Storing an XML document into the OODB

This page contains the algorithms for mapping an XML document with its DTD to an OODB.

```
function map_document(URI) returns DocOID
   DTD := retrieve_dtd(URI)
   DocType := retrieve_doc_type(URI)
   create_schema(DocType,DTD)
   Doc := retrieve_doc(URI)
   TopElemNodes := create_instances(Doc)
   new([DocOID,[uri([URI]),children(TopElemNodes)]]) => xml_doc

function create_schema(TopElement,DTD) returns ∅
   global ElemAttr, EmptyElem:= ∅
   for_each X in DTD
      such_that
         ((X = '<!ELEMENT' ElemName '(#PCDATA)' '>' or
           X = '<!ELEMENT' ElemName '(#PCDATA)*' '>') and
         ElemName ≠ TopElement and
         '<!ATTLIST' ElemName * '>' ∉ DTD)
      do
         ElemAttr := ElemAttr ∪ {ElemName}
         DTD := DTD - {X}
   for_each X in DTD
      such_that
         X = '<!ELEMENT' ElemName 'EMPTY' '>'
      do
         EmptyElem := EmptyElem ∪ {ElemName}
         if (ElemName ≠ TopElement and
            '<!ATTLIST' ElemName * '>' ∉ DTD) then
            ElemAttr := ElemAttr ∪ {ElemName}
            DTD := DTD - {X}
   for_each X in DTD
      such_that
         X = '<!ELEMENT' ElemName ElementDecl '>'
      do
         if ElementDecl = '(' ChildrenElements ')' then
            if ChildrenElements = $Elem_1$|...|$Elem_n$ then
               ChildrenElements := '(' ChildrenElements ')'
         else
            ChildrenElements = ∅
         create_class(xml_seq,ElemName,ChildrenElements)
         DTD := DTD - {X}
         if '<!ATTLIST' ElemName Attributes '>' ∈ DTD then
            AttLst := ∅
            for_each A in Attributes
               such_that
                  A = AttName AttType AttDefault
               do
                  Type := string
                  case AttType of
                     'CDATA','ID','IDREF','ENTITY','NMTOKEN':
                        Card := single
                     'IDREFS','ENTITIES','NMTOKENS':
                        Card := list
                     otherwise:
                        Card := single
                  case AttDefault of
                     '#IMPLIED' :
                        Req := optional
                     '#REQUIRED':
                        Req := mandatory
                     '#FIXED' AttValue :
                        Req := mandatory(AttValue)
                     AttValue :
                        Req := optional(AttValue)
                  AttLst := AttLst ∪ {slot_desc(AttName,Type,Card,Req)}
            put_att_lst(AttLst) => ElemName
            DTD := DTD - {'<!ATTLIST' ElemName Attributes '>'}

function create_class(ClassType,ElemName,ChildrenElements) returns SlotNames
   SlotDefs, SlotNames, ElemOrdAtt, EmptyAtt, AliasAtt:= ∅
   AltCount, SeqCount:= 1
   for_each C in ChildrenElements
      such_that
         C = Elem OccurenceOperator
      do
         if Elem = '(' EncapsulatedElements ')' then
            if EncapsulatedElements = Elem1|...|Elemn then
               AltClassName := ElemName ∪ '_alt' ∪ AltCount
               AltCount := AltCount + 1
               Aliases := create_class(xml_alt,AltClassName,EncapsulatedElements)
               SlotName := AltClassName
               Type := AltClassName
            elseif EncapsulatedElements = Elem1,...,Elemn then
               SeqClassName := ElemName ∪ '_seq' ∪ SeqCount
               SeqCount := SeqCount + 1
               Aliases := create_class(xml_seq,SeqClassName,EncapsulatedElements)
               SlotName := SeqClassName
               Type := SeqClassName
            SlotNames := SlotNames ∪ Aliases
            for_each A in Aliases
               AliasAtt := AliasAtt ∪ {A - SlotName}
         else
            if Elem = '#PCDATA'
               SlotName := content
            else
               SlotName := Elem
            SlotNames := SlotNames ∪ {SlotName}
            if Elem ∈ ElemAttr or Elem = '#PCDATA' then
               Type := string
            else
               Type := Elem
         case OccurenceOperator of
            ∅, '?' :
               Card := single
            '*', '+' :
               Card := list
         if ClassType = xml_alt then
            Req := optional
         else
            case OccurenceOperator of
               ∅, '+' :
                  Req := mandatory
               '?', '*':
                  Req := optional
