Isi Engineering Process Publisher

Document de conception

Version 5.0

Superviseurs de projet : Claude Aubry
Bernard Cherbonneau
Responsable : Sandra Poulain

Historique des révisions

VERSION	DATE	DESCRIPTION DES EVOLUTIONS	AUTEURS
3.0	12/01/2004	Création	**Sandra POULAIN & Nicolas MICHEL**
3.1	27/01/2004	Ajout du paragraphe Affichage des erreurs et suivi de l'avancement des tâches	**Sandra POULAIN**
3.2	31/01/2004	Ajout du paragraphe sur la gestion du Undo/Redo.	**Sylvain LAVALLEY**
5.0	20/03/2004	Ajout du paragraphe sur la gestion du Référentiel	**Sylvain LAVALLEY**

Table des matières

Introduction

Ce document complète le document d'architecture et permet de présenter de manière détaillée les éléments de conception du projet IEPP.

Les paquetages

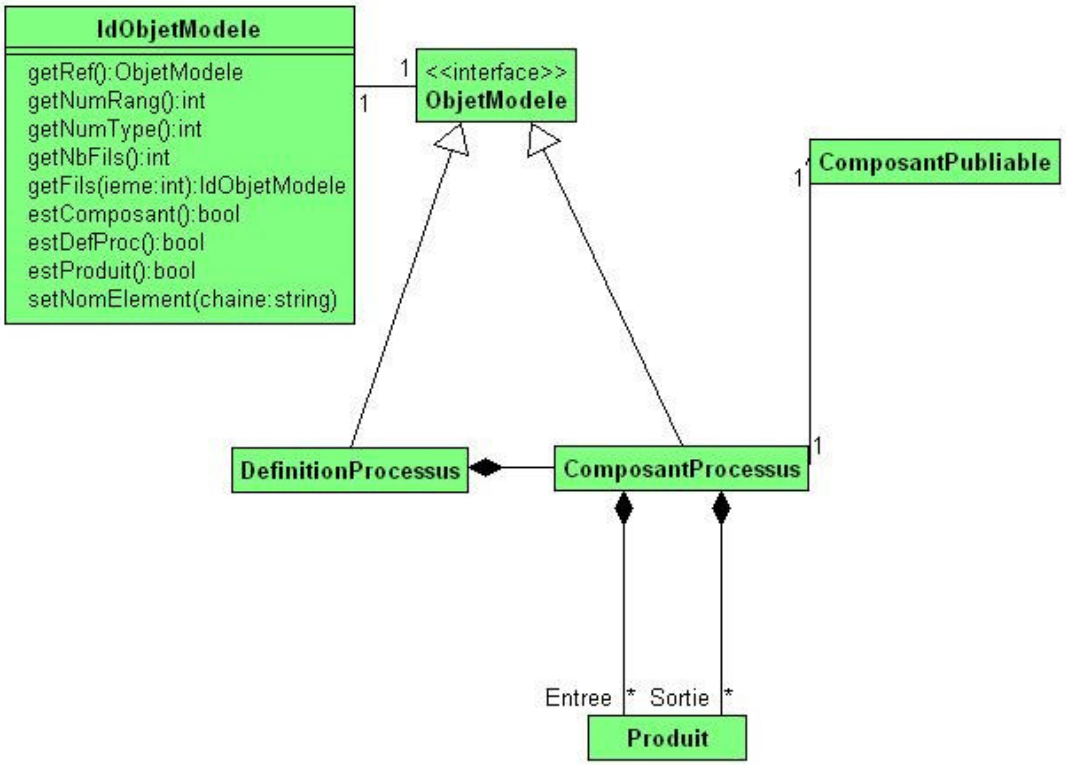
Le paquetage principal est iepp il contient les classes principales Application et Projet. Le sous paquetage application correspond à la couche traitement, ui correspond à la couche de présentation, domaine correspond à la couche des données métier et infrastructure correspond à la couche d'infrastructure, elle contient des classes, des API réutilisables comme jsx. Le paquetage annexe util contient un ensemble de classes utilitaires réutilisées (gestionnaire d'images, classes conteneurs, …).

Application du modèle MVC

Les modèles

Le modèle métier

Les classes métier propres à l'outil sont ComposantProcessus et DefinitionProcessus.



IdModèles :

Les IdModèles sont des objets « masquant » les objets du domaine, et à travers lesquels les autres classes doivent passer pour accéder aux données. Ainsi, lorsque l'arbre ou le graphe demandent des composants, produits, etc…, ce qui leur est en fait retourné est un objet IdModèle. Dans ces objets est définie une interface commune : ainsi tous les objets du domaine répondent à un ensemble commun d'opérations, et il n'est plus nécessaire de coder spécifiquement pour un type d'objet du modèle. Par exemple, pour l'arbre, qui affiche des composants, des produits, …, on utilise toujours les mêmes opérations. Ceci permet par exemple d'utiliser comme objets du domaine des classes extérieures (en codant uniquement l' IdModèle adapté).

Les IdObjetModele permettent d'implémenter le modèle de la Façade. Ce modèle consiste à manipuler un sous-système à travers un objet unique, qui joue le rôle d'interface entre le sous-système et le reste de l'application. Ce rôle est ici joué par la classe ComposantProcessus, qui encapsule en quelque sorte les données métiers issues de l'outil de modélisation (les détails de la modélisation des composants : produits, rôles, activités, leur représentation et surtout l'implémentation des liens entre eux).

