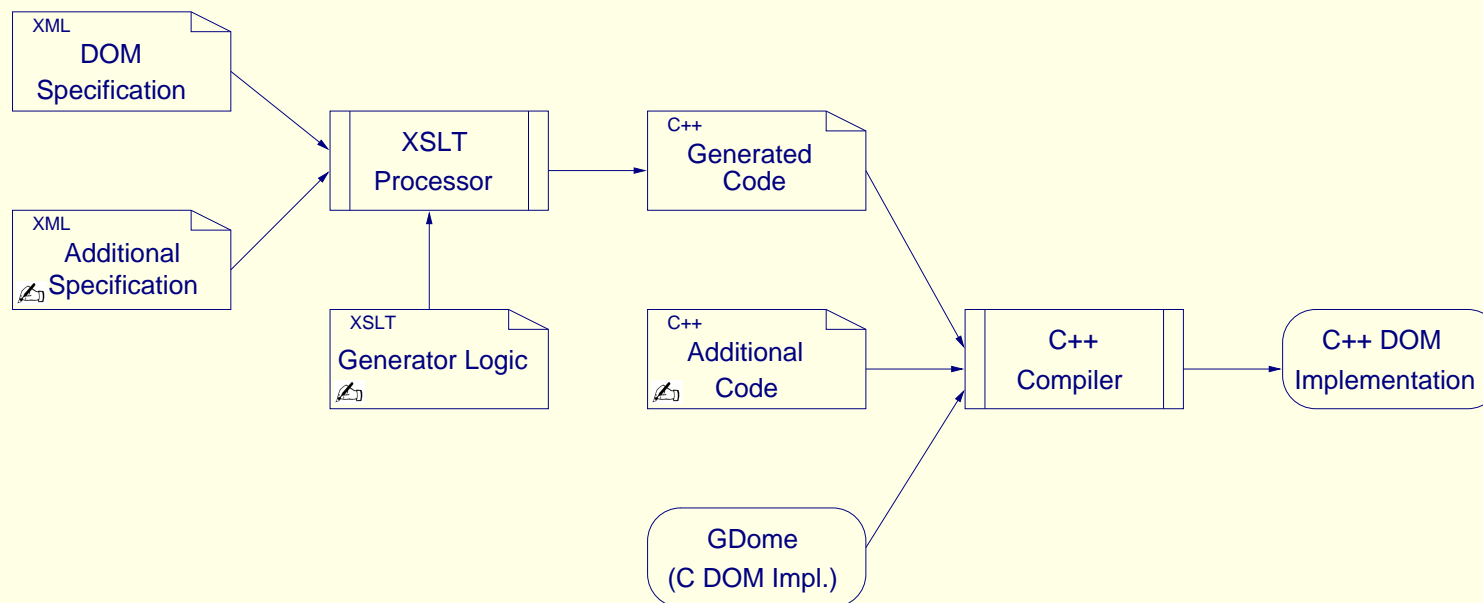
```
<interface name="Element" inherits="Node">
  <descr>...</descr>
  <attribute type="DOMString" name="tagName" readonly="yes"/>
  <method name="setAttribute">
    <parameters>
      <param name="name" type="DOMString" attr="in"/>
      <param name="value" type="DOMString" attr="in"/>
    </parameters>
    <returns type="void"/>
    <raises>
      <exception name="DOMException">...</exception>
    </raises>
  </method>
</interface>
```

DOM API specification in XML

```
<interface name="Element" inherits="Node">
  <descr>...</descr>
  <attribute type="DOMString" name="tagName" readonly="yes"/>
  <method name="setAttribute">
    <parameters>
      <param name="name" type="DOMString" attr="in"/>
      <param name="value" type="DOMString" attr="in"/>
    </parameters>
    <returns type="void"/>
    <raises>
      <exception name="DOMException">...</exception>
    </raises>
  </method>
</interface>
```

Architecture

We use XSLT (transformation language for XML) for writing the **generator logic**



