Table des matières

List of Algorithms

1 Modélisation d'un document xml

La structure d'un document XML est modélisée de manière relativement classique. On notera l'utilisation d'un *design pattern* « composite ». Seule la classe `Node` est abstraite.

FIGURE 1 – TODO : Diagramme

1.1 `Node`

Classe racine abstraite.
Elle définit une méthode pour accéder au noeud parent et une méthode abstraite permettant d'accepter un visiteur.

1.2 `MarkupNode`

Classe instanciable représentant une balise vide (de type `<balise />`, sans contenu) dans l'arbre XML.
Elle permet de parcourir ses attributs (au sens XML) en lecture seule, ainsi que d'accéder à son espace de nom et son nom.

1.3 `CompositeMarkupNode`

Classe instanciable représentant une balise non vide (de type `<balise>...</balise>`, avec éventuellement du contenu) dans l'arbre XML.
Elle permet de parcourir ses noeuds fils en lecture seule, ainsi que toutes les opérations définies par `MarkupNode`.

1.4 `TextNode`

Classe instanciable représentant un texte inclus dans une balise (de type `PCDATA`) dans l'arbre XML.
Elle permet d'accéder à son contenu texte en lecture seule. TODO : Visitor

2 Modélisation d'une dtd

FIGURE 2 – TODO : Diagramme

TODO : Classes DTD, Element, etc. `Content` représente le contenu d'un élément dans la DTD, c'est-à-dire la dernière partie de la balise `<!ELEMENT nom contenu>`. Il permet d'effectuer une validation de la structure d'un noeud XML. Il définit pour cela une interface simple composée des deux versions de la méthode `validate` ; celle-ci renvoie vrai ou faux en fonction de l'adéquation du contenu du noeud passé en paramètre avec le contenu (classe `Content`) sur lequel a été appelé la méthode. On suppose, lorsque ces méthodes sont appelées, que la version appelée correspond au type réel du noeud.

Parmi les classes dérivées directement de `Content`, on retrouve :

`EmptyContent` Contenu vide. Valide n'importe quel noeud de type `MarkupNode`.

`AnyContent` Contenu quelconque. Valide n'importe quel noeud de type `MarkupNode` ou `CompositeMarkupNode`.

`BrowsableContent` Contenu plus complexe, potentiellement composite et organisé sous forme d'arbre, et donc navigable pour la validation.

TODO : QuantifiedContent, Visitor

2.1 Validation

La validation consiste, à partir de la liste des noeuds fils d'un noeud XML, à leur faire correspondre un arbre de contenu (classe `Content`).

2.1.1 Algorithme général

L'algorithme de validation effectue un parcours en profondeur de l'arbre de contenu, avec backtracking. Dans la suite du document, nous étudierons séparément la navigation dans l'arbre, le backtracking et la validation de chaque type particulier de contenu.

2.1.2 Navigation

Une validation suit, classiquement, un parcours en profondeur de l'arbre de contenu.

La classe abstraite `BrowsableContent` met en place l'ensemble des méthodes nécessaires à la navigation dans l'arbre de contenu. En voici une description succinte :

`+validate(:CompositeMarkupNode):boolean` Effectue la validation proprement dite (interface client).

`#newValidation(firstToken:ChildrenIterator, endToken:ChildrenIterator, nextStep:Browsable` Méthode protégée appelée sur le contenu suivant lors d'un accès en « descendant » dans l'arbre. Tente de valider (de « matcher ») un maximum de jetons (noeuds enfants) de la liste donnée en paramètre, avec pour contrainte que la validation de la prochaine étape reste d'actualité.

`#continueValidation(currentToken:ChildrenIterator):boolean` Méthode protégée appelée sur le contenu suivant lors d'un accès en « remontant » dans l'arbre. Continue la validation entreprise lors du dernier appel à newValidation sur le même objet.

