Master Informatique Spécialité SAR
MI041 PSAR - Mai 2014

Manipulation de modèles comportementaux pour les lignes de produits

Etudiants :

Oussema EL ABED

Racha AHMAD

Encadrant :

M. Tewfik ZIADI

Année universitaire 2013/2014

Sommaire

Liste des figures

Introduction

Déplacer le développement de logiciels à une nouvelle qualité de la production industrielle gagne aujourd'hui une place importante dans la recherche informatique. Une approche indispensable est de savoir comment peut-on gérer la variabilité logicielle et faire face aux différents problèmes de développement de la famille de programme.

Plusieurs solutions concernant les différentes phases du processus de développement de logiciels ont été proposées, une idée prometteuse est la transposition efficace des chaines de production industrielles au monde logiciel pour réaliser la dérivation automatique des produits et produire plusieurs versions d'une application.

L'idée de base de nos travaux consiste à :

- Proposer des mécanismes pour la spécification de la variabilité.
- Automatiser la dérivation de produits en utilisant les transformations de programmes.

1 Etat de l'art

1.1 Lignes des produits logiciels :

Une ligne de produit est une famille de logiciels partageant des propriétés communes mais qui sont différentiés aussi par d'autres caractéristiques.

L'ingénierie des lignes de produits logiciels (LdP) est une approche récente du génie logiciel qui est une transposition des chaînes de production au monde du logiciel.

Le principe est de minimiser les coûts de construction de logiciels dans un domaine d'application particulier en ne développant plus chaque logiciel séparément, mais plutôt en le concevant à partir d'éléments réutilisables. Le principe de l'approche LdP réside dans la conception d'une architecture permettant de définir plusieurs logiciels à la fois.

Les membres d'une ligne de produits sont caractérisés par leurs points communs, mais aussi par leurs différences (variabilité). La gestion de cette variabilité est l'une des activités clé des lignes de produits. Une autre activité dans l'ingénierie des LdP concerne la construction d'un produit logiciel (on parle aussi de dérivation de produit) qui consiste en partie à figer certains choix vis-à-vis de la variabilité définie dans la ligne de produits pour générer un produit spécifique.

Dans ce projet nous nous intéressons aux modèles de lignes de produits logiciels ou la variabilité est spécifiée dans des modèles comportementaux (automates, diagrammes de séquence).

Plusieurs outils et environnements (comme le plugin Eclipse FeatureIDE) ont été proposés ces dernières années pour la manipulation des LdPs.

Dans ce qui suit, on présent les outils utilisé dans notre projet.

1.2 Les outils utilisés :

Pour la réalisation du projet on a utilisé principalement deux outils: L'outil FeatureIDE et L'éditeur du diagramme de séquence SDedit.

1.2.1 L'outil FeatureIDE :

Les membres de la LdP diffèrent par un ensemble de caractéristiques : *« features »*. *« Feature »* est une caractéristique d'un logiciel définie par les experts de domaine comme importante pour distinguer les différents produits.

FeatureIDE est un IDE *(integrated development environment*) basé sur Eclipse qui soutient toutes les phases de développement de logiciels *« feature-oriented »* pour le développement des LDP: analyse de domaine, la mise en œuvre de domaine, l'analyse des besoins, et la production de logiciels.

L'environnement FeatureIDE qui est intégré à Eclipse propose donc un cadre permettant d'implémenter ce qu'on a appelé "Ligne de Produits".

Différentes techniques de mise en œuvre LDP sont y intégrées : *feature-oriented programming (FOP), aspect-oriented programming (AOP), delta-oriented programming (DOP)* et préprocesseurs.

Un autre outil indispensable pour notre projet est l'éditeur SDedit.

1.2.2 L'editeur du diagramme de séquence SDedit

SDedit est un outil de création de diagrammes de séquence UML à partir de descriptions textuelles des objets et des messages. Il est aussi facile à utiliser car il suit une syntaxe très simple, et il est très utile pour créer et visualiser les diagrammes que l'on spécifie.

Conception

2 Conception

2.1 Conception fonctionnelle

2.1.1 Editeur des fichiers sdcombin

2.1.1.1 Fonctionnalité

Ce langage fournit aux développeurs un moyen de définir la combinaison de déroulement d'exécution entre les features du produit générer. Cette combinaison définie l'algorithme de séquencement que le diagramme de séquence final doit respecter pour être correct et satisfait la chronologie d'exécution du produit.

2.1.1.2 La syntaxe

La combinaison faite avec ce langage doit être stockée dans un fichier avec l'extension «.sdc ». Ce fichier est composé de deux sections, la section Feature {...}, et la section Combination {...}.

- ***la section Features :*** La section définit les noms des features qui vont intéragir dans la combinaison. La syntaxe de la section est définie comme suit :

```
Features {
            feature1
            feature2
            …
            featureN
}
```

- ***la section Combination :*** Elle définit l'algorithme de séquencement entre les features dans le diagramme de séquence final. Dans cette section on ne peut pas utiliser des features autres que ce sont définie dans la section précédente. La syntaxe de la section est définie comme suit :

```
Combination {
            …
            loop ["condition"] {….}
            …
            seq
            …
            alt ["condition"] {… else…}
            …
            opt ["condition"] {….}
            …
}
```

On remarque qu'on peut utiliser plusieurs types d'expression : les boucles, les alternatives, les options et le séquencement.

- ***les sequences* :** elle encode un simple enchainement, ce sont les enchainements simples entre les feature.
  Exemple, on veut le feature1 s'exécute avant le feature2. Elle se traduit par :

```
…
 Feature1
        seq
 Feature2
        seq
        ….
        seq
 FeatureN
……
```

- ***l'alternative :*** La définition de l'alternative comporte la définition de la condition, l'alternative et l'alternant.

  Remarque : On peut faire l'alternative sans l'alternant.

  Exemple :

```
alt ["appuyer_Annuler  == True"] {

            (contexte alternative)

            else

            (contexte alternant)

}
```

- ***la boucle :*** La définition de la boucle comporte la définition de la condition et le contexte de la boucle.

```
loop ["appuyer_Annuler != True"] {

(contexte du loop)

}
```

- ***l'option :*** La définition de l'option comporte la définition de la condition et le contexte de l''option.

```
 opt ["appuyer_Annuler != True"] {

(contexte de l'option)
 }
```

2.1.1.3 Le méta modèle du langage

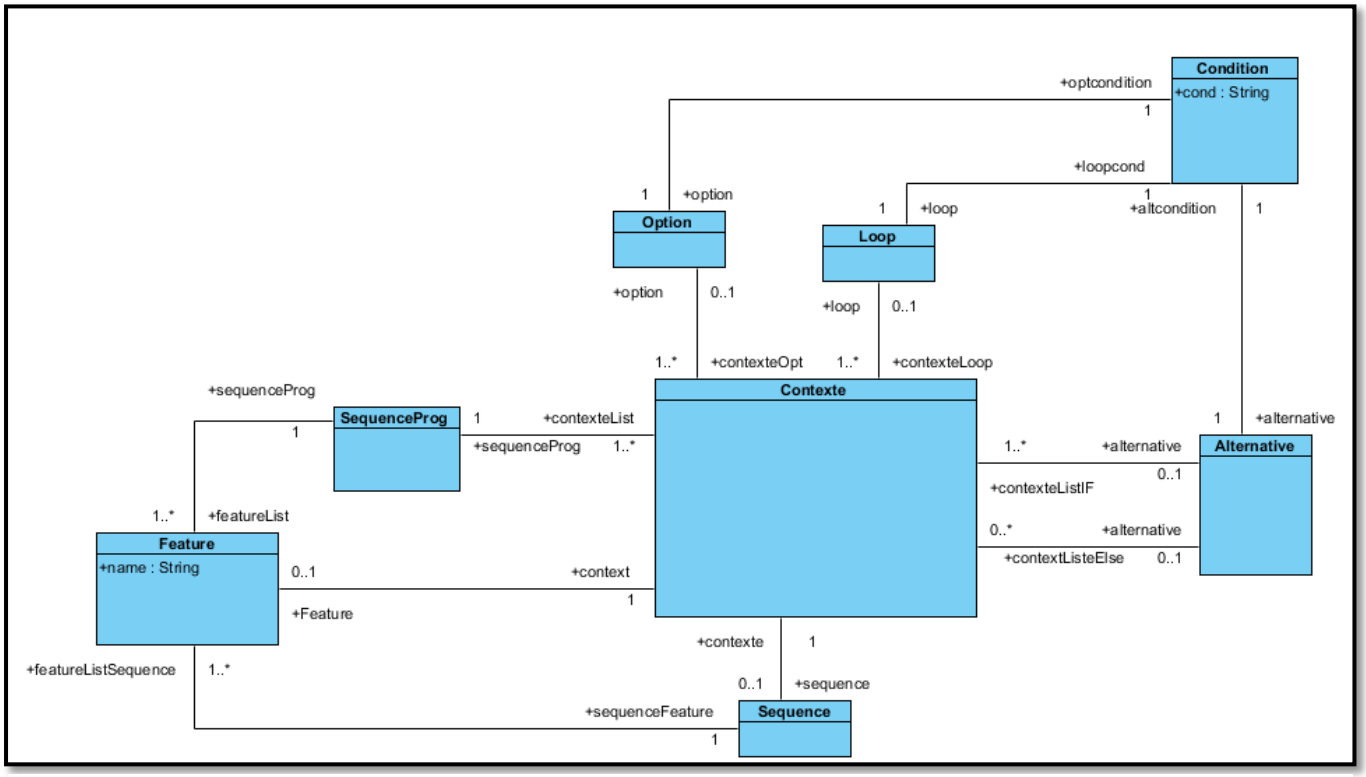


Figure 1 : Méta modèle du langage SdCombin

2.1.2 Editeur des fichiers sdedit

2.1.2.1 Les Fonctionnalités

C'est un éditeur pour le langage spécifique de l'application Sdedit. Il fournit aux développeurs une syntaxe pour définir des objets et les interactions entre ces derniers(les événements, les boucles, les alternatives...).

2.1.2.2 La syntaxe

- **la section de définitions des objets**

  a. ***Définition des objets***

  `<name>:<Type>[<flags>]`

  `name`: nom de l'objet

  `Type`: le type de l'objet

  `flags`: « a » pour un objet anonymes, « p » pour un objet qui se comporte comme un acteur, mais il est représenté par une boîte avec une bordure épaisse, pas par un chiffre, « x » l'objet est détruit quand il envoi sa dernière réponse, noté par une croix en dessous de sa ligne de vie, « r » L'étiquette de la ligne de vie ne sera pas soulignée.

- **la section d'interaction entre les objets**

  a. ***Messages***

  `<caller>:<answer>=<callee>[m].<message>`

  `Caller` : L'objet appelant

  `Callee`: L'objet appelé

  `Message` : le message sous forme d'une méthode avec ou sans les paramètres

  `Answer` : la réponse de l'appelé

b. Fragments

```
<[c:<type> <text>]
        foo:bar.message_1
        foo:bar.message_2
        ...
        foo:bar.message_n
[/c]
```

`Type` : Loop(les boucles), Alt (les alternatives), Opt(les options)

`Text` : la condition de ce fragment

c. Commentaires

```
#comments1 Comment2 …
```

Les commentaires sont définie par une seule ligne avec un « # » au début.

Pour plus de détails sur la syntaxe de Sdedit voir : http://sdedit.sourceforge.net/enter_text/

2.1.3 Extension du FeatureIDE pour la composition des diagrammes de séquences

2.1.3.1 Les fonctions à réaliser

Dans cette extension, on va définir toutes les classes et les méthodes qui vont nous aider à :

- Raffiner la composition des features,
- Construire le Modèle comportemental final

- ***Raffinement de la composition :***

Pour réaliser cette fonctionnalité, on a besoin de 3 types de donnée différentes :

- Un fichier « .sdc » initial,
- L'arbre FeatureModel,
- La configuration choisie (elle est représentée sous la forme des features sélectionnés pour etre present dans le nouveau produit)

Phase 1.1 : D'abord, l'idée ici est de créer un autre arbre, qui est une adaptation de l'arbre FeatureModel. **ModelOfFSelected**, le nom de l'arbre, contient les Features sélectionnées dans le fichier de configuration et les Features les plus spécifiques mentionnée dans le fichier de combinaison initial.