A C++ binding: design choices

We design a family of C++ classes such that

- classes are wrappers for the corresponding C structures
- classes act as smart pointers
- classes respect the DOM hierarchy
- classes have constructors for downcasting

```
class Element : public Node {
    Element(GdomeElement* = 0);
    ~Element();
    GdomeString get_tagName(void) const;
    NodeList getElementsByTagName(const GdomeString& name) const;
    ...
};
```

A C++ binding: generated code

Attr

```
Element::setAttributeNode(const Attr& newAttr) const
{
    GdomeException exc_ = 0;
    GdomeAttr* res_ =
        gdome_el_setAttributeNode(
            (GdomeElement*) gdome_obj,
            (GdomeAttr*) newAttr.gdome_obj,
            &exc_);
    if (exc_ != 0)
        throw DOMException(exc_, "Element::setAttributeNode");
    return Attr(res_, true);
}
```

A C++ binding: generated code

```
Attr
Element::setAttributeNode(const Attr& newAttr) const
{
    GdomeException exc_ = 0;
    GdomeAttr* res_ =
        gdome_el_setAttributeNode(
            (GdomeElement*) gdome_obj,
            (GdomeAttr*) newAttr.gdome_obj,
            &exc_);
    if (exc_ != 0)
        throw DOMException(exc_, "Element::setAttributeNode");
    return Attr(res_, true);
}
```

A C++ binding: generated code

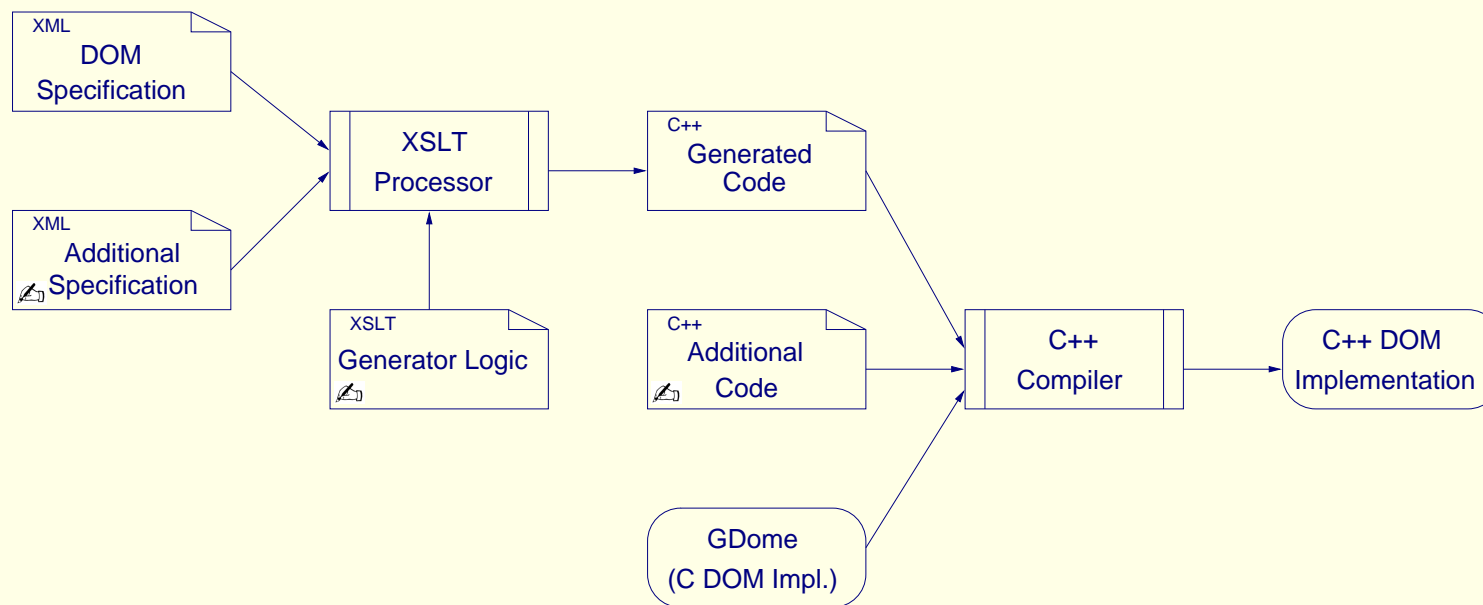
```
Attr
Element::setAttributeNode(const Attr& newAttr) const
{
    GdomeException exc_ = 0;
    GdomeAttr* res_ =
        gdome_el_setAttributeNode(
            (GdomeElement*) gdome_obj,
            (GdomeAttr*) newAttr.gdome_obj,
            &exc_);
    if (exc_ != 0)
        throw DOMException(exc_, "Element::setAttributeNode");
    return Attr(res_, true);
}
```