Pour accéder aux éléments de modélisation, il faut donc passer par la classe ComposantProcessus, mais il reste nécessaire d'identifier ces élements, pour leur associer des vues par exemple. Les IdObjetModele permettent d'identifier les objets

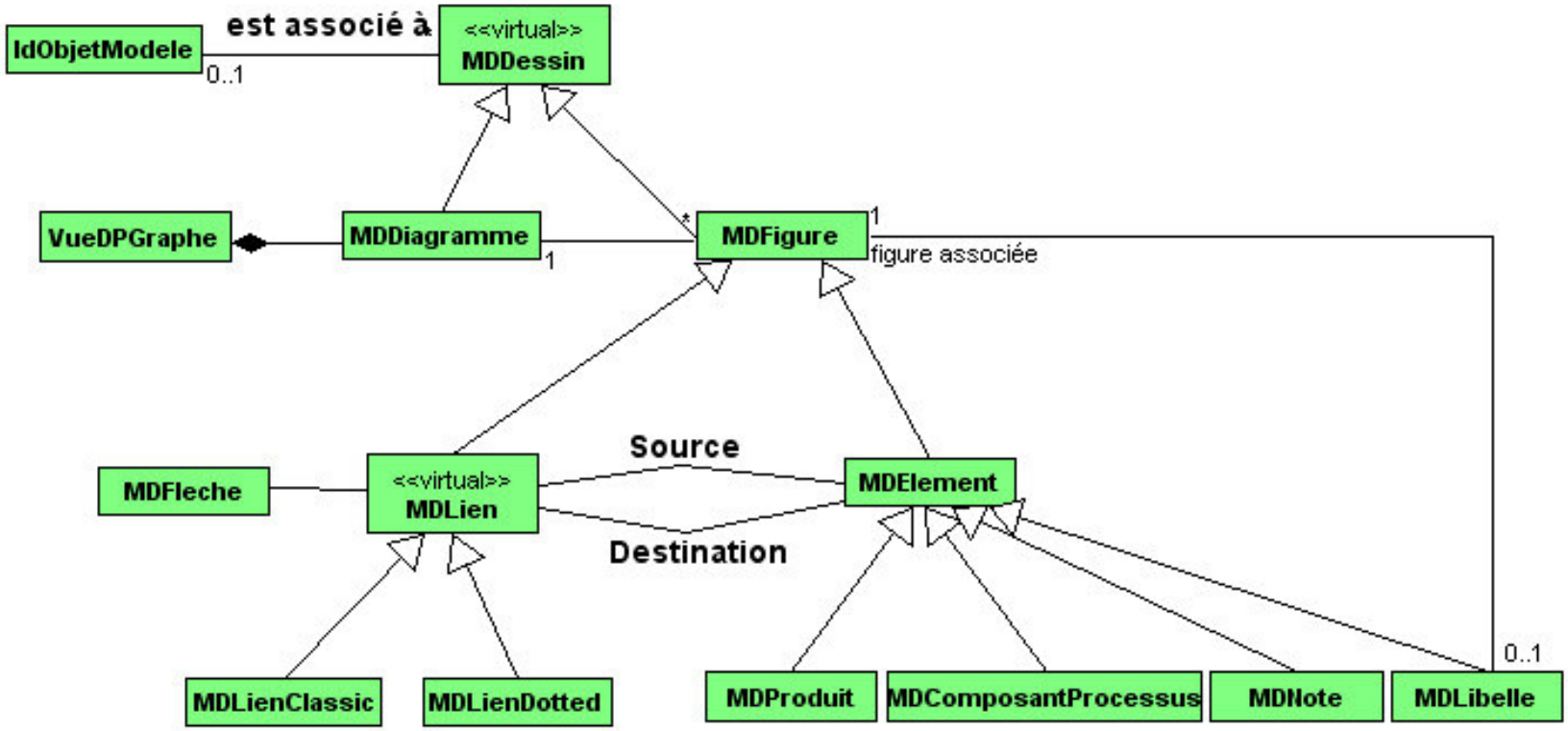
du modèle d'une manière "abstraite" : les autres objets de l'application ne peuvent qu'appeler des méthodes sur ces IdObjetModele, qui se chargent d'invoquer la méthode correspondante du ComposantProcessus, en lui indiquant quel élément de modèle est concerné (les IdObjetModele sont créés et réinterprétés par les ComposantProcessus).

Composant publiable :

Le composant publiable est la classe représentant le composant et ses éléments de présentation. Cette classe est récupérée directement de l'outil 2. Seul notre ComposantProcessus connaît cette classe (où il va « chercher » ses informations) ; et notre application s'adresse uniquement à notre classe ComposantProcessus. Ainsi, si la classe ComposantPubliable est modifiée, il suffira de ré-écrire la classe ComposantProcessus pour prendre en compte les changements dans toute l'application.

Les modèles de dessin

Les modèles de dessin ont été conçus pour les diagrammes d'assemblage de composants de processus. Toutes ces classes sont préfixées de « MD » pour « Modèle de Dessin ».



MDDessin est juste une classe abstraite qui indique que tous ses descendants seront sérialisables.

MDElément possède des attributs dont héritent tous les éléments :

- fillcolor : couleur de remplissage
- linecolor : couleur de lignes et du texte
- x,y : coordonnées de l'élément, par rapport au diagramme qui le contient
- largeur, hauteur : dimensions de l'élément
- largeurMini, hauteurMini : dimensions minimales de l'élément

MDLien possède des attributs dont héritent tous les liens :

