Le langage DeScript

Le langage DeScript a été conçu par Gérard Tisseau, et implémenté dans le système Combien par Jacques Duma et Gérard Tisseau en 2001.

Le langage Descript est défini dans le même esprit que XML : fournir une base syntaxique et structurelle standardisée servant à l'échange de données, et pouvant être spécialisée pour définir différents nouveaux langages.
Il définit une syntaxe (les "expressions") et sa représentation structurelle (les "noeuds"). Par contre, il ne définit pas de sémantique, ou du moins, sa sémantique est seulement structurelle : tout ce qu'on peut dire, c'est qu'une expression représente un arbre formé de nœuds.

Syntaxe de DeScript : les expressions

• Une *expression* est :
- une valeur *valeur*
- un couple avec deux-points *préfixe1 : valeur*
- un couple avec virgule *préfixe2 , valeur*
- un triplet *préfixe1 : préfixe2 , valeur*

• Une *valeur* est :
- un atome
- une liste

• Un *atome* est une chaîne de caractères quotée ou pas :
- un identificateur (chaîne de caractères normaux)
- une chaîne quotée est encadrée par:
  guillemet "
  apostrophe '
  backquote `

• Une *liste* est constituée d' *éléments* encadrés par:
des parenthèses: (...)
des crochets: [...]
des accolades: { ... }

• Les *éléments* sont des *expressions* quelconques.

• *préfixe1* et *préfixe2* sont des *valeurs* quelconques.

La définition complète précise les règles sur la lecture des caractères *spéciaux*, *normaux*, *séparateurs*. Ces règles ne figurent pas ici. Ce sont en gros les règles usuelles dans les langages informatiques (avec le moins de contraintes et de cas particuliers possibles).

Structure de DeScript : les noeuds

Un nœud est une structure possédant les champs suivants, correspondant aux éléments syntaxiques :

Texte (une chaîne de caractères) : pour un atome uniquement

Préfixe1 (un nœud)
Préfixe2 (un nœud)
TypeGuillemets : le type de guillemets (pour un atome uniquement) : guillemets, apostrophe, backquote, ou rien
TypeParenthèses : le type de parenthèses (pour les listes uniquement) : parenthèses, crochets ou accolades
Eléments (une collection de nœuds) : pour les listes uniquement

Les nœuds contenus dans les champs *préfixe1*, *préfixe2*, et chacun des nœuds de la collection contenue dans le champ *éléments* sont considérés comme des *fils* du nœud possédant ces champs, avec respectivement les *types de filiation* suivants : *préfixe1, préfixe2, élément.*
Chaque nœud possède deux champs indiquant de qui il est le fils, et avec quel type de filiation :

Parent (un nœud)
TypeFiliation

L'ensemble des nœuds correspondant à une expression est structuré en arbre : chaque nœud ne peut avoir qu'un parent, et la racine de l'arbre est le nœud associé à l'expression.

Implementation

Noeuds

Dans le système Combien, un nœud est une instance de la classe **`DeScript.Noeud`**. Pour des raisons historiques, les noms des champs ne sont pas exactement ceux de la présentation formelle ci-dessus. La traduction est la suivante (modèle -> implémentation) :

Texte -> texte
Préfixe1 -> cle
Préfixe2 -> meta
TypeGuillemets , TypeParenthèses -> typeLexical
Eléments -> listeDesFils
Parent -> parent
TypeFiliation -> filiation

Saisie interactive de textes DeScript

La classe d'InterActeur **`Combien.EditeurDeDocuments`** fournit une interface pour éditer des documents écrits en DeScript : utiliser la fenêtre du MiniLanceur, menu "Ouvrir un interacteur…", item "EditeurDeDocuments".
Cet éditeur lit un texte DeScript, vérifie la syntaxe, formate le texte (pretty-print), le traduit en arbre et permet de naviguer dans l'arbre. Il peut ouvrir des fichiers et les enregistrer. Il permet d'utiliser des glossaires d'expressions toutes prêtes.
Voir l'aide dans l'interacteur.

Lecture de textes DeScript par programme

La classe **`DeScript.Lecteur`** permet de lire une chaîne de caractères contenant un texte DeScript et de le traduire en arbre. L'arbre (instance de Noeud) est donné par :

```
(DeScript.Lecteur surTexte: '(a:b c:d)') expression
```

Applications

Le nombre d'applications du langage DeScript dans le système Combien ne cesse d'augmenter, en particulier pour exprimer déclarativement des connaissances, qui sont ensuite traitées par des interpréteurs ou compilateurs. En général, ces connaissances sont stockées sous forme de "ressources" à l'aide de méthodes de classes, suivant un procédé inspiré des *resources* VisualWorks associées aux ApplicationModel (menus, windowSpec, etc.). Elles sont alors éditables par des éditeurs de ressources appropriés (bouton droit de la souris : Edit resource).