A C++ binding: generated code

```
Attr
Element::setAttributeNode(const Attr& newAttr) const
{
    GdomeException exc_ = 0;
    GdomeAttr* res_ =
        gdome_el_setAttributeNode(
            (GdomeElement*) gdome_obj,
            (GdomeAttr*) newAttr.gdome_obj,
            &exc_);
    if (exc_ != 0)
        throw DOMException(exc_, "Element::setAttributeNode");
    return Attr(res_, true);
}
```

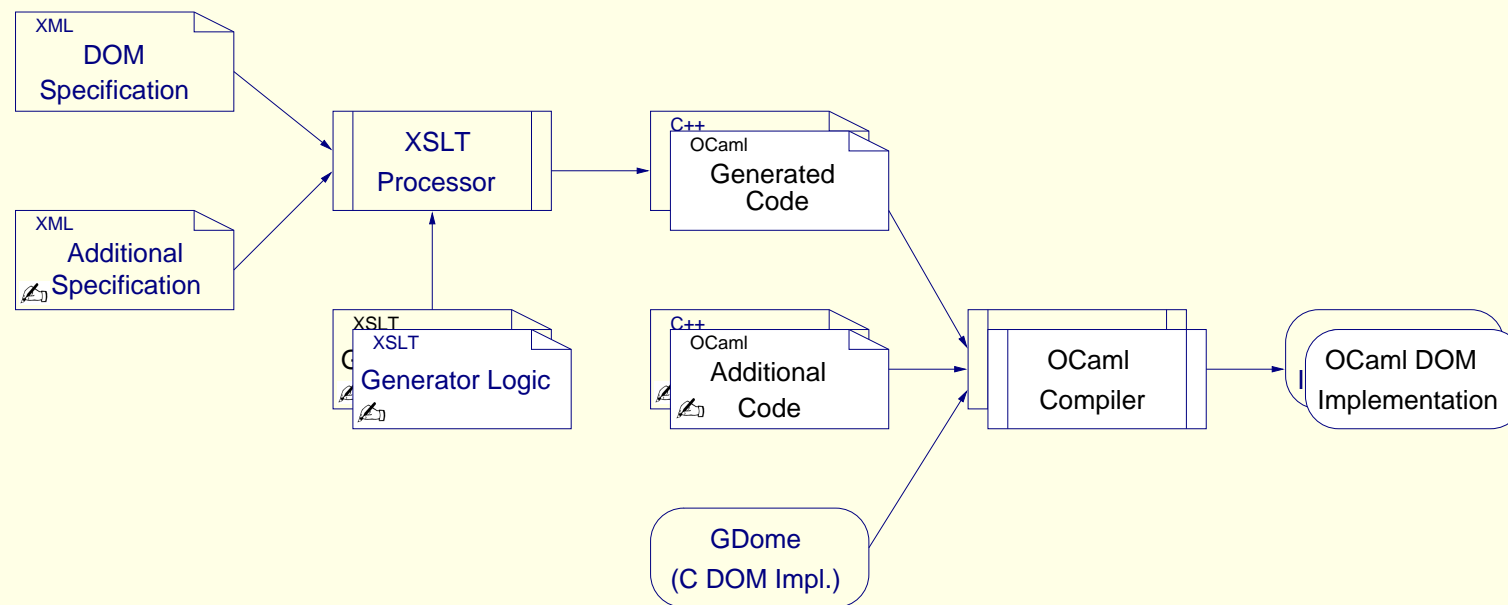
Architecture

We add a **different generator logic** for a **different target language**



Architecture (enriched)