Phase 1.2 : Ensuite, la construction du raffinement de la composition n'est donc qu'un parcours de ce nouvel arbre en ajoutant les relations entre les features (seq, alt, opt).

- ***Construire le diagramme de séquence final :***

Pour réaliser cette fonctionnalité, on a besoin de 3 types de données différentes :

- Les fichiers sdedit qui contiennent les diagrammes de séquence pour chaque features. Ce sont des fichiers localisés dans les dossiers de mêmes noms que le nom feature.
- L'arbre **ModelOfFSelected** pour extraire les arborescences entre les features père et fils,
- L'arbre de fichier sdcombin (résultant du parseur de fichier de combinaison raffiné)

Le fichier sdedit se compose de deux parties, une partie pour la définition des objets et une autre partie pour définir les interactions et les événements entre les objets déjà définies.

Phase 2.1 : Création d'une Table de hachage qui contient la liste des fichiers features qui vont interagir dans le diagramme en associant la liste des objets définies de la 1ère partie (déclaration des objets) pour chaque fichier.

Phase 2.2 : L'écriture de la 1ère partie du fichier demande la copie de tous les objets parcourus dans toutes les parties de définitions des objets des fichiers en évitant la redondance. Si on trouve des objets redéfinie plusieurs fois, on ne laisse qu'une seule.

Phase 2.3 : La réalisation de la 2éme partie du fichier du diagramme de séquence Final (la partie interactions entre les objets) se base sur la partie `combination{…}` du fichier de composition. Le squelette des interactions va être la même comme dans le fichier sdcombin en remplaçant les noms des features par leurs 2éme partie de ses fichiers sdedit.

2.1.3.2 Le schéma fonctionnel

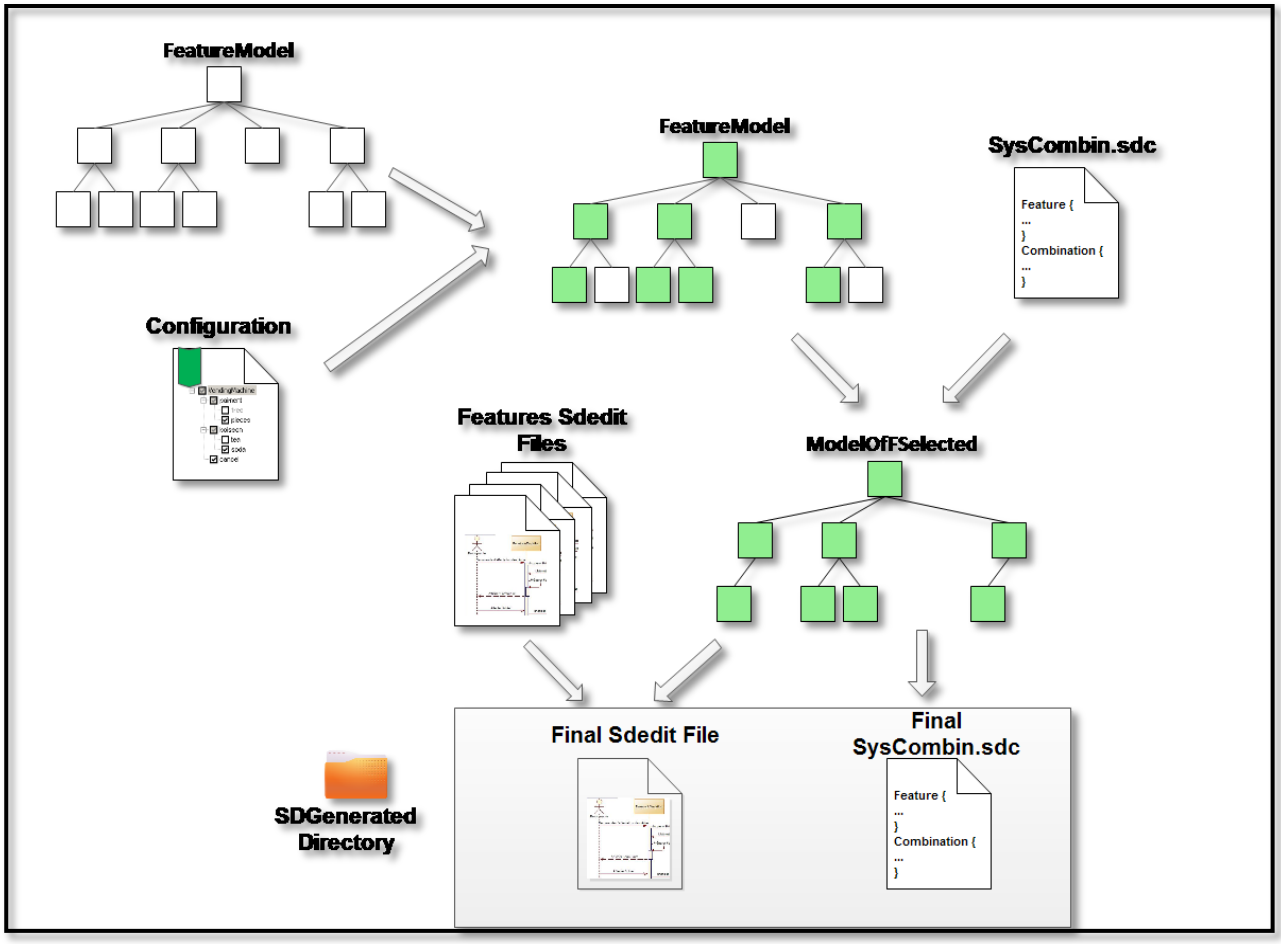


Figure 2:Schéma fonctionnel

Le schéma fonctionnel (figure 2) ci-dessus, représente l'ordre chronologique pour la construction des deux fichiers *Final_Sdedit_File.sd* et *Final_SysCombin.sdc*.

2.2 Diagramme de cas d'utilisation globale

Le schéma de la figure 3, rappelle rapidement les cas d'utilisation du plugin FeatureSeqDiag.