            ElemOrdAtt := ElemOrdAtt ∪append {Elem}
         if Elem ∈ EmptyElem then
            EmptyAtt := EmptyAtt ∪ {Elem}
         SlotDefs := SlotDefs ∪ {slot_desc(SlotName,Type,Card,Req)}
   ClassAtts := {elem_ord(ElemOrdAtt)} ∪ {empty(EmptyAtt)} ∪ {alias(AliasAtt)}
   new([ElemName, ClassAtts ∪ SlotDefs]) => ClassType
```

```
function create_instances(Doc) returns TopElemNodes
   Doc := Doc - {'<!DOCTYPE' RootElement * '>'}
   TopElemNodes := ∅
   for_each X in Doc
      such_that
         X = '<'RootElement Attributes '>' Contents '</'RootElement'>'
      do
         TopElemNode := create_instance(RootElement,Attributes,Contents)
         TopElemNodes := TopElemNodes ∪ {TopElemNode}
```

```
function create_instance(Class,Attributes,Contents) returns OID
   global Contents
   get_elem_order(ElemOrder) => Class
   get_att_lst(AttList) => Class
   get_alias(Alias) => Class
   get_empty(EmptyAtts) => Class
   SlotTuples := ∅
   for_each Att in AttList
      do
         get_slot_desc(slot_desc(Att,AttType,Card,Req)) => Class
         if (Req = mandatory or Req = optional) then
            Attributes := Attributes - {Att '=' Val}
         elseif Req = optional(Default) then
            if {Att '=' Val} ∉ Attributes then
               Val := Default
            else
               Attributes := Attributes - {Att '=' Val}
         elseif Req = mandatory(Default) then
            Val := Default
            Attributes := Attributes - {Att '=' *}
      if Val ≠ ∅ then
         if Card = single then
            Value := {Val}
         else
            Value := Val
         SlotTuples:= SlotTuples ∪ {Att(Value)}
for_each Elem in ElemOrder
   do
      get_slot_desc(slot_desc(Elem,Type,Card,Req)) => Class
      if Type = string then
         if (Elem \= content or Elem ∈ EmptyAtts) then
            Value := ∅
            if (Req = mandatory and Card = single) then
               Contents = [Head|Tail]
               Val := get_val(Head,Elem,EmptyAtts)
               Value := {Val}
               Contents := Tail
            elseif (Req = mandatory and Card = list) then
               repeat
                  Contents = [Head|Tail]
                  Val := get_val(Head,Elem,EmptyAtts)
                  Value := Value ∪ {Val}
                  Contents := Tail
               until not check_head(Contents,Elem)
            elseif (Req = optional and Card = list) then
               while check_head(Contents,Elem)
                  Contents = [Head|Tail]
                  Val := get_val(Head,Elem,EmptyAtts)
                  Value := Value ∪ {Val}
                  Contents := Tail
            elseif (Req = optional and Card = single) then
               if check_head(Contents,Elem) then
                  Contents = [Head|Tail]
                  Val := get_val(Head,Elem,EmptyAtts)
                  Value := {Val}
                  Contents := Tail
         else
            Contents = [Head|Tail]
            if string(Head) then
               Value := {Head}
               Contents := Tail
            else
               Value := {""}
      elseif (*-Elem) ∉ Alias then
         Value := ∅
         if (Req = mandatory and Card = single) then
            Contents = ['<'Elem AttElem '>' ElemContents '</'Elem'>'|Tail]
            Val := create_instance(Elem,AttElem,ElemContents)
            Value := {Val}
            Contents := Tail
         elseif (Req = mandatory and Card = list) then
            repeat
               Contents = ['<'Elem AttElem '>' ElemContents '</'Elem'>'|Tail]
               Val := create_instance(Elem,AttElem,ElemContents)
               Value := Value ∪ {Val}
               Contents := Tail
            until Contents \= ['<'Elem AttElem '>' ElemContents '</'Elem'>'|Tail]
         elseif (Req = optional and Card = list) then
            while Contents = ['<'Elem AttElem '>' ElemContents '</'Elem'>'|Tail]
               Val := create_instance(Elem,AttElem,ElemContents)
               Value := Value ∪ {Val}
               Contents := Tail
         elseif (Req = optional and Card = single) then
            if Contents = ['<'Elem AttElem '>' ElemContents '</'Elem'>'|Tail] then
               Val := create_instance(Elem,AttElem,ElemContents)
               Value := {Val}
               Contents := Tail
      else
         Value := ∅
         if (Req = mandatory and Card = single) then