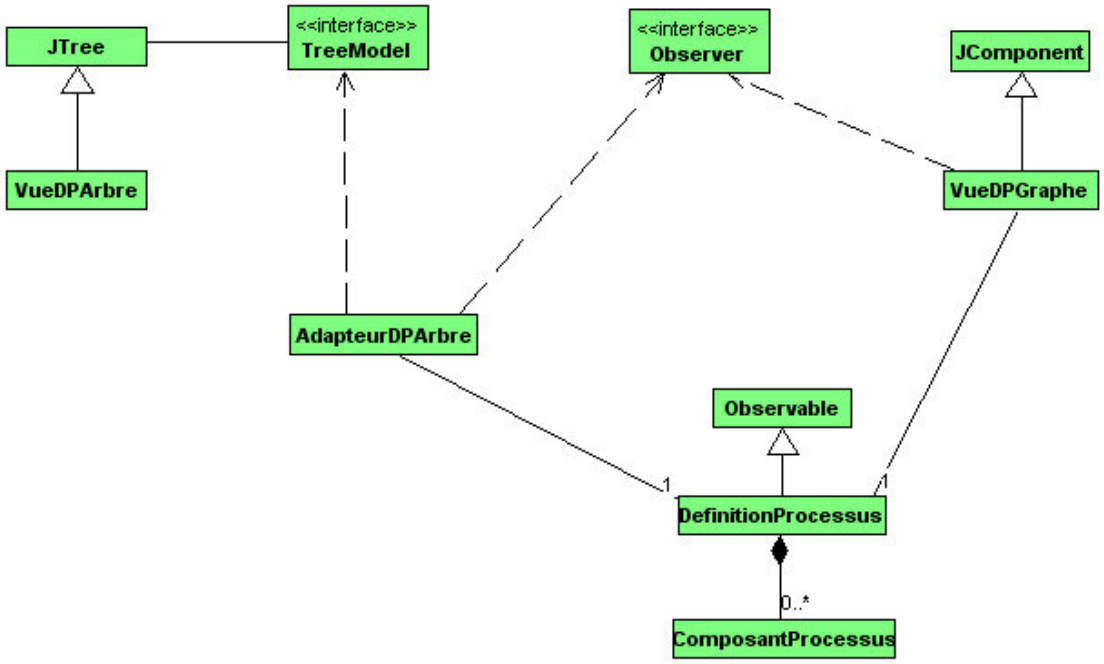
- couleur : couleur du lien
- source : source du lien (MDElement)
- destination : destination du lien (MDElement)
- pointsAncrage : liste des points d'ancrage du lien (nécessaire pour faire des liens brisés)

Seuls MDProduit et MDComposantProcessus ont un IDObjetModele associé, leur permettant de conserver le nom de l'élément (produit ou composantProcessus).

Pour ajouter ou supprimer un élément du diagramme, il faut appeler la méthode ajouterModeleFigure/supprimerModeleFigure avec le MDFigure que l'on désire ajouter/supprimer.

IEPP a un modèle de donnée et deux vues correspondant à ces données. Une vue arborescente et une vue sous forme de diagramme. La vue est créée à partir de données et affiche ces données sous une certaine forme. Les données sont observées par la vue, c'est-à-dire que lorsque une modification est effectuée sur les données, on notifie immédiatement la vue et celle-ci se rafraîchit pour afficher les données mises à jour. Si une modification des données est demandée à partir de la vue, le controleur prend en charge la modification des données.

Nous avons donc deux classes correspondant aux deux vues de la définition de processus à afficher : VueDPArbre et VueDPGraphe.



DefinitionProcessus possède une méthode « maj() » qui indique à tous ces observeurs que la définition de processus ou les éléments qui la composent ont été modifiés.

Le modèle central doit être affichée sous forme d'arbre et sous forme de graphe. Or, ces deux représentations graphiques ne présentent pas exactement les mêmes choses. Il leur faut donc deux modèles différents. Mais alors il y aurait des informations redondantes, et des problèmes de synchronisation des modèles, …

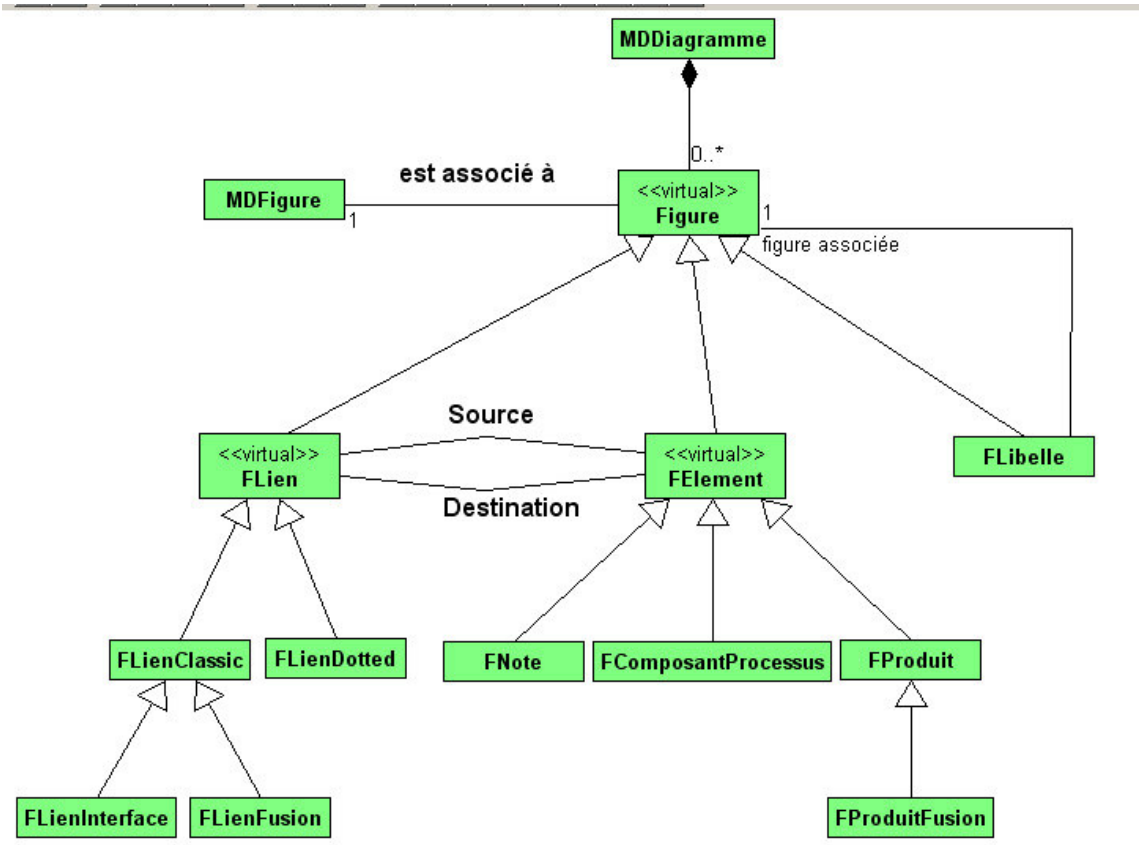
Pour résoudre ce problème, nous avons donc créé pour chaque représentation graphique un adaptateur vers le modèle unique (AdapteurDPGraphe). Cet adaptateur renvoie les données sous la forme attendue par la représentation graphique, ne renvoie que ce qu'il faut afficher, etc… en interrogeant le modèle central de l'application. Les données qui sont renvoyées par l'adapteur sont des IdObjetModeles qui sont traités et intérroger par les classes de rendus de l'arbre.