Ces déclarations peuvent être considérées comme portables dans d'autres environnements informatiques, car elles sont traduisibles assez facilement en XML (il faut d'abord formaliser les structures XML utilisées en écrivant des DTD, puis écrire des traducteurs, mais ce n'est pas un travail énorme).

Déclarations de concepts

Pour chaque concept (sous-classe de ObjetDuModele), on peut déclarer quels sont ses slots (ses attributs) et les propriétés de ces slots.
Voici un exemple pour la classe ConstructionDePartie (voir la méthode de classe `declarationDuConcept`) :

```
( ConstructionDePartie
 { sousPartie } :
     ( conceptValeur : ConstructionDeSousPartie ) )
```

Pour des explications, voir l'aide dans l'editeur permettant d'éditer cette ressource (faire edit resource).
Cela permet de rajouter des contrôles de type (alors que Smalltalk n'est pas un langage typé), et permet de parler des slots en termes déclaratifs (à travers le nom du slot plutôt qu'à travers ses méthodes d'accès procédurales). On se rapproche ainsi des langages qui peuvent exprimer des faits à l'aide de relations binaires (en particulier les logiques de descriptions). Ces possibilités sont utilisées pour décrire déclarativement des instances (voir plus loin : descriptions).

Déclarations d'éditeurs de faits

Pour chaque interacteur, on peut déclarer les éditeurs de faits qu'il utilise et comment ils sont connectés. Voici un exemple, pour l'interacteur *Combien.ConstrPartie_CE* (voir la méthode de classe `editeursDeFaits`):

```
( editeurPrincipal :
     ( classe : ConstructionDePartie
      fils :
          ( editeurPrincipal
           dans : filsSousPartie
           slot : sousParties )
      fils :
          ( editeurMultiplicite
           dans : self
           slot : multiplicite ) )
 editeurMultiplicite :
     ( classe : Multiplicite ) )
```

Déclaration des schémas d'erreurs associés à une machine

Pour déclarer les schémas d'erreurs associés à une machine, on utilise maintenant une ressource DeScript.
En voici un exemple pour la machine CE (méthode de classe `baseSchemasErreurs` de la classe `Machine_ConstructionEnsemble`) :

```
schema :
    ( cible :
         ( solution
          construction )
     contexte :
         ( solution
          construction )
     detection : "La multiplicité de la construction est
fausse"
     explication : "La multiplicité donnée par l'élève pour sa
construction ne correspond pas à cette construction."
     verificateur : ConstructionCombienValeurFausse )
```

Messages remontants

Maintenant, les objets du modèle forment une arborescence et connaissent leur père. Ils peuvent alors déléguer certains messages à leur père. Il existe (= nous avons implémenté) pour cela un mode spécial d'envoi de message, les *messages remontants*. Pour envoyer un message suivant ce mode, on utilise :
`x envoyerMessage: m`
x commence par examiner s'il est capable de répondre à ce message. Sinon, il le délègue à son père qui le traite suivant le même mode.
Pour qu'un objet puisse savoir s'il est capable de répondre à un message remontant, et pour savoir comment il doit le traiter, il examine sa ressource `reglesMessages` (méthode de classe) écrite en DeScript. Voici par exemple cette ressource pour la classe `Solution` :

```
{ referentiel :
     ( [  ] : referentiel )
 referentielBut :
     ( [  ] : referentielBut )
 referentielSource :
     ( [  ] : referentielSource )
 univers :
     ( [  ] : univers ) }
```

La première règle dit : quand je reçois le message remontant `referentiel` en provenance d'un quelconque de mes descendants, alors j'y réponds avec ma méthode personnelle `referentiel`.
L'existence des messages remontants a permis de supprimer la notion de "contexte lexical", procédurale, pénible à utiliser et source d'erreurs. Cela permet à chaque objet d'exploiter son contexte lexical sans se préoccuper de stocker ce contexte ni même de le connaître : il envoie des messages remontants "dans la nature", en espérant que quelqu'un y répondra (ce qui pour l'instant se produit judicieusement).
Il y a assez peu de règles de ce type (la possibilité de dire "répondre à tous mes descendants" est très puissante).