            Val := create_encaps_instance(Elem,Alias,Contents)
            Value := {Val}
         elseif (Req = mandatory and Card = list) then
            Val := create_encaps_instance(Elem,Alias,Contents)
            repeat
               Value := Value ∪ {Val}
               Val := create_encaps_instance(Elem,Alias,Contents)
            until Val = ∅
            elseif (Req = optional and Card = list) then
               Val := create_encaps_instance(Elem,Alias,Contents)
               while Val ≠ ∅
                  Value := Value ∪ {Val}
                  Val := create_encaps_instance(Elem,Alias,Contents)
            elseif (Req = optional and Card = single) then
               Val := create_encaps_instance(Elem,Alias,Contents)
               if Val ≠ ∅
                  Value := {Val}
         SlotTuples := SlotTuples ∪ {Elem(Value)}
   OID := create_object(Class,SlotTuples)

function create_encaps_instance(SystemClass,Aliases,Contents) returns OID
   get_instance_of(MetaClass) => SystemClass
   if MetaClass = xml_seq then
      OID := create_encaps_instance(SystemClass,∅,Contents)
   else
      OID := create_encaps_alt_instance(SystemClass,Aliases,Contents)

function create_encaps_alt_instance(SystemClass,Aliases,Contents) returns OID
   global Contents
   get_alias(Alias) => SystemClass
   get_empty(EmptyAtts) => SystemClass
   SlotTuples, ElemContents, OID:= ∅
   if (Contents = [Head|Tail] and string(Head) and content-SystemClass ∈ Aliases) then
      Contents := Tail
      OID := create_object(SystemClass,content([Head]))
   elseif ((Contents = ['<'Elem AttElem '>' ElemContents '</'Elem'>'|Tail] or
               Contents = ['<'Elem AttElem '/>'|Tail]) and
               Elem-SystemClass ∈ Aliases) then
      get_slot_desc(slot_desc(Elem,Type,Card,Req)) => SystemClass
      if Type = string then
         Value := ∅
         if Card = single then
            if ElemContents ≠ ∅ then
               Value := {ElemContents}
            else
               Value := {yes}
            Contents := Tail
         elseif Card = list then
            ElemContents := ∅
            while (Contents = ['<'Elem'>' ElemContents '</'Elem'>'|Tail] or
                  Contents = ['<'Elem'/>'|Tail])
               if ElemContents ≠ ∅ then
                  Val := ElemContents
               else
                  Val := yes
               Value := Value ∪ {Val}
               Contents := Tail
         SlotTuple := Elem(Value)
      elseif (Elem-*) ∉ Alias then
         Value := ∅
         if Card = single then
            Val := create_instance(Elem,AttElem,ElemContents)
            Value := {Val}
            Contents := Tail
         elseif Card = list then
            ElemContents := ∅
            while (Contents = ['<'Elem AttElem '>' ElemContents '</'Elem'>'|Tail] or
                  Contents = ['<'Elem AttElem '/>'|Tail])
               Val := create_instance(Elem,AttElem,ElemContents)
               Value := Value ∪ {Val}
               Contents := Tail
         SlotTuple := Elem(Value)
      else
         (Elem-EncapsClass) ∈ Alias
         Value := ∅
         if Card = single then
            Val := create_encaps_instance(EncapsClass,Alias,Contents)
            Value := {Val}
         elseif Card = list then
            Val := create_encaps_instance(EncapsClass,Alias,Contents)
            while Val ≠ ∅
               Value := Value ∪ {Val}
               Val := create_encaps_instance(Elem,Alias,Contents)
         SlotTuple := EncapsClass(Value)
      OID := create_object(SystemClass,[SlotTuple])
```

```
function create_object(Class,SlotTuples) returns OID
   if (get(OID) => Class) and
      (for_each X in SlotTuples
         such that
            (X = Slot(Value)) and (get_Slot(Value) => OID)) then
      true
   else
      new([OID, SlotTuples]) => Class

function get_head(Head,Elem,EmptyAtts) returns Val
   if (Head = '<' Elem '/>' or Head = '<' Elem '>' '</' Elem '>') then
      if Elem ∈ EmptyAtts
         Val := yes
      else
         Val := ""
   else
      Head = '<' Elem '>' Val '</' Elem '>'

function check_head(Contents,Elem) returns Flag
   if (Contents = ['<' Elem '/>' | Tail] or
      Contents = ['<' Elem '>' '</' Elem '>' | Tail] or
      Contents = ['<' Elem '>' Val '</' Elem '>' | Tail]) then
      Flag := true
   else
      Flag := false
```