Ainsi, la navigation pourrait être modélisée comme suit :

Procedure 1 `newValidation`

Input: $firstNodeIterator$ and $endNodeIterator$, iterators to a list of nodes
Input: *nextStep*, another BrowsableContent (optional)
Output: True if the node has been validated, false otherwise.

```
Perform some validations on the node (potentially incrementing firstNodeIterator)
for all child node child do
    Call child.newValidation(firstNodeIterator, endNodeIterator, self)
end for
Perform some other validations on the node (potentially incrementing
firstNodeIterator)
if each validation was successful then
    if nextStep is a parameter then
        return nextStep.continueValidation(firstNodeIterator)
    else
        if firstNodeIterator has reached the end of the list then
            return true
        else
            return false
        end if
    end if
else
    return false
end if
```

Ce qui correspond aux diagrammes suivants :

FIGURE 3 – Diagramme de séquences d'un exemple de parcours

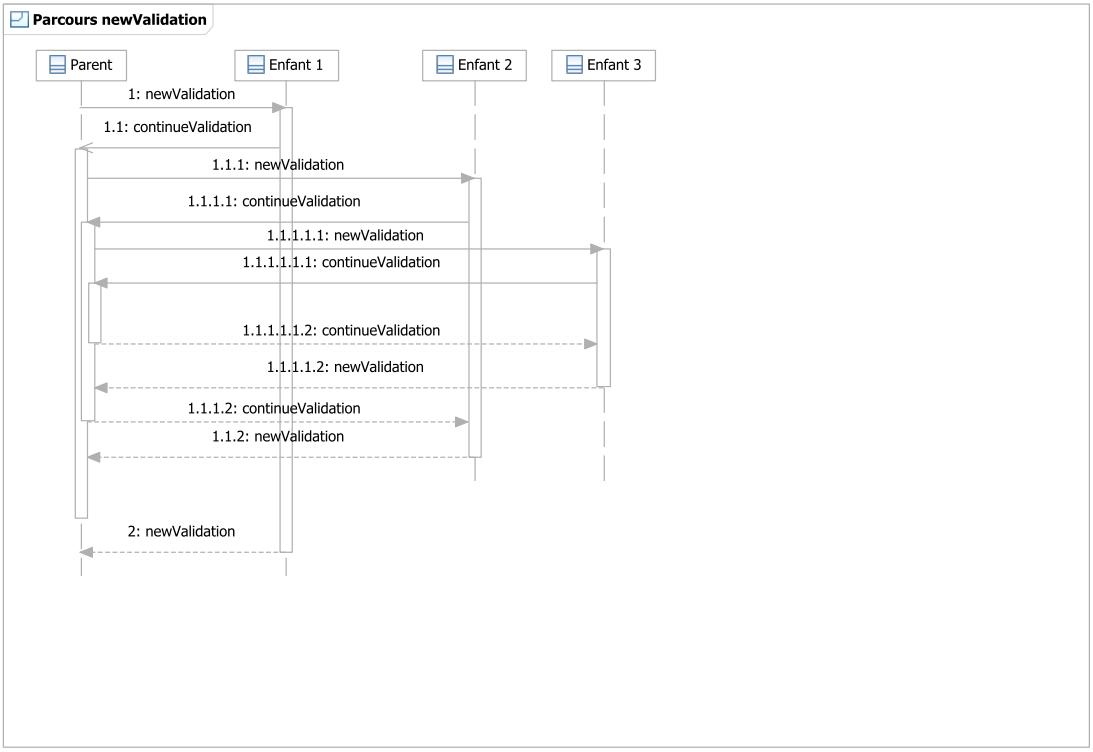
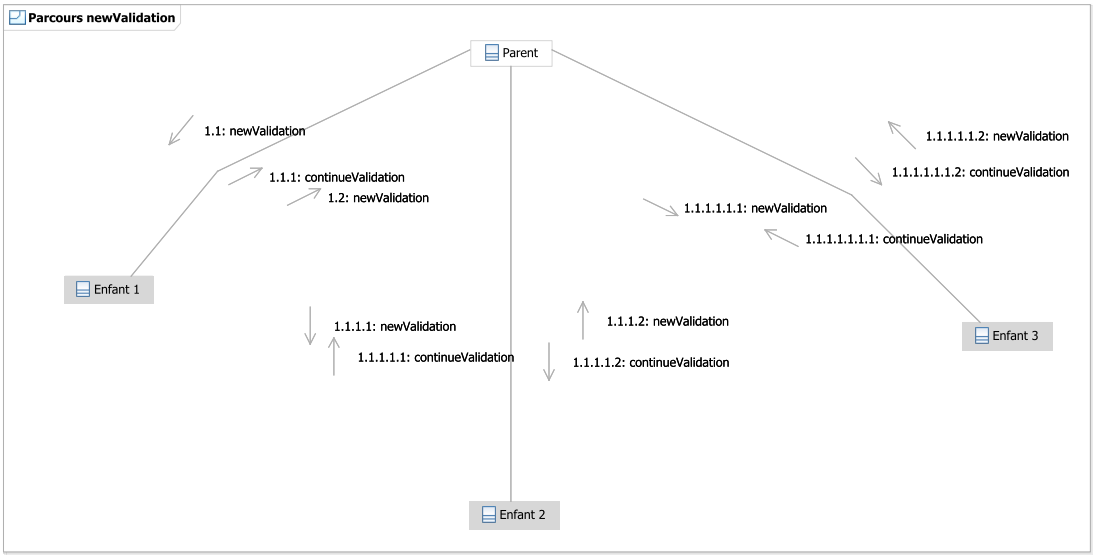