We add a **different generator logic** for a **different target language**



Objective Caml

Core features

- functional language, imperative constructs, garbage collector
- strongly typed, type-inference
- module system
- possible to bind native C functions to OCaml functions

Object-oriented features

- class-based (?) extension of Caml
- subtyping not related to inheritance:
 `$\tau = \langle \text{method as_xml: xml ; ... } \rangle$`
- **not** possible to bind a native C functions to methods

Objective Caml: a layered binding

Object-oriented level:

- **delegation** of invocations to pre-methods
- **projections/promotions**

Very close to the C++ binding!

Functional level:

- **garbage collector** and reference counting
- **callbacks** as closures
- **typing**: defining a collection of OCaml types \mathcal{T} for which the subtyping relation is in accordance with the inheritance relation between DOM interfaces

The object-oriented binding: generated code

```
class element (self : TElement.t) =
  object
    inherit (node (self :> TNode.t))
    method as_Element = self
    method setAttributeNode ~newAttr =
      let res =
        IElement.setAttributeNode
          ~this:self
          ~newAttr:((newAttr : attr)#as_Attr)
      in
        new attr res
  end
```

The object-oriented binding: generated code

```
class element (self : TElement.t) =
  object
    inherit (node (self :> TNode.t))
    method as_Element = self
    method setAttributeNode ~newAttr =
      let res =
        IElement.setAttributeNode
          ~this:self
          ~newAttr:((newAttr : attr)#as_Attr)
      in
      new attr res
  end
```

The object-oriented binding: generated code

```
class element (self : TElement.t) =
  object
    inherit (node (self :> TNode.t))
    method as_Element = self
    method setAttributeNode ~newAttr =
      let res =
        IElement.setAttributeNode
          ~this:self
          ~newAttr:((newAttr : attr)#as_Attr)
      in
      new attr res
  end
```

The object-oriented binding: usage

```
let rec iter =  
  function  
    None ->  
      ()  
  | Some p when p#get_nodeType = GdomeNodeTypeT.ELEMENT_NODE ->  
    let p' = Gdome.element_of_node p in  
      (* do something with p' *)  
      iter p#get_nextSibling  
  | Some p ->  
    iter p#get_nextSibling  
in  
  iter el#get_firstChild
```

Not as elegant as the C++ binding, but the DOM API is first-order!

The functional binding: first attempt

OCaml **abstract data types** can be used to represent native C pointers (in particular pointers to Gdome structures):

- use different abstract data type for each Gdome structure
- provide functions for **explicit up/down casts**

This approach is **inefficient** and **cumbersome**:

- cast functions are identities, but **cannot be optimized**
- number of casts is **proportional** to the distance of the classes in the DOM hierarchy

Phantom types

$\alpha \tau$ is called a **phantom type** when α is not used in the definition of the type

Two values of distinct types $\sigma_1 \tau$ and $\sigma_2 \tau$ have the same memory representation

Assuming that there is a subtyping relation $<:$ between σ_1 and σ_2 , we can specify the variance of the phantom type

$\alpha \tau$ is **covariant** in α :

$$\sigma_1 <: \sigma_2 \Rightarrow \sigma_1 \tau <: \sigma_2 \tau$$

$-\alpha \tau$ is **contravariant** in α

$$\sigma_1 <: \sigma_2 \Rightarrow \sigma_2 \tau <: \sigma_1 \tau$$

Polymorphic variants

A **polymorphic variant value** is a tag associated with an optional value

A **close polymorphic variant type** lists several distinct tags:

$$\sigma_1 = [\text{'EventTarget} \mid \text{'Node}]$$

Property: $[\text{'Node}]$ is a subtype of $[\text{'EventTarget} \mid \text{'Node}]$

A **parametric polymorphic variant type** has an unnamed parameter representing an unspecified tag set

$$\sigma_2 = [> \text{'EventTarget} \mid \dots]$$

Property: $\sigma_1 \tau$ matches $\sigma_2 \tau$

Tags as capabilities

Consider the following interpretation for tags:

- tags in a closed contravariant phantom type instance as a list of **provided capabilities**: [`'Node` | `'Element`]
- tags in a parametric contravariant phantom type instance as a list of **required capabilities**: [`> 'Node` | `..`]

A DOM method type

`Node` \rightarrow `Element`

becomes the OCaml type

`[> 'Node` | `..] τ \rightarrow ['Node | 'Element] τ`

Multiple inheritance: [`'EventTarget` | `'Node` | `'Element`]

Why is it working?