Descriptions

Pour définir les objets prédéfinis, avant on écrivait des programmes. Maintenant, on écrit des ressources DeScript de type *description.*

Une description décrit une instance d'un concept donné, en utilisant les slots déclarés pour ce concept. Voici une partie d'un exemple de description d' Attribut :

```
(Attribut
     predicat :
          ( PredicatAppartenance
           domaine :
                ( Enumeration
                 element : 'Valet'
                 element : 'Dame'
                 element : 'Roi'
                 typeElements : #texte
                 titreModele : 'figures' )
           titreModele : 'une figure'
           identificateur : #figure )
     ordre :
          ( OrdreParEnumeration
           domaine :
                ( Enumeration
                 element : '7'
                 element : '8'
                 element : '9'
                 element : '10'
                 element : 'Valet'
                 element : 'Dame'
                 element : 'Roi'
                 element : 'As'
                 typeElements : #texte
                 titreModele : 'hauteurs jeu de 32 cartes' )
           titreModele : 'ordre 8 hauteurs'
           identificateur : #hauteurs )
     typeDeValeur : #texte
     titreModele : 'la hauteur'
     identificateur : #hauteur)
```

Il existe un interpréteur de description, capable de créer l'instance à partir du texte. Faire :
`(DeScript.LecteurDescription surTexte: <le texte>) expression evaluer.`

Inversement, un objet du modèle peut s'auto-décrire. Faire :
`x description`

Scripts actionnant une machine

Pour tester les machines, avant il fallait les actionner à la main, ce qui était vite pénible. Maintenant on les actionne par une sorte de programme écrit en DeScript, un *script.* Un script dit en gros sur quels boutons il faut appuyer, quels items de menu il faut choisir, etc. Voici un extrait d'un script pour la machine CE :

```
(filsChoixExercice:
     (choisir: 'Ex 1: 1 dame 1 as'
      valider)
 filsSolution:
     (filsUnivers:
          (saisirCardinal: '2'
           saisirReferentiel: 'Un jeu de 32 cartes'
           valider)
      filsConstruction:
```

```
          (filsSousPartie:
               (saisirCardinal: '1'
                fixerNbProprietes: '1 propriété'
                filsPropriete1:
                    (saisirAttribut: 'la hauteur'
                     saisirComparateur: 'est'
                     filsCompPred:
                         (saisirValeurCible: 'Dame'))
                valider: 'Ajouter l''Étape')
           creerInstance: 'Nouvelle Étape'
           saisirMultiplicite_AvecScript:
               [Noeud
                expression:
                    (faireProduit: 'Faire le Produit')]
           valider: 'Valider la Construction')
      valider: 'J''ai terminé !'))
```

Les scripts sont interprétés par la classe **`InterpreteurDeScript`**. Pour envoyer un script à une machine, faire

```
InterpreteurDeScript executer: <script> dansContexte: <machine>
```

Scripts de configuration de machines

Pour traduire les labels d'une machine en anglais, avant il fallait plonger dans le canvas, chercher les textes à remplacer, etc. Maintenant, on utilise un script de configuration de labels (qui est une autre sorte de script). Voici un exemple pour la machine CE :

```
(langue, anglais
 choixExercice:
     (choisir_xChoix: 'List of Exercises'
      annuler_xBtn: 'Change Exercise'
      valider_xBtn: 'Validate')
 filsSolution:
     (choixUnivers:
          (saisirReferentiel_Etiquette: 'elements chosen in'
           cadreUnivers: 'Universe'
           annuler_xBtn: 'Change Universe'
           saisirCardinal_Etiquette: 'I want subsets with'
           aide_xBtn: '?'
           valider_xBtn: 'Ok')
      valider_xBtn: 'I am done !'
      annuler_xBtn: 'Erase Solution'))
```

Pour demander à un interacteur ia de configurer ses labels suivant un script, faire :

```
ia changerLabels: script
```

Scripts de spécification de machines

Avant, pour tester les idées d'implémentation d'une machine, il fallait commencer tout de suite par implémenter la machine, ce qui était pénible à mettre au point et aboutissait à des confusions entre les fonctions purement conceptuelles et les fonctions purement interactives. De plus, les intentions et les choix d'interactivité n'étaient pas déclarés explicitement. Il en résultait des confusions et des oublis.

Il y a maintenant une possibilité de tester les fonctions conceptuelles sans implémenter la partie interactive, tout en explicitant certains choix d'interaction et en les simulant au travers de dialogues Smalltalk standard. On utilise pour cela des *scripts de spécification d'interaction* (nom provisoire).

En voici un exemple, pour la machine CE (extrait) :

```
(exercice:
   [selection
    choix: lesExercices],
     ()
 solution:
   [abstraction],
     (univers:
        [abstraction
         classe: UniversParties],
          (cardinalDesParties:
             [selection
              choix: lesCardinaux],
               ()
           referentiel:
             [selection
              choix: lesReferentiels],
               ())))
```

Pour executer un tel script, faire :

```
GT.TesteurInteraction executer: script dansContexte:
unObjetDuModele
```