Les Vues

Chaque vue est liée à un modèle (attribut privé) et possède les méthodes « Modele getModele() » et setModele(Modele) » permettent respectivement de récupérer et de fixer le modèle d'une vue.

Les figures

Les figures sont les représentations graphiques des modèles de dessin. C'est le diagramme lui-même qui appelle la fonction paint(Graphics g) de chacune de ses figures, en leur passant en paramètre son contexte graphique g, afin que celles-ci puissent y dessiner directement.



La classe mère est Figure. Tous les noms de classes sont préfixés de « F ».

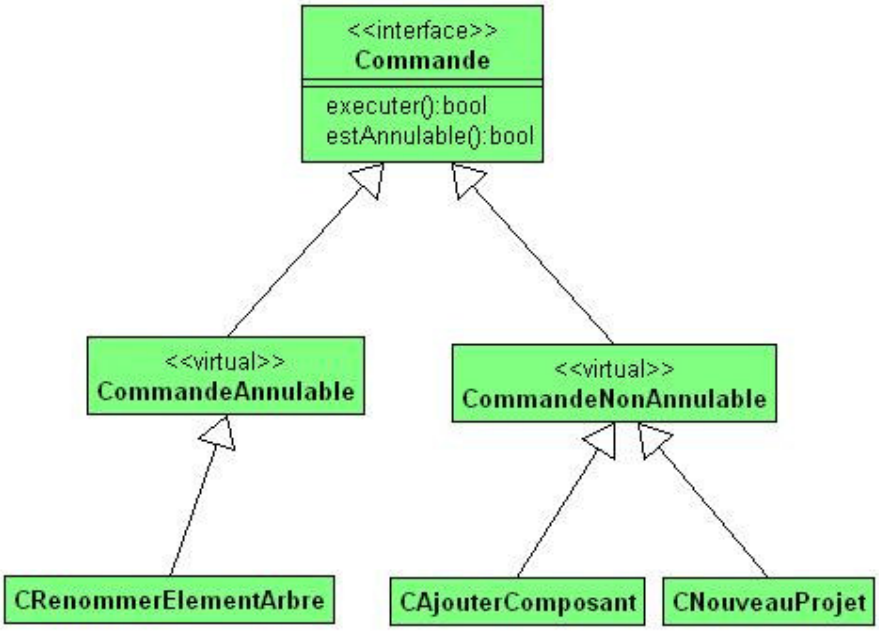
La classe figure possède une méthode doOnRightClick() à implémenter qui est appelée lorsque l'utilisateur effectue un double click sur la figure. On affichera la fenêtre de propriété de l'élément cliqué.

La classe FElement possède plusieurs méthodes qui permettent de différencier la représentation graphique d'un élément fixe et un élément qui est en train d'être déplacé.

Chaque Figure a une méthode displayHandles(Graphics g) permettant d'afficher ses handles (les poignées). On peut grâce à ces poignées redimensionner une figure ou distinguer une figure sélectionnée (les poignées s'affichent) d'une figure non sélectionnée.

FlienInterface et FLienFusion descendent tous les deux de FlienClassic et permettent respectivement de créer des liens non supprimables (on ne peut pas supprimer une interface d'un composant processus) et des liens supprimables.

Les commandes



Les commandes sont des classes du paquetage application, elles effectuent chacune un traitement bien défini. On distingue deux sortes de commandes : les commandes annulables et les commandes non annulables.

Pour appeler une commande, on créer une instance de la commande et on appeler la méthode executer(). Celle-ci indique si l'exécution de la commande a pu être finie (erreur survenue) ou pas.

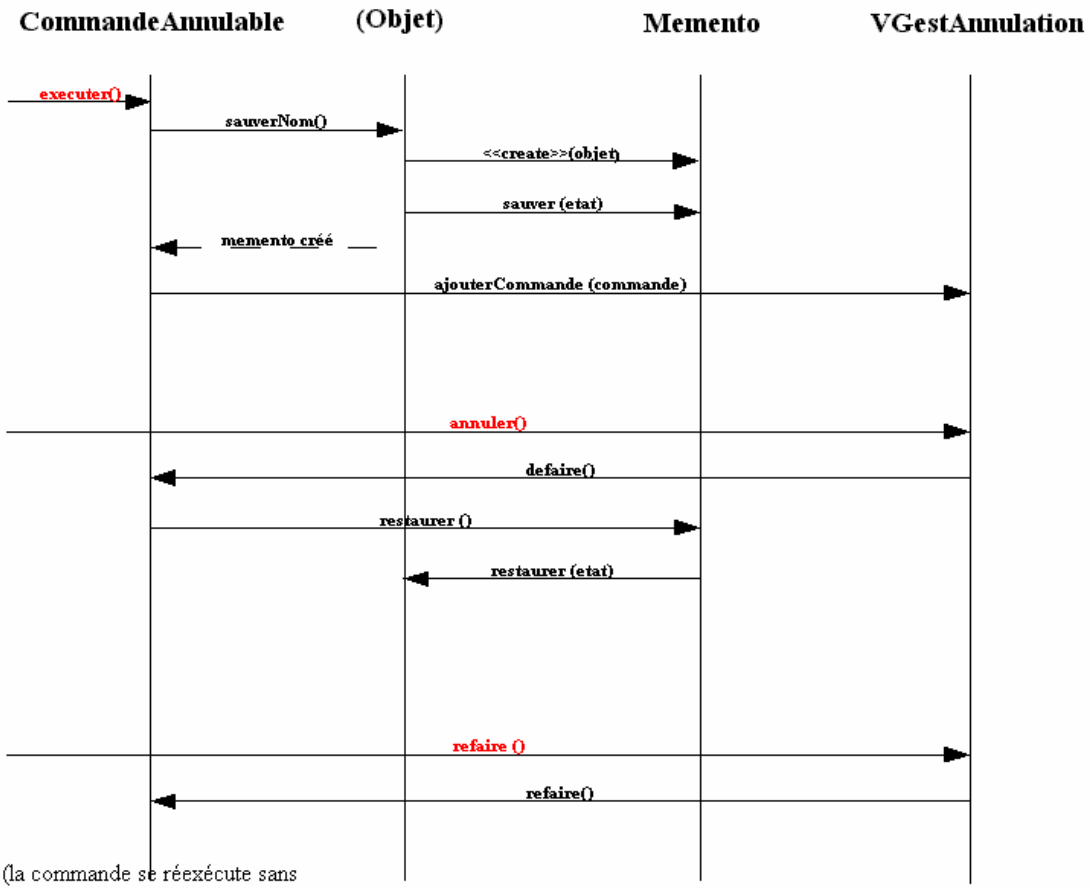
Tous les noms de commandes sont préfixés de « C ».