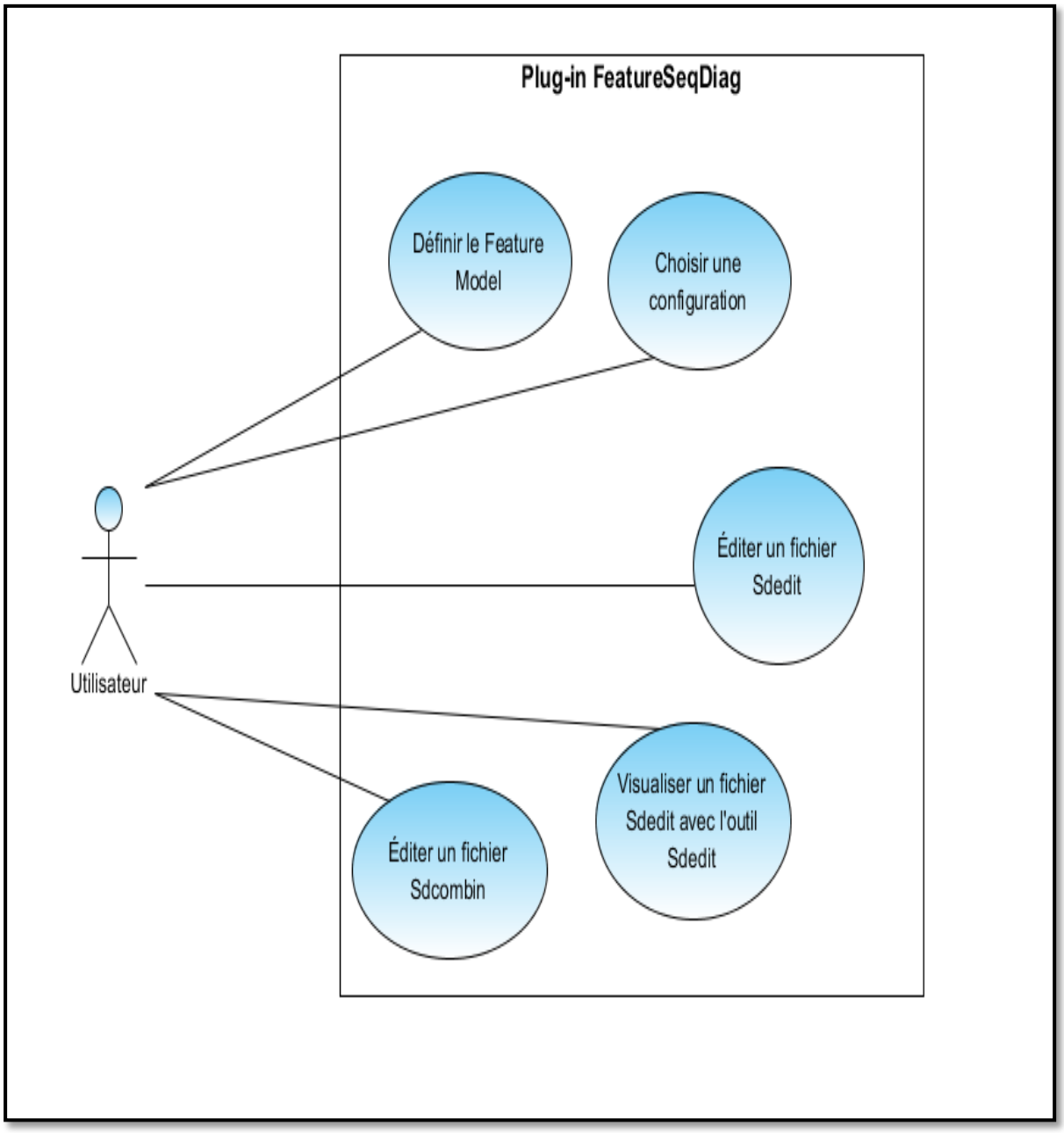


Figure 3: Diagramme de cas d'utilisation globale

2.3 Diagramme de séquence globale

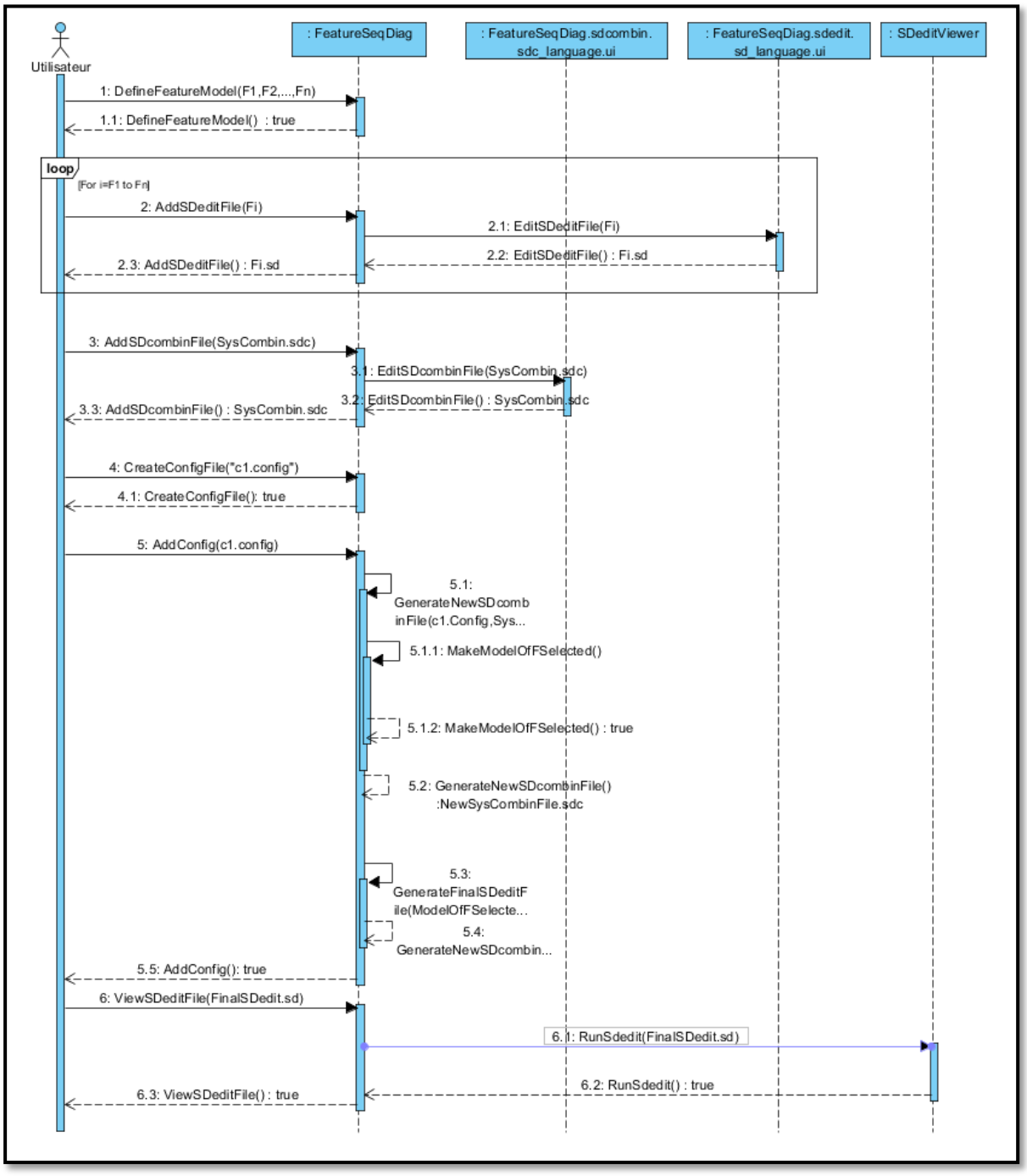


Figure 4: Diagramme de séquence global

Le diagramme de séquence ci-dessus (figure 4), décrit les étapes de déroulement de tous les projets *FeatureSeqDiag* dés la création du *featureModel* jusqu'à la génération de diagramme de séquence Final.

2.4 La structure générale de plugin

La figure 5 présente la structure générale du plugin et ces dépendances.

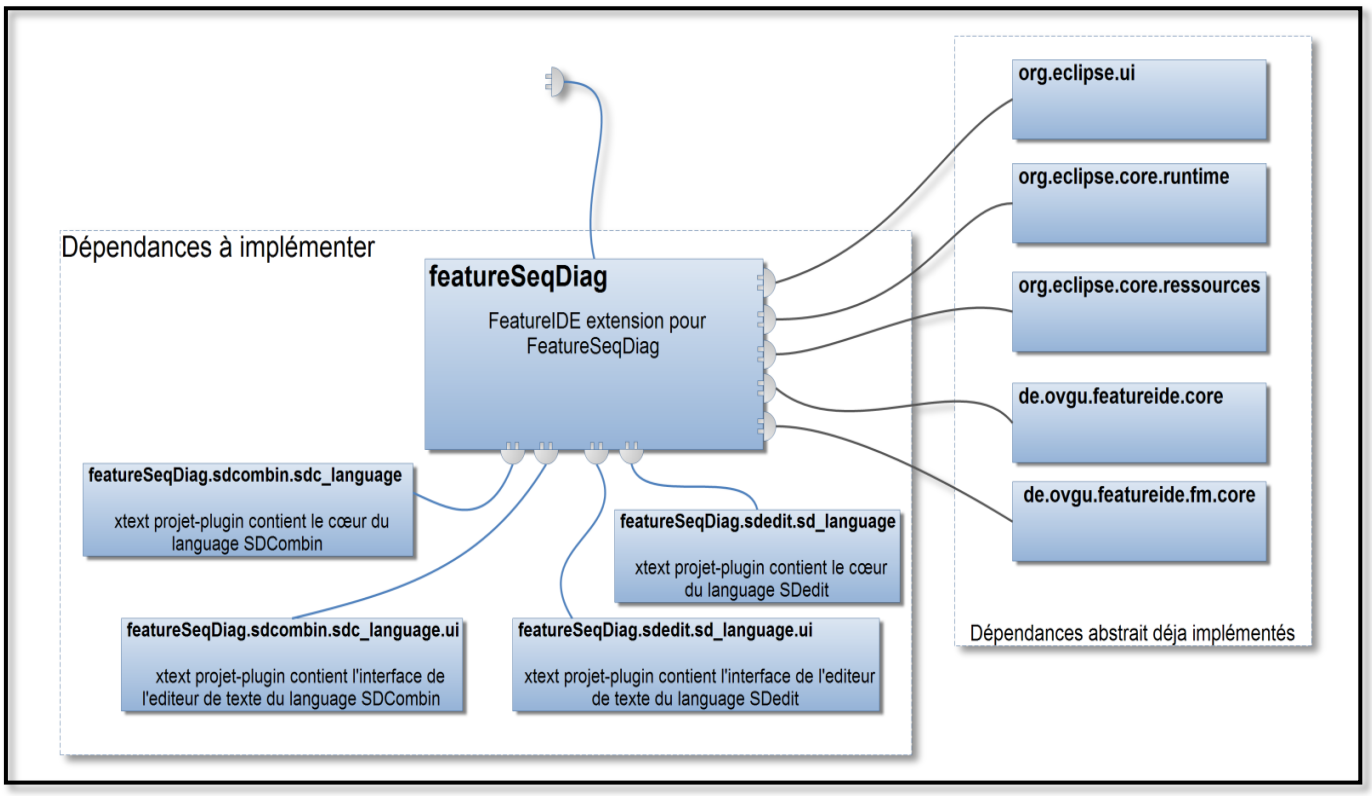


Figure 5 : Schéma de dépendance général du plugin

.

Réalisation

3 Réalisation

3.1 Le projet plug-in sdcombin

Dans cette partie de développement, on intègre l'utilisation de l'outil Xtext, que l'on a défini dans une autre section de ce rapport, et qui va nous permettre de développer l'éditeur du langage Sdcomin.

3.1.1 Structure du plug-in

Le développement du langage va être réalisé en 4 phases :

- ***Phase 1***: Création du projet Xtext Sdcombin et définition de l'extension du langage « .sdc », qui est l'acronime du *« Sequence Diagram Combination »*. L'outil Xtext, va nous créer deux nouveaux project. Chacun de ces projets a un but distinct.

  featureSeqDiag.sdcombin.sdc_language: C'est le cœur du langage sdCombin. Il contient les deux fichiers « ***SDC_language.xtext***» et «***GenerateSDC_language.mwe2***». Ces fichiers vont nous permettre respectivement de décrire la syntaxe du langage et de générer automatiquement toutes les classes de gestion de ce langage et de l'IDE.

  featureSeqDiag.sdcombin.sdc_language.ui: Contient l'interface de l'éditeur du langage et toutes les classes propres à son fonctionnement.

- ***Phase 2:*** Ecriture de la grammaire dans fichier «***SDC_language.xtext***».

- ***Phase 3:*** Génération de l'infrastructure associée à la grammaire en lançant le fichier «**GenerateSDC_language.mwe2**». Ce fichier définit les fragments qui vont être utilisés par Xtext pour la génération de l'éditeur de notre langage. Exemple de ces fragments généré :

Fragment	Éléments générés
EcoreGeneratorFragment	Code EMF pour les modèles générés
ParseTreeConstructorFragment	Sérialisation modèle vers texte
OutlineTreeProviderFragment	Construction de la vue Outline

• ***Phase 4:*** Dans cette phase, on va créer la classe `sdcRunner` qui implémente la méthode statique « `SequenceProg ParseSDCProgram(String uri)` » qui a pour but de parser un fichier on lui passant son chemin absolus comme argument. Il retourne un objet de type `SequenceProg` qui est le fichier sdcombin sous la forme d'une arbre.

3.1.2 La grammaire du langage

```
grammar featureSeqDiag.sdcombin.SDC_language with
org.eclipse.xtext.common.Terminals

generate sDC_language "http://www.sdcombin.featureSeqDiag/SDC_language"

SequenceProg:
	('Features {'
		(features+=Feature)+
	'}')
	('Combination {'
		(contexte+=Contexte)+

	'}');

Feature:
	name=ID
;

Contexte:
	(
		objof=[Feature] (sequence=Sequence)? |
		alternative=Alternative |
		loop=Loop |
		option=Option
	)


;

Condition:
	condition=STRING
;


Loop:
```

```
	'loop [' condition=Condition '] {' (contexte+=Contexte)+ '}'
;

Sequence:
	('seq' objofR+=[Feature])+
;

Alternative:
	'alt [' conditionAlt=Condition '] {' (contexteIF+=Contexte)* 'else'
(contexteELSE+=Contexte)* '}'
;

Option:
	'opt [' conditionOpt=Condition '] {' (contexte+=Contexte)+ '}'
;
```

3.2 Le projet plug-in sdedit

Le projet plug-in sdedit, prend la même démarche que le projet Xtext SdCombin.

3.2.1 Structure du plug-in

De même, le développement de l'éditeur du langage sdedit va être réalisé en 4 phases :

- ***Phase 1***: Création du projet Xtext Sdedit et définition de l'extension du langage « .sd » qui est l'extention des fichiers de l'application Sdedit. L'outil Xtext va créer deux nouveaux projects. Chacun de ces projets a un but distinct.

  featureSeqDiag.sdedit.sd_language: C'est le cœur du langage sdedit. Il contient les deux fichiers « **SD_language.xtext**» et «**GenerateSD_language.mwe2**».

  featureSeqDiag.sdedit.sd_language.ui: Contient l'interface de l'éditeur du langage et toutes les classes propres à son fonctionnement.

- ***Phase 2:*** Ecriture de la grammaire dans le fichier « **SD_language.xtext**».

- ***Phase 3:*** Génération de l'infrastructure associée à la grammaire en lançant le fichier «**GenerateSD_language.mwe2**».

• ***Phase 4:*** Dans cette phase, on va crée la classe `sdRunner` qui est la méthode statique « `SDEditModel ParseSDProgram (String uri)` », et qui a pour but de parser un fichier sdedit, en lui passant son chemin absolus comme argument. Il retourne un objet de type `SDEditModel` qui est le fichier sdedit sous la forme d'un arbre.

3.2.2 La grammaire du langage

```
grammar featureSeqDiag.sdedit.SD_language with
org.eclipse.xtext.common.Terminals

generate sD_language "http://www.sdedit.featureSeqDiag/SD_language"

SDEditModel:
    (elements+=SDEditObj)*
    (elements+=SDEditCombinableDiagElt)*
;

SDEditObj:
    objName=SDEditObjname ':' type=ID ('['('a'|'p'|'r'|'x')']')?
;

SDEditObjname:
      name=ID
;

SDEditCombinableDiagElt:
    SDEditCall|
    SDEditCombinedFragment
;

SDEditCall:
    caller=[SDEditObjname] ':'(returnMessage=ID '=')?
called=[SDEditObjname] '.' name=ID '('
                                          (params=ID
                                    (','paramLists+=ID)*)?
                                    ')'
;


SDEditCombinedFragment:
     SDEditAlt|
     SDEditLoop|
     SDEditOpt
;

SDEditAlt:
    '[c:alt' condAlt=Condition ']'
       conseq+= SDEditCombinableDiagElt+
    ('--[else]'
      (alters+=SDEditCombinableDiagElt)*)+
    '[/c]'
;
```

```
SDEditLoop:
    '[c:loop' condLoop=Condition ']'
       body+= SDEditCombinableDiagElt+
    '[/c]'
;

SDEditOpt:
    '[c:opt' condOpt=Condition ']'
        opts+= SDEditCombinableDiagElt+
    '[/c]'
;
Condition:
      (condLeft+=ConditionElm)+ (conlistRight+=ConditionlistRight)*
;
ConditionlistRight:
      ('IS
NOT'|'IS'|'!='|'=='|'<'|'>'|'<='|'>='|'!'|'||'|'&&'|'and'|'^'|'or')
condRight=ConditionElm
;
ConditionElm:

      operandN=INT| operandStr=ID

;

terminal SL_COMMENT     : '#' !('\n'|'\r')* ('\r'? '\n')?;
```

3.3 Le projet plug-in FeatureSeqDiag

3.3.1 Structure du plug-in

Le plugin se compose de deux grandes packages dépendantes : **featureseqdiag** et **featureseqdiag.services**, et le répertoire **Tools** qui contient l'application « *sdedit-4.01.jar* » (voir la figure 6).