FIGURE 4 – Diagramme de collaboration d'un exemple de parcours



Cependant, les méthodes `newValidation` et `continueValidation` sont protégées, et ne peuvent donc pas être appelées depuis d'autres instances de classes dérivées (en C++, tout au moins). Afin d'accéder à ces méthodes (protégées) depuis d'autres objets de classes dérivées de `BrowsableContent`, il faut définir les méthodes suivantes :

`#browseDown(childContent:BrowsableContent, firstToken:ChildrenIterator, endToken:Childre`
: Exécute la descente dans l'arbre de contenu. Consiste à appeler la méthode `newValidation` sur le contenu `childContent`.

`#browseUp(parentContent:BrowsableContent, currentToken:ChildrenIterator, endToken:Childre`
: Exécute la remontée dans l'arbre de contenu. Consiste à appeler la méthode `newValidation` sur le contenu `childContent`. Si l'arbre a été totalement parcouru (nextStep n'existe pas), cette méthode se charge de renvoyer vrai si la liste des enfants a été totalement parcourue, faux sinon.

Ces méthodes sont appelées sur lui-même par un objet désirant continuer la navigation. On obtient le schéma de navigation suivant :

Procedure 2 `newValidation`

Input: $firstNodeIterator$ and $endNodeIterator$, iterators to a list of nodes
Input: $nextStep$, another BrowsableContent (optional)
Output: True if the node has been validated, false otherwise.

Perform some validations on the node (potentially incrementing $firstNodeIterator$)
for all child node *child* **do**
  Call $self.browseDown(child, firstNodeIterator, endNodeIterator, self)$
end for
Perform some other validations on the node (potentially incrementing $firstNodeIterator$)
if each validation was successful **then**
  return $self.browseUp(nextStep, firstNodeIterator, endNodeIterator)$
else
  return false
end if

En réalité, la méthode `newValidation` n'implémente pas directement la validation. Elle se contente d'appeler des patrons de méthodes permettant d'effectuer un prétraitement, une validation proprement dite, et un post-traitement. Cela permet d'effectuer des empilements d'états nécessaires à l'implémentation des opérateurs de quan-

tification (cf. section **??**). Les patrons en questions (méthodes virtuelles définies dans les classes dérivées) sont les suivants :

`#beforeValidation(firstToken:ChildrenIterator, endToken:ChildrenIterator, nextStep:Browsa`
: Pré-traitement précédant une validation.

`#startValidation(firstToken:ChildrenIterator, endToken:ChildrenIterator, nextStep:Browsab`
: Première étape d'une validation, potentiellement suivie d'appels à `continueValidation`. C'est cette méthode en particulier qui implémente les spécificités du « matching » de chaque type de contenu.