	DOM	OCaml
Supertype	Node	$[\text{'Node}] \tau$
Subtype	Element	$[\text{'Node} \mid \text{'Element}] \tau$
Relation	Element \leq : Node because Element extends Node	$[\text{'Node} \mid \text{'Element}] \tau \leq$: $[\text{'Node}] \tau$ α is contravariant in $\alpha \tau$ and $[\text{'Node}] \leq$: $[\text{'Node} \mid \text{'Element}]$ since $\{\text{'Node}\} \subset \{\text{'Node}, \text{'Element}\}$
Method	$m : \text{Node} \rightarrow T$	$m : [> \text{'Node} \mid \dots] \tau \rightarrow [T]_o$
Object	$e : \text{Element}$	$e : [\text{'Node} \mid \text{'Element}] \tau$
Application	$m(e)$ well typed because $e : \text{Element} \Rightarrow e : \text{Node}$ by subsumption	$m(e)$ well typed because $[\text{'Node} \mid \text{'Element}] \tau$ matches $[> \text{'Node} \mid \dots] \tau$

The functional binding: generated code

```
external insertBefore :
  this:[> 'Node] GdomeT.t ->
  newChild:[> 'Node] GdomeT.t ->
  refChild:[> 'Node] GdomeT.t option ->
  ['EventTarget | 'Node] GdomeT.t
= "ml_gdome_n_insertBefore"

value
ml_gdome_n_insertBefore(value self, value p_newChild, value p_refChild)
{
  CAMLparam3(self, p_newChild, p_refChild);
  GdomeException exc_;
  GdomeNode* res_;
  res_ = gdome_n_insertBefore(Node_val(self),
                              Node_val(p_newChild),
                              ptr_val_option(p_refChild,Node_val),
                              &exc_);
  if (exc_ != 0) throw_exception(exc_, "Node.insertBefore");
  g_assert(res_ != NULL);
  CAMLreturn(Val_Node(res_));
}
```

```
<xsl:template match="method">
  <xsl:param name="interface" select="''"/>
  <xsl:param name="prefix" select="''"/>
  <xsl:variable name="name" select="@name"/>
  <xsl:text> method </xsl:text>
  <xsl:value-of select="@name"/>
  <xsl:apply-templates mode="left" select="parameters">
    <xsl:with-param name="name" select="@name"/>
  </xsl:apply-templates>
  <xsl:text> = </xsl:text>
  <xsl:call-template name="call_pre_method">
    <xsl:with-param name="type" select="returns/@type"/>
    <xsl:with-param name="isNullable"
      select="document($annotations)/Annotations/Method
        [@name=$name]/@nullable='yes'"/>
    <xsl:with-param name="action">
      <xsl:text>I</xsl:text>
      <xsl:value-of select="$interface"/>.
      <xsl:value-of select="@name"/>
      <xsl:text> ~this:obj </xsl:text>
      <xsl:apply-templates mode="right" select="parameters">
        <xsl:with-param name="name" select="@name"/>
      </xsl:apply-templates>
    </xsl:with-param>
  </xsl:call-template>
</xsl:template>
```

```

<xsl:template match="method">
  <xsl:param name="interface" select="''"/>
  <xsl:param name="prefix" select="''"/>
  <xsl:variable name="name" select="@name"/>
  <xsl:text> method </xsl:text>
  <xsl:value-of select="@name"/>
  <xsl:apply-templates mode="left" select="parameters">
    <xsl:with-param name="name" select="@name"/>
  </xsl:apply-templates>
  <xsl:text> = </xsl:text>
  <xsl:call-template name="call_pre_method">
    <xsl:with-param name="type" select="returns/@type"/>
    <xsl:with-param name="isNullable"
      select="document($annotations)/Annotations/Method
        [@name=$name]/@nullable='yes'"/>
    <xsl:with-param name="action">
      <xsl:text>I</xsl:text>
      <xsl:value-of select="$interface"/>.
      <xsl:value-of select="@name"/>
      <xsl:text> ~this:obj </xsl:text>
      <xsl:apply-templates mode="right" select="parameters">
        <xsl:with-param name="name" select="@name"/>
      </xsl:apply-templates>
    </xsl:with-param>
  </xsl:call-template>
</xsl:template>