Le package **featureseqdiag** rassemble les classes qui vont réagir dans le cycle de vie du plugin :

- **`FeatureSDCorePlugin.java`**: hérite de la classe abstrait AbstractCorePlugin. Il redéfinit la méthode start qui déclenche l'activation du plugin et la méthode stop effectue le travail inverse.
- **`FeatureSDComposer.java`** : hérite de la classe abstrait `ComposerExtensionClass`. Il redéfinit les méthodes suivantes:

- `performFullBuild (IFile config)` : processeur du plugin qui change la structure du projet FeatureIDE en changeant à chaque fois le fichier de configuration.
- `initialize(IFeatureProject project)` : c'est le « factory » du projet FeatureIDE. C'est dans cette méthode où on peut changer la structure du projet FeatureIDE.

- **`SDeditViewer.java`** : elle met en place la méthode **`run`** où on va implémenter l'action du menu contextuel pour lancer les fichiers sdedit avec l'application « sdedit-4.01.jar ».

Le package **featureseqdiag.services** est le cœur de notre composer. Il rassemble les classes et les méthodes de raffinement qui nous mène à la Diagramme de séquence final.

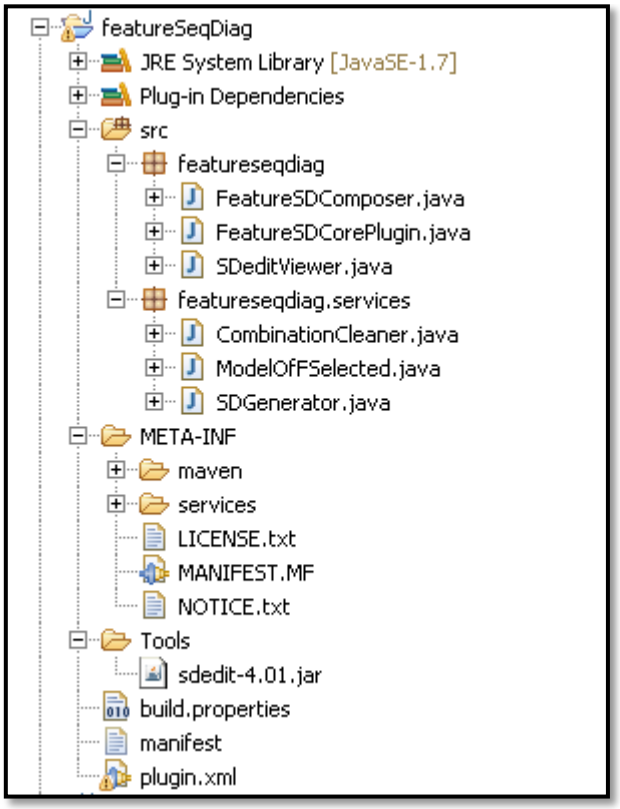


Figure 6 : L'arborescence du plugin FeatureSeqDiag

3.3.2 Les plugins de dépendances

Le plugin **featureseqdiag,** requis plusieurs autres plugins (voir figure 5). Toutes les dépendances sont spécifiées dans le fichier **plugin.xml**. On introduit deux sortes de dépendances :

- Les dépendances standards Java JRE (Java Runtime Environment)
- Les dépendances de Plug-in :
  - les plugins de base: `org.eclipse.core.*`
  - les plugins d'interface graphique: `org.eclipse.ui.*`
  - les points d'extension abstrait de FeatureIDE :
    - `de.ovgu.featureide.core`
    - `de.ovgu.featureide.fm.core`
  - les plugins des éditeurs des deux langages déjà implémenté :
    - `featureSeqDiag.sdcombin.sdc_language`
    - `featureSeqDiag.sdcombin.sdc_language.ui`
    - `featureSeqDiag.sdedit.sd_language`
    - `featureSeqDiag.sdedit.sd_language.ui`

3.3.3 La classe ModelOfFSelected

C'est l'arbre raffiné de notre FeatureModel (arbre de type Feature générer automatiquement par le FeatureIDE) qui contient les Features choisis dans le config.

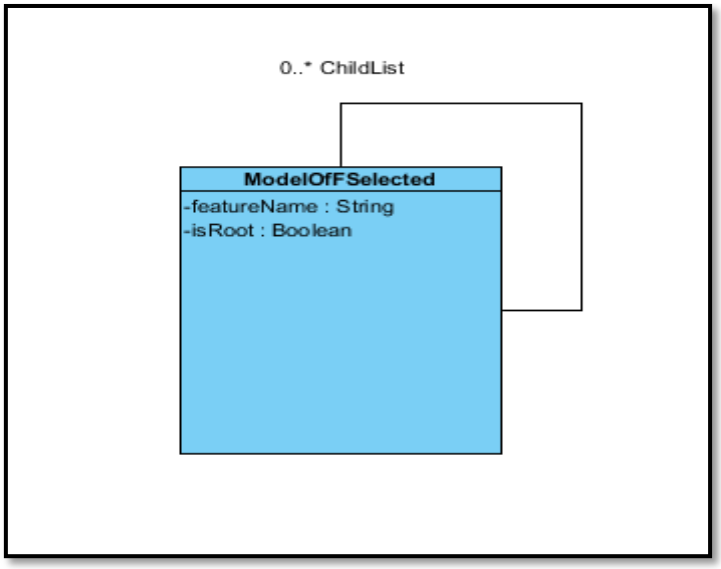


Figure 7 : La classe ModelOfFSelected

Les attributs :

- `String featureName`: L'identifiant unique du feature.
- `Boolean isRoot`: True lorsque le feature s'agit du feature racine.
- `Boolean isAbstract`: True lorsque le feature est abstrait.

Les Méthodes:

- `ModelOfFSelected isModelOfFSelected(String Name)`: elle prend en paramètre le nom de feature et rend un objet de type **`ModelOfFSelected`** s'il existe dans l'arbre. Sinon, elle retourne **null**.

3.3.4 La classe CombinationCleaner

On a rassemblé, dans cette classe, toutes les méthodes qui interagissent dans le raffinement du fichier initial "SysCombin.sdc". Elle implémente les méthodes suivantes:

- ModelOfFSelected tryWithConf (Feature feat, HashMap<String,Feature> listFeatSelcted): Méthode dédié pour la recherche des features sélectionnées et la création du nouvel arbre raffiné «ModelOfFSelected».Elle utilise l'algorithme « Mark and Sweep ».

- listFeatSelcted : liste des features sélectionnées dans le fichier config
- feat : l'arbre du model.

Algorithme :

```
Si Feature est Racine

     M = new ModelOfFSelected() ;

     Si Feature est Abstract alors  M.isabstract = True

     Si Feature a des enfants

           Pour i de 0 à Feature.getChildrenCount()
                 Mi=tryWithConf(feat.getChildren().get(i), listFeatSelcted)

        Si Mi != null alors M.listChild.add(Mi)

 Sinon
           Si  Feature est Hidden retourner null
           Sinon
                 Si Feature est Abstract alors  M.isabstract = True
              Si Feature a des enfants
              Pour i de 0 à Feature.getChildrenCount()
                              Mi   =   tryWithConf(feat.getChildren().get(i),
           listFeatSelcted)

              Si Mi != null alors M.listChild.add(Mi)
```

- **String getlinkedFeatureName(ModelOfFSelected M):** retourne la partie « feature{…} » dans le nouveau fichier New_SysCombin.sdc

```
Features {
            feature1
            feature2
            …
            featureN
}
```

- **String ContexteGenerator(EList<Contexte> seqContext) :** retourne la partie « Combinaison{…} » dans le nouveau fichier New_SysCombin.sdc

```
Combination {
            …
            loop ["condition"] {….}
            …
            seq
            …
            alt ["condition"] {… else…}
            …
            opt ["condition"] {….}
            …
}
```

  - **SeqContext:** c'est la structure contexte que l'on obtient avec le "parseur" de fichier SysCombin.sdc

- **String CleanSysComb(SequenceProg seqProg, HashMap<String, Feature>):** rassemble la génération des 2 parties et définie la mise en forme du nouveau fichier de combinaison New_SysCombin.sdc.
- **static CombinationCleaner.CleanSysComb() :** C'est le responsable de la génération du **New_SysCombin.sdc** dans la classe **FeatureSDComposer** (les lignes de commandes mises en gras):

```
Public void performFullBuild(IFile config) {
      .......
      SequenceProg sysComb =
      SdclRunner.ParseSDCLProgram(fileSysCombin.getAbsolu
      tePath());
      .......
      String sOut =
      CombinationCleaner.CleanSysComb(sysComb,
      selectedFeatures, fRoot);
      .......
  }
```

3.3.5 La classe SDGenerator

On a rassemblé, dans cette classe, toutes les méthodes qui interagissent dans la génération du diagramme de séquence Final. La génération se fait selon cet ordre chronologique :

1. On cherche les fichiers « .sd » des features qui sont sélectionnées et aussi les fichiers « .sd » des features pères non Abstrait et qui a l'un de ces enfants est sélectionné.
2. On rassemble tous ces fichiers dans un **`HashMap<String, String> sdlist`**. Elle contient le nom de feature comme clés et le chemin absolu du fichier `sdedit` comme objet.
3. On génère la partie déclarations des objets `SDEditObj` par :
   a. chercher les objets qui sont déclarés plusieurs fois dans les fichiers.
   b. créations du **`HashMap<String, SDEditObj> objectlist,`** la liste des objets déclarés, en évitant la redondance.
4. On parcourt l'arbre **contexte,** le résultat du parse de la partie combinaison du fichier « SysCombin.sdc », en copiant la partie **`SDEditCombinableDiagElt (`**ou se trouve les interactions entre les objets) pour chaque fichier « .sd».
5. On rassemble toutes ces opérations dans une classe qui s'appelle SDGenerator.

Les Méthodes principales sont :

- `String` **`searchSDFile(String name)`**: cherche le fichier .sd associé au feature de nom « name ». si elle ne le trouve pas, elle retourne une erreur.
- `void setSDLists(ModelOfFSelected M):` C'est une méthode récursive dédiée à remplir la table de hachage sdlist. (méthode récursive)
- `String ObjectInitSD( ):` elle génère la partie de déclaration des objets dans le fichier c1_config_FinalSD.sdel . cette méthode est le responsable de remplir l'objectlist.
- `String combineDiagElement(EList<Contexte>seqContext):` cette méthode crée la partie interactions entre les objet dans notre fichier « FinalSD.sd ».
- `String unifyingSDfile(SequenceProg Sc, String SDF, String sep):` c'est la méthode qui rassemble les deux partie et définie la mise en forme du nouveau fichier sdedit.

  - Sc : c'est la structure générale que l'on obtient qu'après le "parser" de fichier SysCombin.sdcl

- SDF : le nom de fichier c1_config_FinalSD.sdel
- sep : le séparateur dans la structure du chemin absolut. il y a une différence entre les séparateurs de linux et windows.