`#afterValidation():boolean` Post-traitement précédant une validation.

2.1.3 Backtracking

Etant donnée la manière dont l'arbre est parcouru, chaque noeud effectuant un choix (`Choice`, `QuantifiedContent`, `MixedContent`) est mis au courant si la suite de la validation n'a pas pu se faire. Il est donc possible, au niveau de chacun de ces noeuds, de gérer le backtracking :

- après un `Choice` ou un `MixedContent`, si la suite de la validation n'a pas pu être effectuée, on essaye le choix suivant. Si on a atteint la fin de la liste, on renvoie faux pour signaler au contenu parent qu'il faut effectuer un backtracking.
- après un `QuantifiedContent`, si la suite de la validation n'a pas pu être effectuée, on tente de « matcher » un jeton de moins, puis de relancer la suite de la validation. Si on arrive en dessous du nombre minimum de « matching », on renvoie faux pour signaler au contenu parent qu'il faut effectuer un backtracking.

2.1.4 Validation spécifique

Ici sont décrites les spécificités de la validation de chaque type de contenu, donc les différentes implémentations de `startValidation`, et s'il y a lieu de `continueValidation`.

Procedure 3 `TextContent::startValidation`

Input: $firstNodeIterator$ and $endNodeIterator$, iterators to a list of nodes
Input: $nextStep$, another BrowsableContent (optional)
Output: True if the node has been validated, false otherwise.

```
if node pointed by firstNodeIterator is an instance of TextNode then
  return self.browseUp(nextStep, firstNodeIterator, endNodeIterator)
else
  return false
end if
```

Procedure 4 `ElementReference::startValidation`

Input: $firstNodeIterator$ and $endNodeIterator$, iterators to a list of nodes
Input: $nextStep$, another BrowsableContent (optional)
Output: True if the node has been validated, false otherwise.

```
if node pointed by firstNodeIterator is an instance of TextNode then
  return false
else
  if namespace and name of firstNodeIterator match with this object's ones then
    return self.browseUp(nextStep, firstNodeIterator, endNodeIterator)
  else
    return false
  end if
end if
```

Procedure 5 `Choice::startValidation`

Input: $firstNodeIterator$ and $endNodeIterator$, iterators to a list of nodes
Input: $nextStep$, another BrowsableContent (optional)
Output: True if the node has been validated, false otherwise.

```
while end of child list has not been reached and no matching has been found among childs do
    self.browseDown(child, firstNodeIterator, endNodeIterator, self)
end while
if a matching has been found then
    return true
else
    return false
end if
```

Procedure 6 `Choice::continueValidation`

Input: $currentNodeIterator$, iterator to a list of nodes
Output: True if the node has been validated, false otherwise.

```
Load endNodeIterator (has to be stored in beforeValidation procedure)
Load nextStep (has to be stored in beforeValidation procedure)
return self.browseUp(nextStep, firstNodeIterator, endNodeIterator)
```

Procedure 7 `MixedContent::startValidation`

Input: $firstNodeIterator$ and $endNodeIterator$, iterators to a list of nodes
Input: $nextStep$, another BrowsableContent (optional)
Output: True if the node has been validated, false otherwise.

```
if node pointed by firstNodeIterator is an instance of TextNode and
self.browseUp(nextStep, firstNodeIterator, endNodeIterator) then
    return true
else
    while end of child list has not been reached and no matching has been found among
    childs do
        self.browseDown(child, firstNodeIterator, endNodeIterator, self)
    end while
    if a matching has been found then
        return true
    else
        return false
    end if
end if
```

Procedure 8 `MixedContent::continueValidation`

Input: $currentNodeIterator$, iterator to a list of nodes
Output: True if the node has been validated, false otherwise.

```
Load endNodeIterator (has to be stored in beforeValidation procedure)
Load nextStep (has to be stored in beforeValidation procedure)
return self.browseUp(nextStep, firstNodeIterator, endNodeIterator)
```