Annuler et refaire des commandes

Cette fonctionnalité permet d'annuler des commandes qui viennent d'être exécuteés et de refaire des commandes qui viennent d'être annulées. Elle se base sur deux éléments importants :

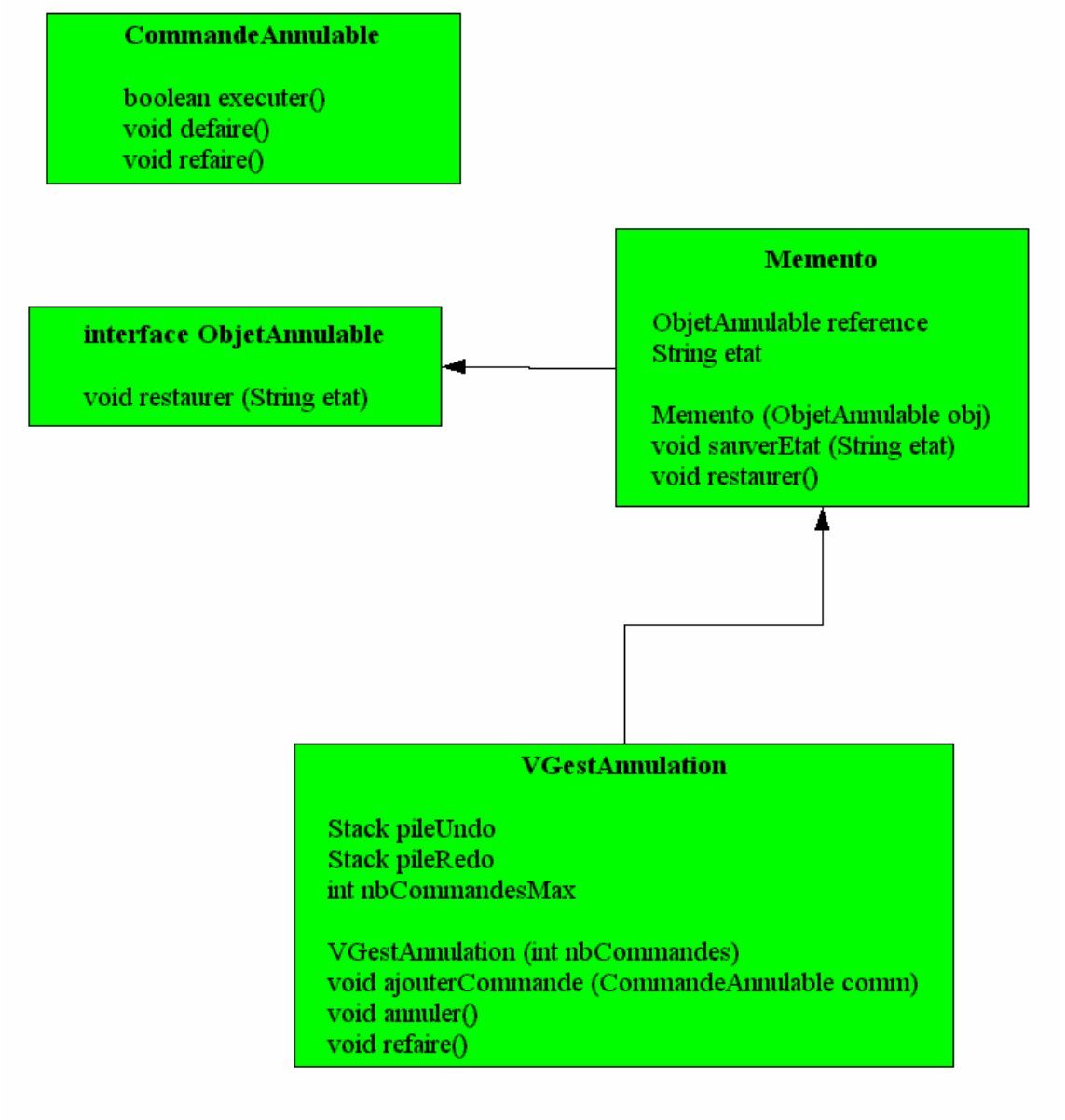
- des objets de mémorisation capables de stocker l'état d'autres objets pour pouvoir les restaurer par la suite
- un « gestionnaire d'annulation », qui permet de mémoriser les état stockés successivement pour pouvoir revenir à l'état précédent.

Une commande qui veut modifier des objets leur demande de stocker les informations modifiées dans un objet de stockage (Memento). Puis elle range cet objet dans le gestionnaire d'annulation. Lorsque l'utilisateur choisit d'annuler la dernière commande appliquée, le gestionnaire demande au Memento correspondant d'effectuer la restauration. Celui-ci contacte l'objet qui l'a créé et lui rend les informations stockées, de façon à ce que l'objet puisse lui-même se restaurer. Le système du memento permet ainsi d'assurer qu'aucun autre objet n'a accès aux données privées de l'objet stocké.