Résultat

4 Résultat

4.1 Test du langage sdcombin

On créant un ficher d'extension « .sdc» l'éditeur intervient directement pour nous aider à écrire la syntaxe sdcombin. En appuyant sur « ctrl+espace », il nous affiche les choix des mots qu'on peut les choisir.

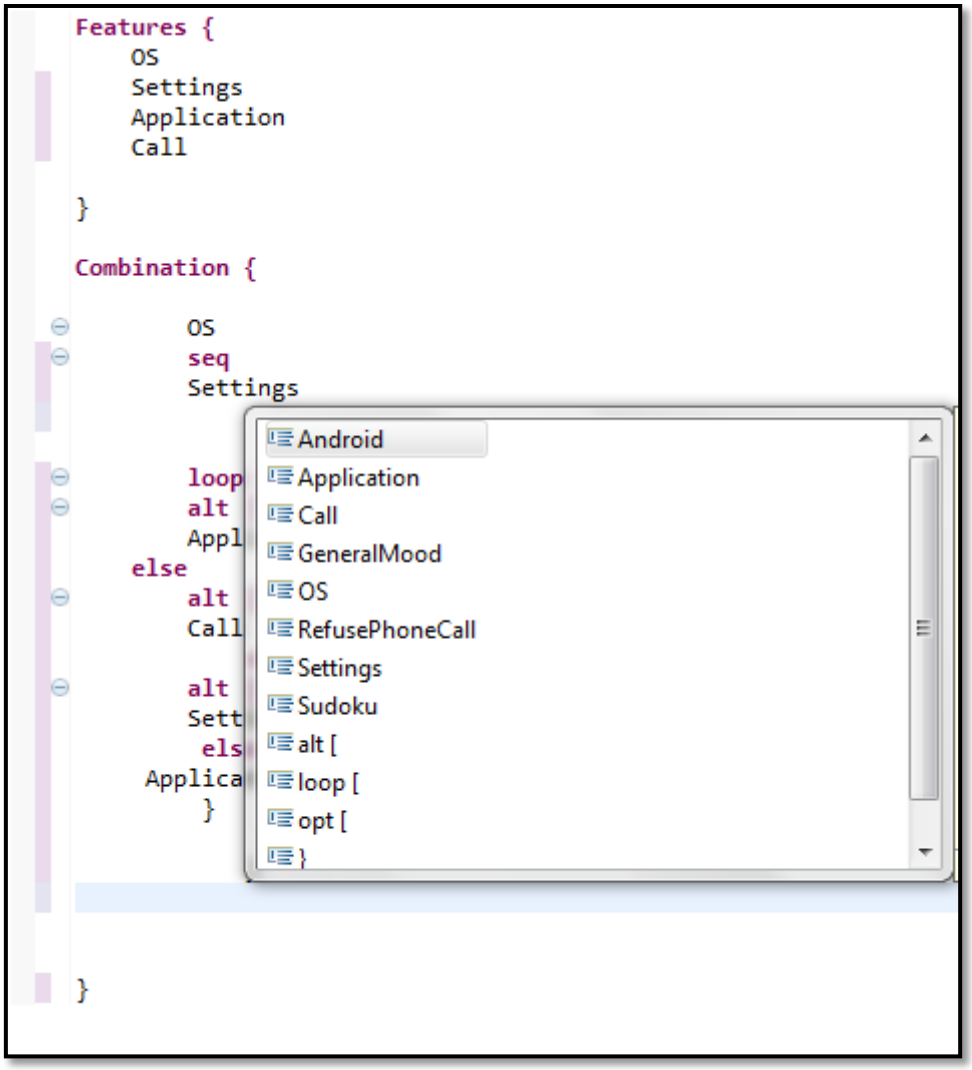


Figure 8 : Menu déroulant qui affiche les choix de la syntaxe

4.2 Test du langage sdedit

On créant un ficher d'extension « .sd» l'éditeur intervient directement pour nous aider à écrire la syntaxe sdedit. En appuyant sur « ctrl+espace », il nous affiche les choix des mots qu'on peut les choisir.

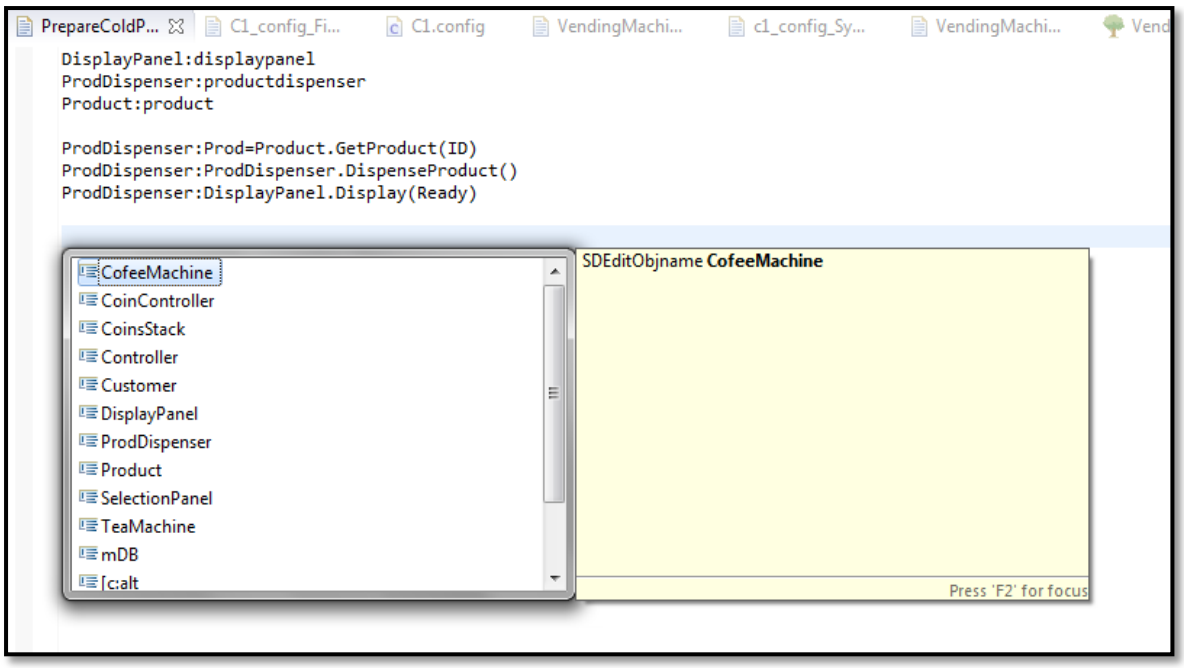


Figure 9 : Menu déroulant qui affiche les choix de la syntaxe

4.3 Exemple du projet featureIDE « VendingMachine »

4.3.1 Les Donnés

4.3.1.1 L'arbre FeatureModel

La figure 10 présente l'arbre FeatureModel de la ligne de produit du projet VendingMachine.

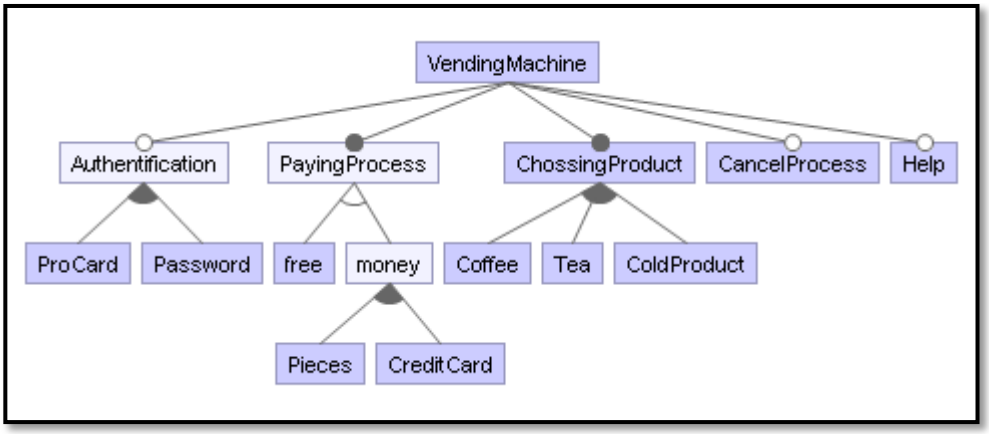


Figure 10 : FeatureModel du VendingMachine

4.3.1.2 Le ficher SysCombin.sdc

On a choisi de définir la combinaison entre les features ascendant du feature racine VendingMachine.

```
Features {
	Authentification
	Help
	PayingProcess
	CancelProcess
	ChoosingProduct
}

Combination {
	Authentification
	loop [" while UserDesire == TRUE "] {
		opt ["Help IS True"] {
			Help
		}
		PayingProcess
		alt ["CancelProcess  == True"] {
			CancelProcess
			else
			ChoosingProduct
		}
	}
}
```

4.3.1.3 Quelque fichiers sdedit des features

Pour chaque feature non abstrait il faut ajouter explicitement un fichier sdedit propre à lui. Chaque fichier est implémenter à l'aide de l'outil éditeur Sdedit en les vérifiant le code dans l'application Sdedit.

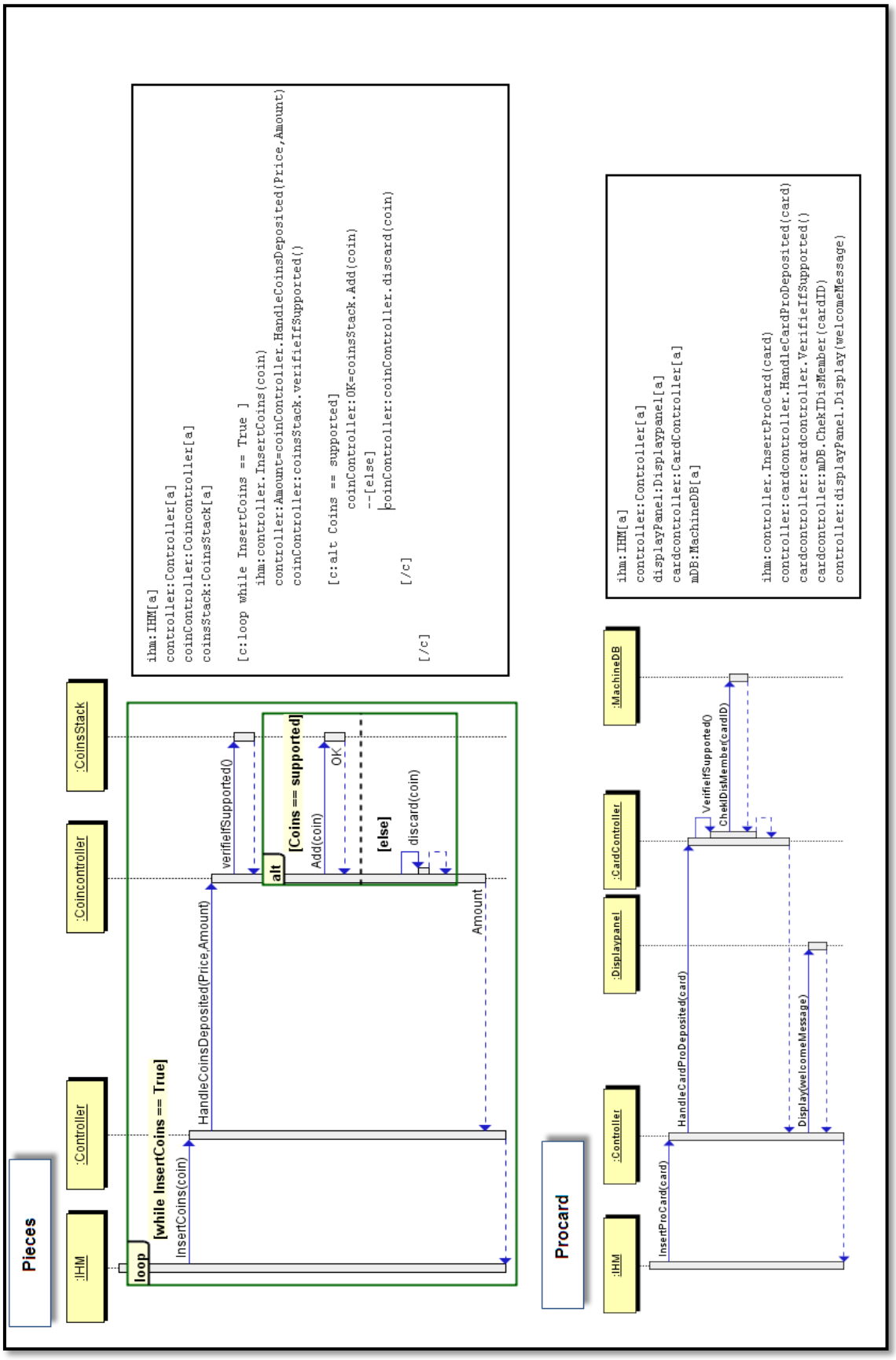


Figure 11 : Fichiers sdedit des features « Pieces » et « ProCard »

4.3.1.4 L'arborescence du projet

La figure 12 représente l'arborescence complète du projet VendingMachine avec tous les éléments nécessaire (les fichiers sdedit, le config). Cette présentation est capturé après la génération de (DS) Final.