Procedure 9 `Sequence::startValidation`

Input: $firstNodeIterator$ and $endNodeIterator$, iterators to a list of nodes
Input: $nextStep$, another BrowsableContent (optional)
{The $startValidation$ procedure is not different from this one.}
return $self.continueValidation(firstNodeIterator$

Procedure 10 `Sequence::continueValidation`

Input: $currentNodeIterator$, iterator to a list of nodes
Output: True if the node has been validated, false otherwise.
Load $endNodeIterator$ (has to be stored in $beforeValidation$ procedure)
Load $nextStep$ (has to be stored in $beforeValidation$ procedure)
Load current $child$ (has to be stored in $beforeValidation$ procedure)
if The last child has not been reached **then**
{We should match each single child}
return $self.browseDown(child, firstNodeIterator, endNodeIterator, self)$
else
{We matched all childs. The result is up to the next step.}
return $self.browseUp(nextStep, firstNodeIterator, endNodeIterator)$
end if

Procedure 11 `OptionalContent::startValidation`

Input: $firstNodeIterator$ and $endNodeIterator$, iterators to a list of nodes
Input: $nextStep$, another BrowsableContent (optional)
if $self.browseDown(child, firstNodeIterator, endNodeIterator, self)$ **then**
{We first try to include the child in the matching pattern.}
return true
else if $self.browseUp(nextStep, firstNodeIterator, endNodeIterator)$ **then**
{We couldn't include the child in the matching pattern. We try without it.}
return true
else
{It's definitely not possible to match in these conditions. Try backtracking.}
return false
end if

Procedure 12 `OptionalContent::continueValidation`

Input: $currentNodeIterator$, iterator to a list of nodes
Output: True if the node has been validated, false otherwise.
Load $endNodeIterator$ (has to be stored in $beforeValidation$ procedure)
Load $nextStep$ (has to be stored in $beforeValidation$ procedure)
return $self.browseUp(nextStep, firstNodeIterator, endNodeIterator)$

Procedure 13 `RepeateableContent::startValidation`

Input: $firstNodeIterator$ and $endNodeIterator$, iterators to a list of nodes
Input: $nextStep$, another BrowsableContent (optional)
{The $startValidation$ procedure is not different from this one.}
return $self.continueValidation(firstNodeIterator$

Procedure 14 `RepeateableContent::continueValidation`

Input: *currentNodeIterator*, iterator to a list of nodes
Output: True if the node has been validated, false otherwise.
Load *endNodeIterator* (has to be stored in *beforeValidation* procedure)
Load *nextStep* (has to be stored in *beforeValidation* procedure)
if $self.browseDown(child, firstNodeIterator, endNodeIterator, self)$ **then**
{We first try to include the child in the matching pattern.}
return true
else if $self.browseUp(nextStep, firstNodeIterator, endNodeIterator)$ **then**
{We couldn't include the child in the matching pattern. We try without it.}
return true
else
{It's definitely not possible to match in these conditions. Try backtracking.}
return false
end if

Procedure 15 `RepeatedContent::startValidation`

Input: $firstNodeIterator$ and $endNodeIterator$, iterators to a list of nodes
Input: *nextStep*, another BrowsableContent (optional)
{The child has to be matched at least once}
return $self.browseDown(child, firstNodeIterator, endNodeIterator, self)$

Procedure 16 `RepeatedContent::continueValidation`

Input: *currentNodeIterator*, iterator to a list of nodes
Output: True if the node has been validated, false otherwise.
Load *endNodeIterator* (has to be stored in *beforeValidation* procedure)
Load *nextStep* (has to be stored in *beforeValidation* procedure)
if $self.browseDown(child, firstNodeIterator, endNodeIterator, self)$ **then**
{We first try to include the child in the matching pattern.}
return true
else if $self.browseUp(nextStep, firstNodeIterator, endNodeIterator)$ **then**
{We couldn't include the child in the matching pattern. We try without it.}
return true
else
{It's definitely not possible to match in these conditions. Try backtracking.}
return false
end if

10