```

```

<xsl:template match="method">
  <xsl:param name="interface" select="''"/>
  <xsl:param name="prefix" select="''"/>
  <xsl:variable name="name" select="@name"/>
  <xsl:text> method </xsl:text>
  <xsl:value-of select="@name"/>
  <xsl:apply-templates mode="left" select="parameters">
    <xsl:with-param name="name" select="@name"/>
  </xsl:apply-templates>
  <xsl:text> = </xsl:text>
  <xsl:call-template name="call_pre_method">
    <xsl:with-param name="type" select="returns/@type"/>
    <xsl:with-param name="isNullable"
      select="document($annotations)/Annotations/Method
        [@name=$name]/@nullable='yes'"/>
    <xsl:with-param name="action">
      <xsl:text>I</xsl:text>
      <xsl:value-of select="$interface"/>.
      <xsl:value-of select="@name"/>
      <xsl:text> ~this:obj </xsl:text>
      <xsl:apply-templates mode="right" select="parameters">
        <xsl:with-param name="name" select="@name"/>
      </xsl:apply-templates>
    </xsl:with-param>
  </xsl:call-template>
</xsl:template>

```

```

<xsl:template match="method">
  <xsl:param name="interface" select="''"/>
  <xsl:param name="prefix" select="''"/>
  <xsl:variable name="name" select="@name"/>
  <xsl:text> method </xsl:text>
  <xsl:value-of select="@name"/>
  <xsl:apply-templates mode="left" select="parameters">
    <xsl:with-param name="name" select="@name"/>
  </xsl:apply-templates>
  <xsl:text> = </xsl:text>
  <xsl:call-template name="call_pre_method">
    <xsl:with-param name="type" select="returns/@type"/>
    <xsl:with-param name="isNullable"
      select="document($annotations)/Annotations/Method
        [@name=$name]/@nullable='yes'"/>
    <xsl:with-param name="action">
      <xsl:text>I</xsl:text>
      <xsl:value-of select="$interface"/>.
      <xsl:value-of select="@name"/>
      <xsl:text> ~this:obj </xsl:text>
      <xsl:apply-templates mode="right" select="parameters">
        <xsl:with-param name="name" select="@name"/>
      </xsl:apply-templates>
    </xsl:with-param>
  </xsl:call-template>
</xsl:template>

```

Annotations

Information that is not available from the DOM specification can be separately provided as **annotations**.

```
<Annotations>
```

```
...
```

```
<Attribute name="parentNode" nullable="yes"/>
```

```
<Attribute name="childNodes" nullable="no"/>
```

```
...
```

```
<Method name="insertBefore" nullable="no">
```

```
  <Param name="newChild" nullable="no"/>
```

```
  <Param name="refChild" nullable="yes"/>
```

```
</Method>
```

```
...
```

```
</Annotations>
```

A quantitative evaluation

	DOM Core module			+ DOM Events module		
	Generator logic (XSLT)	Hand written code (target language)	Generated code (target language)	Generator logic (XSLT)	Hand written code (target language)	Generated code (target language)
C++	768	1405	4289	+40	+74	+915
+ Caml	+1514	+1305	4557	+15	+176	+807
+ OCaml	+640	+291	+2407	+4	+44	+284

Summary

- **XSLT**-based generator
- flexible XML-based architecture
- concrete advantages
- code easier to generate than comments
- several DOM modules and APIs still to implement

<http://gmetadom.sourceforge.net/>