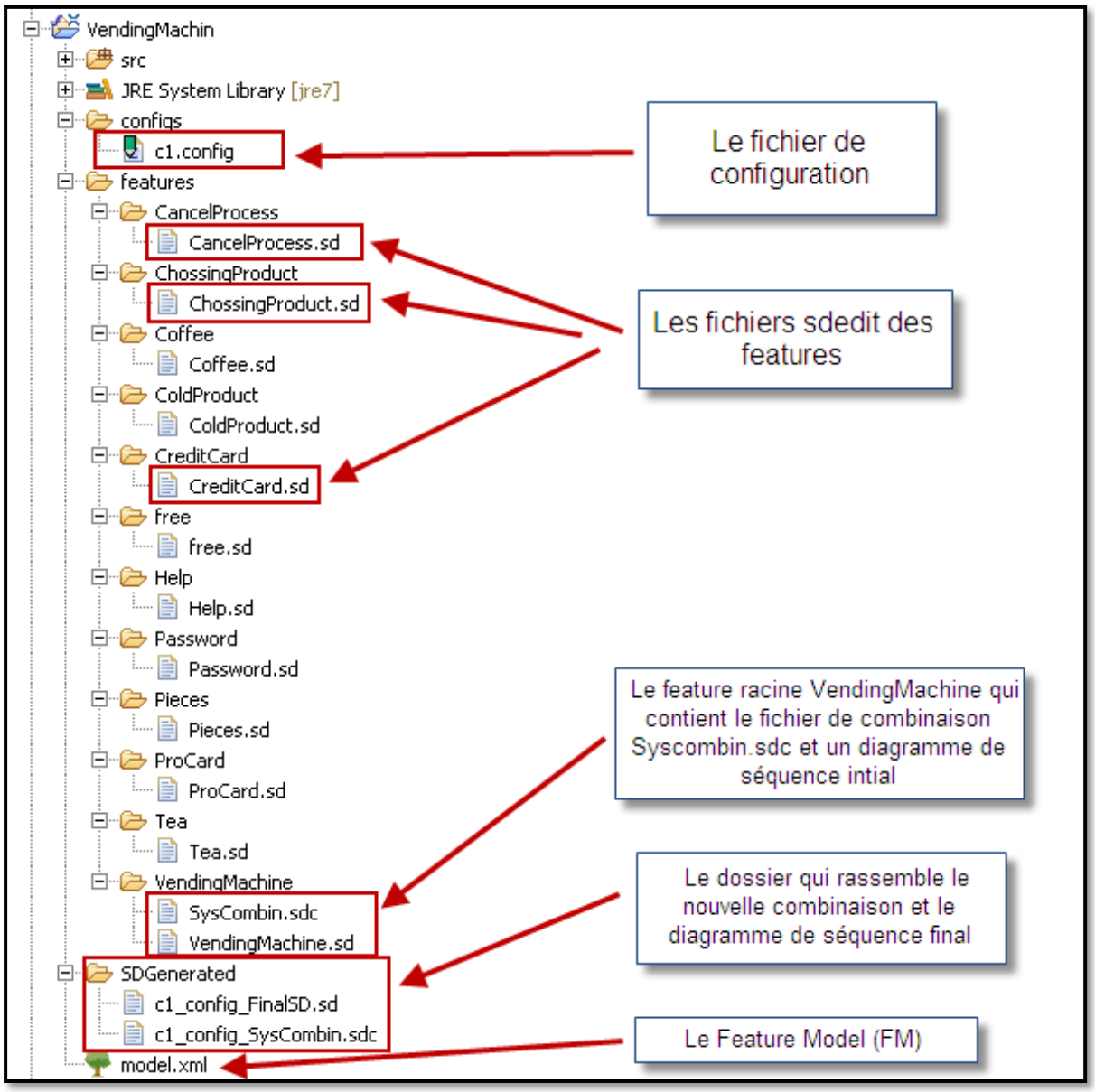


Figure 12 : L'arborescence du projet VendingMachine

4.3.1.5 Le fichier de configuration

La sélection des features était dans le choix suivant spécifié dans la figure 13.

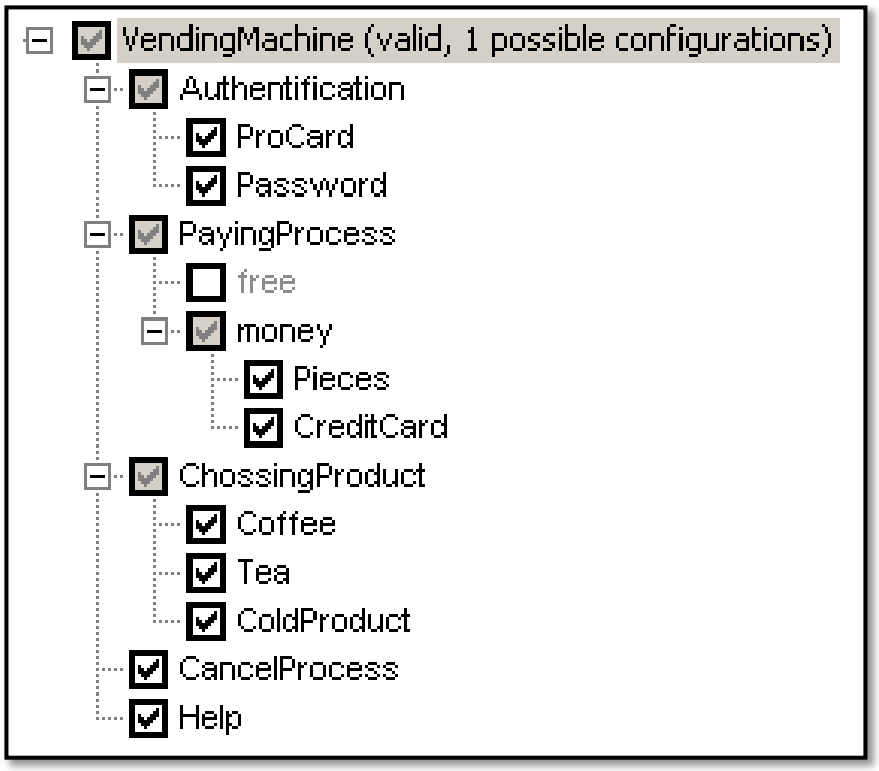


Figure 13 : le fichier c1.config

4.3.2 Les résultats

4.3.2.1 Le fichier « c1_config_SysCombin.sdc »

Le raffinement est sur les composants des arbres ascendentes des pére choisie dans la configuration. Exemple pour le feature « ChoosingProduct » dans la figure

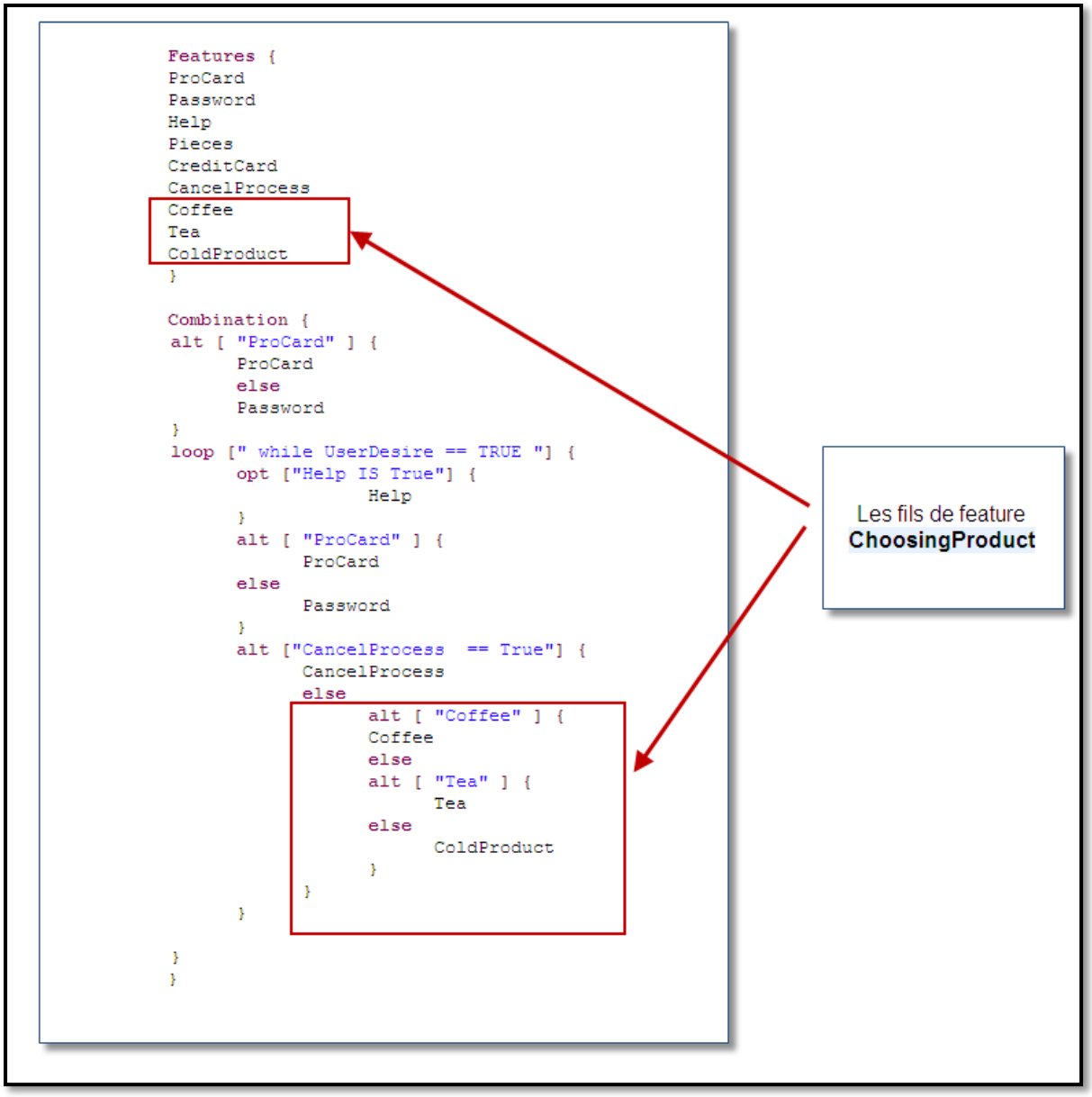


Figure 14 : le fichier « c1_config_SysCombin.sdc »

4.3.2.2 Le fichier « c1_config_FinalSD.sd »

On n'a pas pu représenter le diagramme de séquence dans le rapport. On l'a enregistré dans un fichier **pfd** associé au projet. Dans ce fichier est représenté le diagramme de séquence final.