(la commande se réexécute sans rien sauvegarder ni s'inscrire dans le gestionnaire d'annulation)

Diagramme de séquence executer-annuler-refaire
(changement de nom d'un objet, un composant par exemple)
* Le système fonctionne avec des objets supprimés, car l'objet «détruit » est mémorisé par le Memento, référencé par la Commande. Lorsque la comande est supprimée du gestionnaire d'annulation, l'objet est réellement perdu.
* Les objets sauvegardent comme ils veulent les paramètres demandés. Ils peuvent éventuellement avoir une méthode unique qui sauvegarde tout. L'important est qu'ils puissent restaurer ensuite un état passé.
* L'objet de gestion des annulation est référencé par le projet.
* Le gestionnaire d'annulation gère deux piles de comandes : celles effectuées et celles annulées, comme indiqué dans le modèle de conception Undo/redo


CommandeAnnulable
boolean executer()
void defaire()
void refaire()
Memento
ObjetAnnulable reference
String etat
Memento (ObjetAnnulable obj)
void sauverEtat (String etat)
void restaurer()
interface ObjetAnnulable
void restaurer (String etat)
VGestAnnulation
Stack pileUndo
Stack pileRedo
int nbCommandesMax
VGestAnnulation (int nbCommandes)
void ajouterCommande (CommandeAnnulable comm)
void annuler()
void refaire()

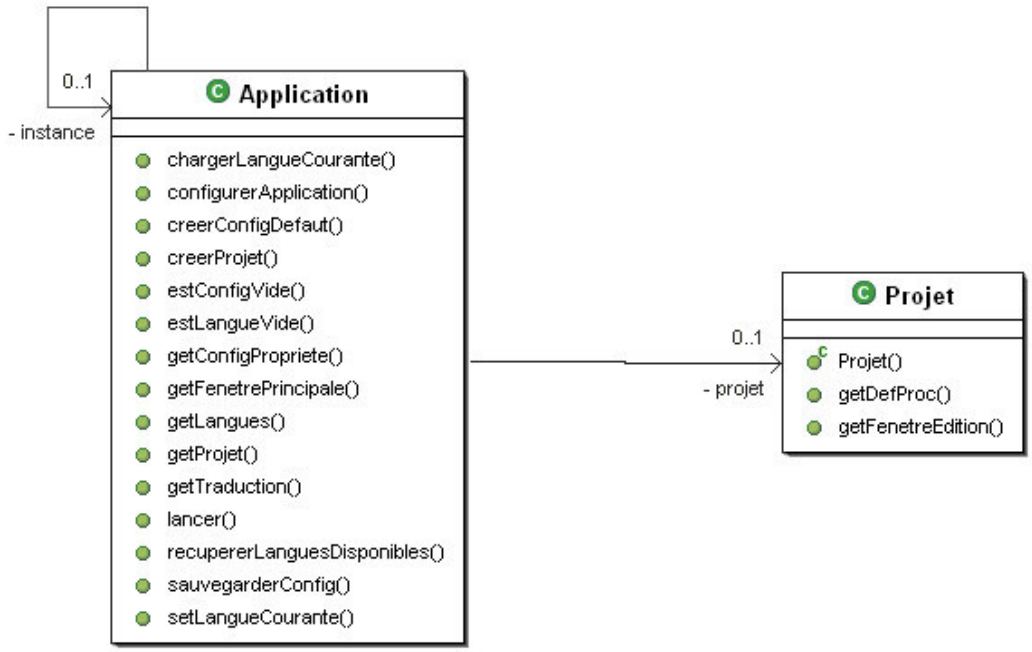
Internationalisation et configuration de l'outil

Les fichiers de langue et le fichier de configuration (application.properties) sont gérés de la même manière. Chaque ligne de ces fichiers est de la forme :

Cle=Valeur

Ces données sont récupérées via la classe Properties (deux attributs privés de la classe principale Application), le chargement des fichiers se fait grâce à la méthode configurerApplication() et chargerLangueCourante(). On accède à une donnée grâce à la méthode getConfigPropriete(s) ou getTraduction(s) où « s » est la Clé dont on recherche la valeur associée. Lorsque l'utilisateur demande un changement dans la configuration de l'outil au travers de boîte de dialogue par exemple, la méthode sauvegarderConfig() est appelée et sauvegarde dans le fichier application.properties tous les couples cle/valeur nécessaires. On ne peut pas modifier la traduction d'un mot dans l'application, il faudra modifier « à la main » en éditant le fichier .lng correspondant.

Lorsque l'on essaye d'accéder à une clé qui n'existe pas, une valeur par défaut est renvoyée. Cette valeur est définie par la constante CHAINEERREUR de la classe Application.



Pour appeler les méthodes citées, il faut tout d'abord récupérer l'instance de l'application courante. L'Application étant implémentée sous la forme d'un singleton, il n'y a qu'une seule instance de l'application à un moment donné. La méthode statique getApplication() permet de récupérer cette instance. On peut ensuite appliquer ces méthodes. L'instance de l'Application est créée grâce à la méthode statique creerApplication().

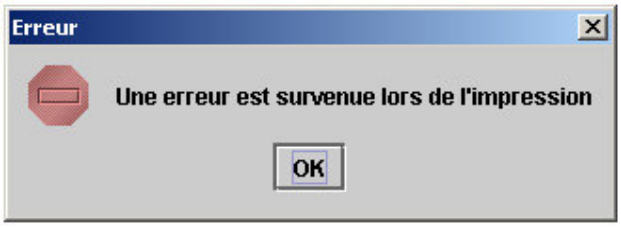
Pour chaque élément traductible il faut créer dans le fichier de langue un couple clé/valeur. Lorsque l'on créé cet élément (par exemple un JButton) il faut utiliser cette méthode **: Application.getApplication().getTraduction("clé").** Il faut ensuite prévoir une méthode rafraîchir() qui permet de modifier l'élément dès que l'utilisateur demande le changement de la langue courante.