Conclusion

Dans ce projet, on a traité les modèles de lignes de produits logiciels où la variabilité est spécifiée dans des modèles comportementaux (automates, diagrammes de séquence).

On a implémenté deux mini DSL (Domain Specific Language), en utilisant les technologies autour de Xtext, pour spécifier textuellement les modèles comportementaux (MC). Ensuite, on s'est intéressé à l'implémentation des outils pour « parser » et visualiser les MC. On a proposé un algorithme de dérivation pour les MC qu'on a intégré dans l'environnement FeatureIDE.

Nous avons améliorés nos compétences techniques en java, en implémentation des plugins sous l'environnement Eclipse, en intégrant des algorithmes récursifs dans la catégorie Algorithmes Ramasse-miettes, et en approfondissant nos connaissances dans les notions du monde de produit logiciels.

Bibliographie

Lignes des produits logiciels

- *Les Lignes de Produits Logiciels* par ZIADI TeWfik, UPMC- Master 2

FeatureIDE

- *Development*, Thomas Thuma and Jens Meinicke, January 25, 2013
- *An Extensible Framework for Feature-Oriented Software Development*, Thomas Thuma - Christian Kastnerb - Fabian Benduhna -Jens Meinickea - Gunter Saakea - Thomas Leichc, University of Magdeburg, Germany

Eclipse

- Projet Eclipse :http://www.eclipse.org
- *Tutoriel Plugins Eclipse*, Frédéric Peschanski, 2007-2008

Xtext

- *Introduction à Xtext*, BERNARD Alain, Publié le 28 octobre 2012 http://alain-bernard.developpez.com/tutoriels/eclipse/creation-grammaire-xtext/

Sdedit

- Le site officiel du Sdedit : http://sdedit.sourceforge.net/

Annexe

Annexe

A. Manuel d'utilisation FeatureSeqDiag

Pour pouvoir utiliser notre plugin, il faut suivre les étapes suivantes:

- Sous l'onglet **File** de l'Eclipse cliquez sur **New -> Other**
- Dans le volet Eclipse new sélectionnez **FeatureIDE Project** appuyiez sur **Next**
- Dans le volet FeatureIDE Project selectionnez comme composers : Feature **SeqDiag** appuyiez sur **Next** (voir figure 8).

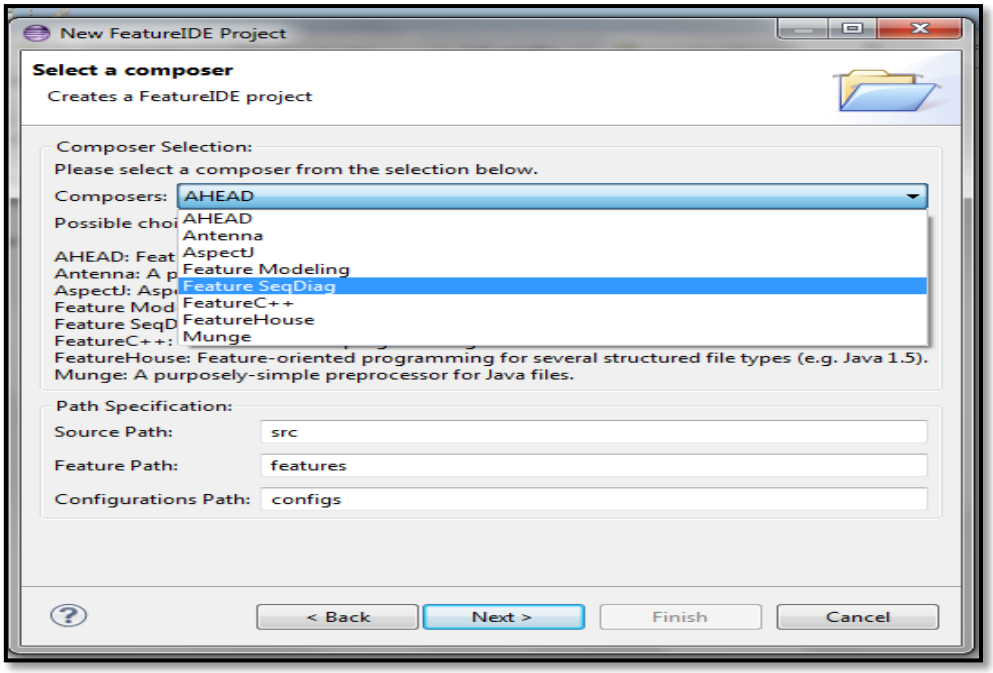


Figure 15: Menu du choix du composer du FeatureIDE

- Dans le volet **New FeatureIDE Project** , tapez le nom souhaité.
- Sélectionner model.xml dans la barre latérale Projet Explorer puis dessinez le modèle d'application souhaité par le client.

- Une fois le modèle est dessinez et sauvegardé, on trouve dans le dossier **features** de la barre latérale **Projet Explorer** qu'un dossier de même nom est créé par item dessiné. Et dans ces dossiers le client va crée un fichier de diagramme de séquence écrit sous une syntaxe de langage SDEdit et dans le dossier de feature de base, un fichier qui décrit l'expression root de l'application écrit sous une syntaxe SdCombin.
- Dans le dossier **config** il faut crée le fichier de configuration
- Ensuite, le client peut choisir la configuration souhaitée et les enregistrer et en fonction de ça un dossier de résultats SDGenerated va apparaitre dans la barre latérale **Projet Explorer** qui contient les résultats de plugin qui sont le diagramme de séquence final et un fichier **SysCombin** qui représente l'expression correspondant a ses configurations.

B. L'outil Xtext

Dans cette section, nous allons présenter l'outil de création de DSL Xtext qui est une composante de TMF (Textual Modeling Framework) intégré dans le framework de modélisation d'Eclipse (Eclipse Modeling Framework : EMF).

a. Vue d'ensemble de l'outil Xtext :

Xtext permet le développement de la grammaire des langages spécifiques aux domaines (DSL : Domain Specific Languages) et d'autres langages textuels en utilisant une grammaire qui ressemble au langage EBNF (Extended Backus-Naur Form). Il est étroitement lié à l'Eclipse Modeling Framework (EMF). Il permet de générer un méta-modèle Ecore, un analyseur syntaxique (parser, en anglais) basé sur le générateur ANTLR ou JavaCC et un éditeur de texte sous la plate-forme Eclipse afin de fournir un environnement de développement intégré IDE spécifique au langage.

La Figure 9 fournit une vue d'ensemble de l'outil Xtext. La forme textuelle du DSL constitue le point de départ. La grammaire du DSL est le point d'entrée de l'outil. Xtext produit, en sortie, un analyseur syntaxique, un éditeur et un méta-modèle pour le DSL.

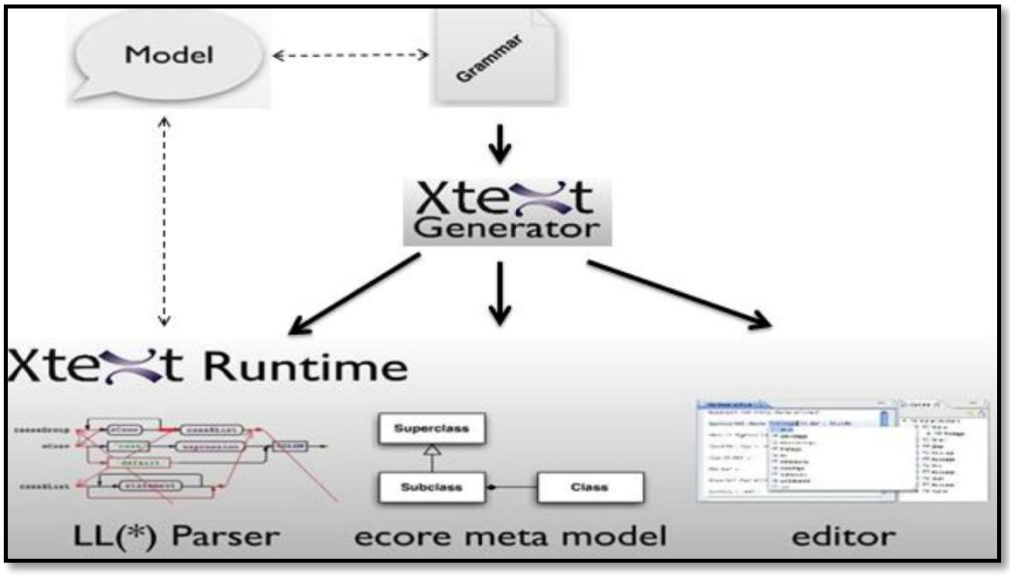


Figure 16: Vue d'ensemble de l'outil Xtext

Xtext permet de considérer la forme textuelle du DSL comme un modèle conforme à la grammaire d'entrée, donc au méta-modèle généré. Présentation de la grammaire de Xtext sur un exemple.

Dans cette partie nous allons donner un petit aperçu sur la grammaire de Xtext en utilisant l'exemple qui se trouve par défaut lors de la création d'un projet Xtext. La capture d'écran de la Figure 10 donne la syntaxe de cet exemple.

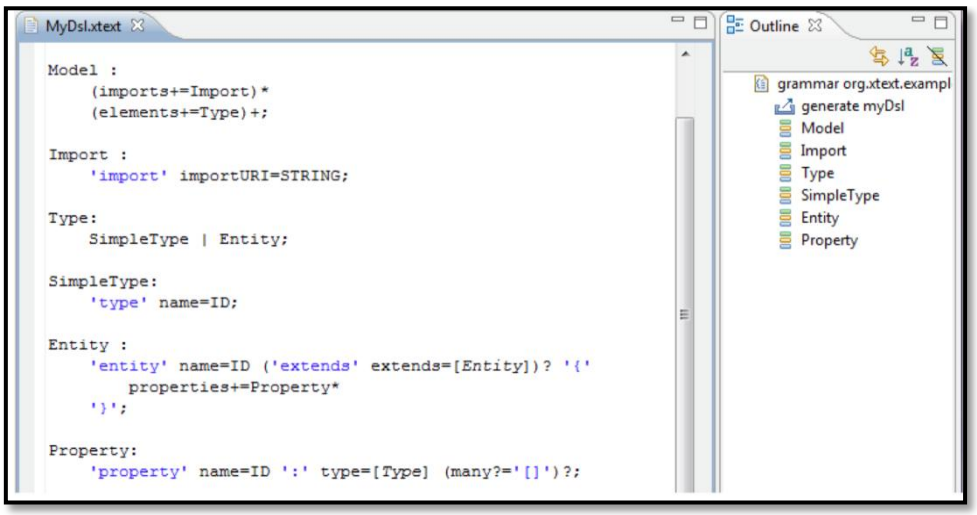


Figure 17: L'éditeur du langage

La grammaire de xtext est composée d'un ensemble de règles (ici Model, Import, Type…). Une règle décrit l'utilisation d'un ensemble de règles terminales et non terminales. Les règles telles qu'ID et STRING sont des règles terminales.

La première règle Model de notre fichier s'appelle la règle d'entrée. Il s'agit d'un conteneur pour tous les Import et Type. Comme nous pouvons avoir zéro ou plusieurs imports, la cardinalité est « * » et un ou plusieurs types, la cardinalité est « + ». L'opérateur d'affectation « += » dénote qu'il s'agit d'un ensemble d'imports et d'un ensemble d'éléments types.

La deuxième règle est la règle Import, cette règle commence obligatoirement par le mot clef „import' suivie d'une chaine de caractères. L'opérateur d'affectation « = » dénote qu'il s'agit d'une règle terminale de type chaine de caractères.

La troisième règle Type indique via l'opérateur d'alternance « | » qu'un type peut être soit un type simple (SimpleType) soit une entité (Entity). Un type simple commence par le mot clef „type' suivi d'un identificateur ; son nom. Une entité commence par le mot clef „entity' suivi d'un identificateur (son nom), d'une accolade ouvrante, un certain nombre de propriétés (zéro ou plusieurs) et se termine par une accolade de fermeture. Une entité peut référencer une autre entité de son super type précédé par le mot clef « extends », Il s'agit d'une référence croisée. Ce cas est optionnel, on le note par la cardinalité « ? ». Pour définir les références croisées il faut utiliser les crochets.