Affichage des erreurs

La classe ErrorManager permet de gérer les messages d'erreur à afficher à l'utilisateur grâce à sa méthode `display(String title, String msg)` où title est le titre de la boîte de dialogue qui s'affiche et msg le message s'affichant dans la boîte de dialogue. La méthode effectue directement la traduction de ces paramètres, il faut donc appeler la méthode comme suit :

```
ErrorManager.getInstance().display("ERR", "ERR_Imprimante");
```

ERR et ERR_Imprimante sont des clés de traduction qui ont été définies dans les fichiers langues dans le paragraphe dédié aux erreurs. On obtient :



La méthode `display(Throwable t)` sera utilisée lorsque la raison de l'erreur/echec est inconnu. Le message affiché sera le message système créé par la JVM. On aura en général :

```
try
{
      // instructions qui lancent une exception
}
catch (Throwable t)
{
      t.printStackTrace();
      ErrorManager.getInstance().display(t);
}
```

Suivi des tâches conséquentes

Plusieurs tâches de l'outil peuvent prendre du temps, par exemple le chargement d'un composant publiable dépend du volûme de contenu du composant. Or ces tâches en général bloque l'outil et laisse penser à l'utilisateur que le programme ne fonctionne plus. Un système a dont été mis en place pour afficher l'avancement de la tâche et des commentaires liés à la tâche.

Les classes MonitoredTaskBase (classe abstraite) et TaskMonitorDialog ont été réutilisées pour mettre au point ce système.

- MonitoredTaskBase : classe abstraite représentant une tâche dont l'avancement peut être visualisé. Elle contient une méthode go() qui appelle la méthode `processingTask()` de la tâche à visualiser. La méthode done() indique si la tâche est terminée ou non.
- TaskMonitorDialog : boîte de dialogue affichée lorsque la tâche commence, elle contient une barre de progression qui indique l'avancée de la tâche et un panneau textuel dans lequel s'affiche des commentaires chaque fois qu'une partie de la tâche est effectuée.

Il faut que la classe qui effectue la tâche extends MonitoredTaskBase. Il y a donc plusieurs méthodes à implémenter :

```
protected Object processingTask()
{
	// instructions de la tâche
	return null;
}

public void setTask( TaskMonitorDialog task )
{
	this.mTask = task;
}

/**
* Print a new message to the TaskMonitorDialog
*
* @param msg
*/
private void print( String msg )
{
	setMessage(msg);
	if( mTask != null )
	{
		mTask.forceRefresh();
	}
}
```

La méthode print() affiche dans le panneau de texte de la boîte de dialogue, le message msg. On peut dont suivre l'avance de la tâche grâce aux commentaires affichés. Il faut appeler la méthode print() chaque fois qu'une instruction significative a été terminée.

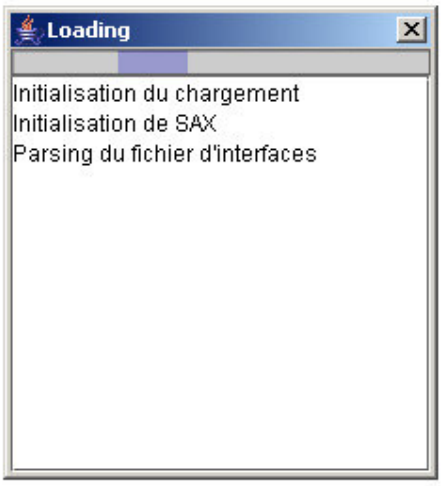
La classe doit aussi avoir un attribut représentant la boîte de dialogue à afficher :

```
/**
* Boite de dialogue permettant d'afficher l'avancement des tâches
*/
private TaskMonitorDialog mTask = null;
```

La classe qui appelle le long traitement (en général la classe Commande, par exemple la commande CajouterComposant appelle le traitement chargercomposant de ChargeurXML) va se charger du lancement du traitement comme suit :

```
/**
* Dialog permettant de visualiser l'avancement du chargement du composant
*/
private TaskMonitorDialog dialogAvancee = null;

…

// affiche la boîte d'avancement
this.dialogAvancee = new
TaskMonitorDialog(Application.getApplication().getFenetrePrincipale(),
chargeur);

this.dialogAvancee.setTitle(Application.getApplication().getTraduction("Cha
rgement"));

// chargeur est la tâche à surveiller descend de MonitoredTaskBase
chargeur.setTask(this.dialogAvancee);
// affiche et lance le traitement
this.dialogAvancee.show();
```

On obtient ceci :



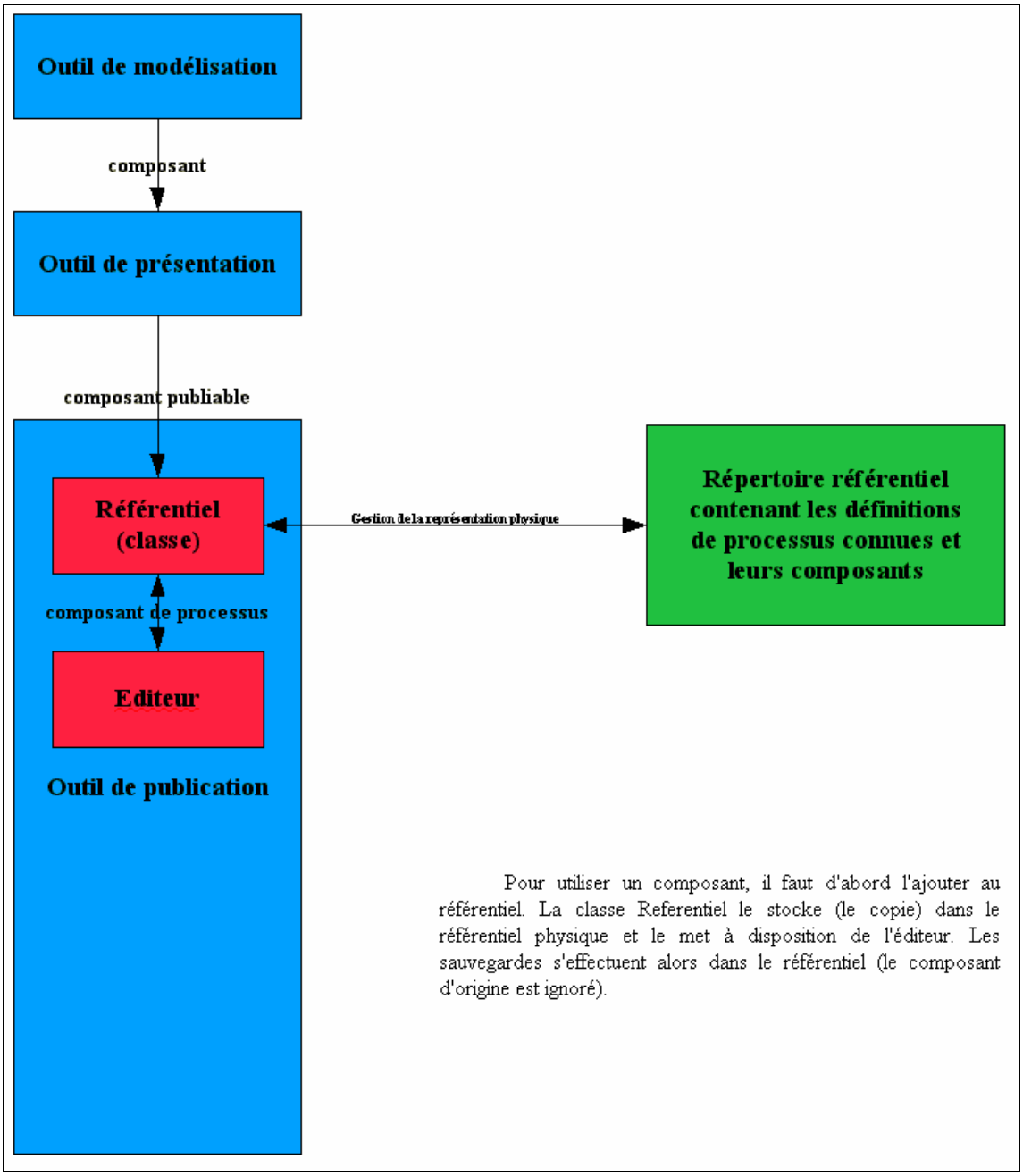
Gestion du référentiel

I Présentation du référentiel

L'application reçoit en entrée des composants, qui sont chargés dans des définitions de processus pour être assemblés en un processus. Ces composants ne peuvent pas être chargés directement dans une définition de processus. Ils doivent passer par le référentiel, une structure regroupant les composants et définitions de processus connus par l'application.

L'application gère en fait plusieurs référentiels, chacun regroupant un certain nombre de composants et de définitions de processus. L'utilisateur choisit un référentiel de travail et peut y créer des processus et y charger des composants. Lorsqu'il édite un processus, il peut y ajouter des composants choisis dans le référentiel.

II Fonctionnement technique



Structure logique, structure physique :

Le référentiel est une zone physique (un répertoire sur le disque) ayant une représentation logique dans l'application (la classe Referentiel) chargée de la gérer. Toute intervention sur la structure physique doit se faire à travers la classe Referentiel.

La classe Referentiel est en quelque sorte une vue de la structure physique pour l'application : l'outil voit le référentiel à travers la classe qui le représente et peut lui demander de charger des composants, d'enregistrer des processus..., sans se soucier de la réalisation physique de ces actions.

Arborescence :

Les référentiels sont stockés dans le répertoire iepp/Referentiels. L'arborescence d'un référentiel est organisée comme suit (les flèches désignant les fichiers) :

<Nomreferentiel>
- => <Nomreferentiel>.ref [index]
- Composants
  - 1
    - => <Nom composant> [fichier composant]
  - 2
    - => <Nom composant> [fichier composant]
- DP
  - 3
    - => <Nom DP> [fichier processus]
- Presentations
  - 4
    - => <Nom présentation> [fichier présentation]
  - 5
    - => <Nom présentation> [fichier présentation]

Le référentiel est donc enregistré dans un répertoire qui porte son nom contenant un fichier d'index (contenant la liste des éléments présents dans le référentiel ainsi que le dernier id attribué à un élément) et 3 sous-répertoires, contenant respectivement les composants, définitions de processus et paquetages de présentation. Chacun des éléments se trouvant dans l'un de ces sous-répertoires est lui-même un répertoire ayant pour nom l'identifiant de l'élément et contenant le fichier concerné (composant, ...).

Représentation dans l'interface :

La classe Referentiel mémorise un arbre représentant l'arborescence physique du référentiel (les répertoires sur le disque). Cet arbre contient des objets de type ElementReferentiel, qui représentent les différents répertoires, composants, définitions de processus, ... L'arbre graphique représentant le référentiel est une vue de la classe référentiel (de l'instance courante en fait), mise à jour dès que les éléments de cette classe sont modifiés. Cette relation modèle/vue est implémentée par la classe Observable et l'interface Observer de Java, qui se chargent d'une grande partie du travail (enregistrement et gestion des vues sur un

modèle principalement). Il ne reste à l'observateur qu'à implémenter la méthode de mise à jour de la vue en cas de changement des données observées.

Identifiants :

Chaque élément contenu dans le référentiel possède un identifiant unique, attribué lors de sa création (son entrée dans le référentiel). Lorsqu'un élément du référentiel (composant, processus, paquetage de présentation) est chargé en mémoire, le référentiel mémorise dans des tables de hachage (Hashmap) l'association entre l'identifiant et la référence de l'objet en mémoire (dans les deux sens : deux tables d'association). Ceci permet de savoir si un objet d'identifiant donné est déjà chargé en mémoire, de pouvoir le contacter et inversement de retrouver l'identifiant d'un objet mémorisé (pour effectuer des actions le concernant dans le référentiel). De même, lorsqu'un élément est retiré de la mémoire, l'association est supprimée.

Des méthodes du référentiel permettent de gérer ces tables, mais elles sont appelées par plusieurs commandes dans le code, dont voici la liste :

- Ajouter si on charge un composant (vide ou non) Referentiel.chargerComposant
- Ajouter si on crée un composant (vide) Referentiel.chargerComposant
- Ajouter si on crée une DP au démarrage CNouveauProjet
- Ajouter (après retrait des autres) si on crée une DP CNouveauProjet
- Ajouter (DP + composants) si on charge une DP Referentiel
- Retirer si on supprime un composant (vide ou non) CRetirerComposant
- Retirer tout si on ferme une DP CFermerProjet
- Ne pas prendre en compte pour les présentations (ne sont pas ajoutées à la DP)