La règle Property commence par le mot clef „property', suivi du nom de la propriété, d'une référence vers son type (précédé par „ :') et d'un suffixe optionnel '[]'. Ce suffixe est optionnel, il faut donc utiliser l'opérateur d'affectation « ?= » et la cardinalité « ? ».

Après avoir exécuté le générateur (Run As $->$ MWE Workflow). Nous pouvons avoir l'éditeur de texte de notre DSL, pour cela il suffit d'exécuter notre projet xtext (clic droit sur le projet $->$ Run As $->$ Eclipse Application). Cet éditeur permet une coloration syntaxique, un mode plan et une validation du code en temps réel. La Figure11, donne une capture d'écran sur un modèle de l'exemple présenté avec l'éditeur de texte de xtext spécifique au DSL crée.

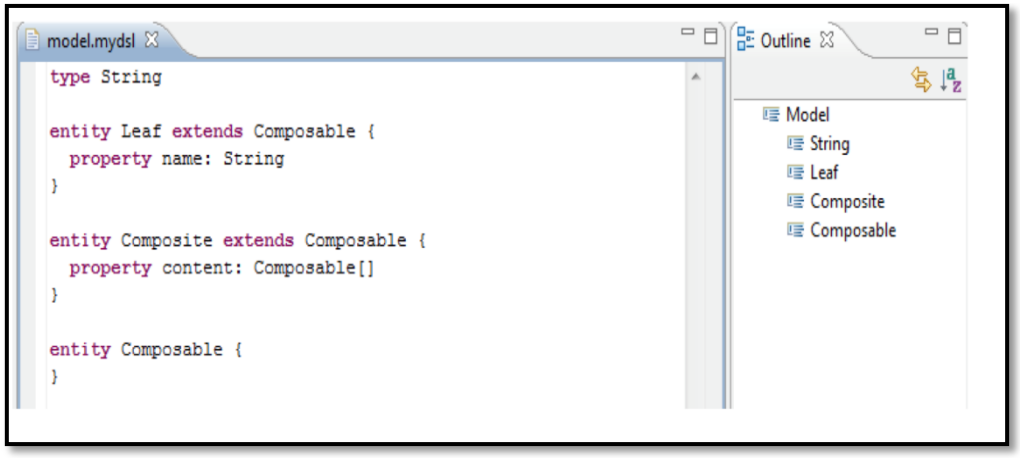


Figure 18: Editer avec le nouveau langage

C. Les plug-ins Eclipse

a. Présentation d'Eclipse :

La plateforme Eclipse est un IDE (integrated development environment) qui peut être utilisé pour créer diverses applications comme des sites web, des programmes Java, des programmes C++. Eclipse est donc un IDE très généraliste.

Eclipse est une plateforme complète qui est en constante évolution grâce à sa modularité et à sa politique d'ouverture sur des produits existants. La plateforme Eclipse a été bâtie autour d'un mécanisme d'intégration et de lancement de modules appelés plugins. Ces derniers vont composer l'ensemble des outils et des fonctions de l'environnement de développement.

Ainsi pour ajouter de nouvelles fonctionnalités à Eclipse comme le support d'un nouveau langage (SdCombin), il suffit de développer un plugin. Eclipse permet de développer des plugins au sein même de son environnement.

b. Présentation technique :

La plateforme Eclipse a été conçue pour satisfaire les points suivants :

- Fournir un environnement pour le développement d'applications.
- Support pour manipuler des contenus très différents (HTML, Java, C, JSP, XML…)
- Faciliter l'intégration d'outils existants et variés.
- Support pour le développement d'applications graphiques ou non.

La plateforme est disponible pour de nombreux systèmes d'exploitation (Windows, Unix, MacOSX).

Le principal rôle de la plateforme Eclipse est de fournir un ensemble d'outils définis par des mécanismes et des règles qui constituent l'environnement de développement. Ces mécanismes sont décrits grâce à des API (Application Programming

Interface), classes et méthodes. La plateforme fournit également un puissant framework qui facilite le développement de nouveaux outils.

La Figure 12, montre les principaux composants et APIs de la plateforme Eclipse.

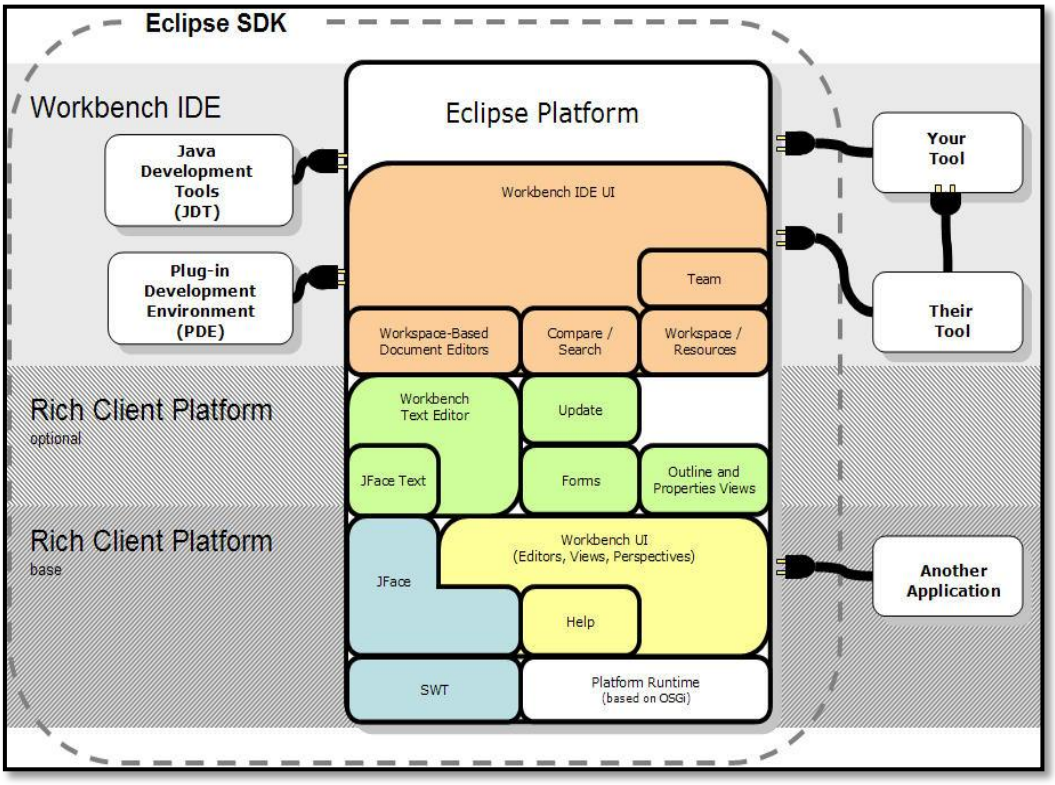


Figure 19:Les principaux composants et APIs de la plateforme Eclipse

(Source: eclipse.org, eclipse-overview)

Un plug-in est le plus petit élément de la plateforme Eclipse qui peut être développé et distribué. Dans la majorité des cas, un nouvel outil est développé sous la forme d'un seul plug-in alors que les fonctionnalités d'un outil complexe seront réparties sur plusieurs plugins.

Toutes les fonctionnalités d'Eclipse sont gérées par des plug-ins sauf pour le noyau de la plateforme appelé plateforme runtime.

Les plug-ins sont entièrement codés en Java. Un plug-in classique est constitué de code Java dans une librairie JAR, de fichiers en lecture seule et d'autres ressources comme des images. Certains plug-ins ne contiennent pas du tout de code. Le plug-in

gérant l'aide en ligne en est un exemple. Il existe aussi un mécanisme qui permet à un plug-in d'être défini par un ensemble de fragments de plug-in. Ceci est souvent utilisé pour le support multi langue des plug-ins.

Chaque plug-in est associé à un fichier décrivant les interconnexions avec les autres plug-ins. Ce fichier est appelé manifest file. Le model d'interconnexions est simple : un plug-in déclare un certain nombre de points d'extensions (extension points), et de liens vers d'autres points d'extensions. Le fichier manifeste est un fichier XML.

Un point d'extension d'un plug-in peut être étendu par d'autres plug-ins. Par exemple, le workbench plug-in déclare un point d'extension pour les préférences utilisateurs. Tout autre plug-in a la possibilité d'avoir ses propres préférences utilisateurs en définissant extension du point d'extension du workbench plug-in.

Un point d'extension peut avoir une API correspondante. D'autres plug-ins peuvent contribuer à l'implémentation de cette interface via des extensions des points d'extensions.

Tout plug-in peut définir de nouveaux points d'extensions et fournir une nouvelle API.

Au lancement, la plateforme Runtime recherche l'ensemble des plug-ins disponibles, lie les fichiers manifestes et construit une structure de dépendance entre les plug-ins, appelée base de registre des plug-ins. La plateforme associe chaque nom des déclarations d'extension avec les points d'extensions correspondants déclarés. Tous les problèmes détectés, comme l'impossibilité d'associer une déclaration à un point d'extension, sont enregistrés dans un fichier log. Les plug-ins ne peuvent pas être ajoutés après le lancement de la plateforme.

Ce système de plug-ins est utilisé pour partitionner la plateforme Eclipse elle-même. Donc on dispose d'un plug-in gérant l'espace de travail (workspace), un autre pour l'environnement graphique (workbench)… Même le noyau d'Eclipse a son propre plug-in. Ainsi pour lancer Eclipse sans interface graphique, il suffit juste de supprimer le plug-in workbench et les autres dépendants de ce dernier.

c. Workspaces :

On appelle workspaces les espaces de travail. Les différents outils introduits dans la plateforme Eclipse interagissant avec les fichiers de l'espace de travail de l'utilisateur. Le workspace regroupe un ensemble de projets qui ont un emplacement propre dans le système de fichiers. Chaque projet regroupe un ensemble de fichiers.

Eclipse introduit un mécanisme de markers qui permet d'associer une valeur à une ressource. Ainsi on peut spécifier la nature d'un projet (projet java, c…). De même on pourra spécifier le compilateur à utiliser pour un type de ressource. Les plug-ins peuvent bien sûr créer de nouveaux markers pour fournir de nouvelles fonctionnalités.

Toutes les opérations sur le workspace sont sauvegardées. Ainsi lors d'une nouvelle session, on retrouvera l'état du workspace comme on l'avait laissé auparavant.

d. Workbench et UI Toolkits :

L'interface utilisateur (UI) de la plateforme Eclipse est gérée par une structure appelée workbench. Cette dernière apporte tous les outils nécessaires à la personnalisation de l'interface utilisateur. L'API du workbench et son implémentation repose sur deux toolkits :

SWT (Standard Widget Toolkit) est une bibliothèque graphique libre pour Java, créée par IBM. Cette bibliothèque se compose d'une bibliothèque de composants graphiques (texte, label, bouton, panel…), de tous les utilitaires nécessaires pour développer une interface graphique en Java, et d'une implémentation native spécifique à chaque système d'exploitation qui sera utilisée à l'exécution du programme. Par contre son API est entièrement indépendante du système d'exploitation.

JFace fournit un ensemble de classes pour gérer facilement de nombreuses tâches sur les interfaces graphiques. L'implémentation et l'API de JFace sont indépendants du système de fenêtres graphiques. De plus JFace a été conçu pour fonctionner avec SWT. C'est cet outil qui va gérer toutes les fenêtres, et vues d'Eclipse.

e. Structure générale :

Dans le schéma illustré dans la Figure 13, on a représenté l'architecture globale de la plateforme eclipse, en ajoutant le framework FeatureIDE et ses plug-in on cite la position du notre plug-in featureSeqDiag.

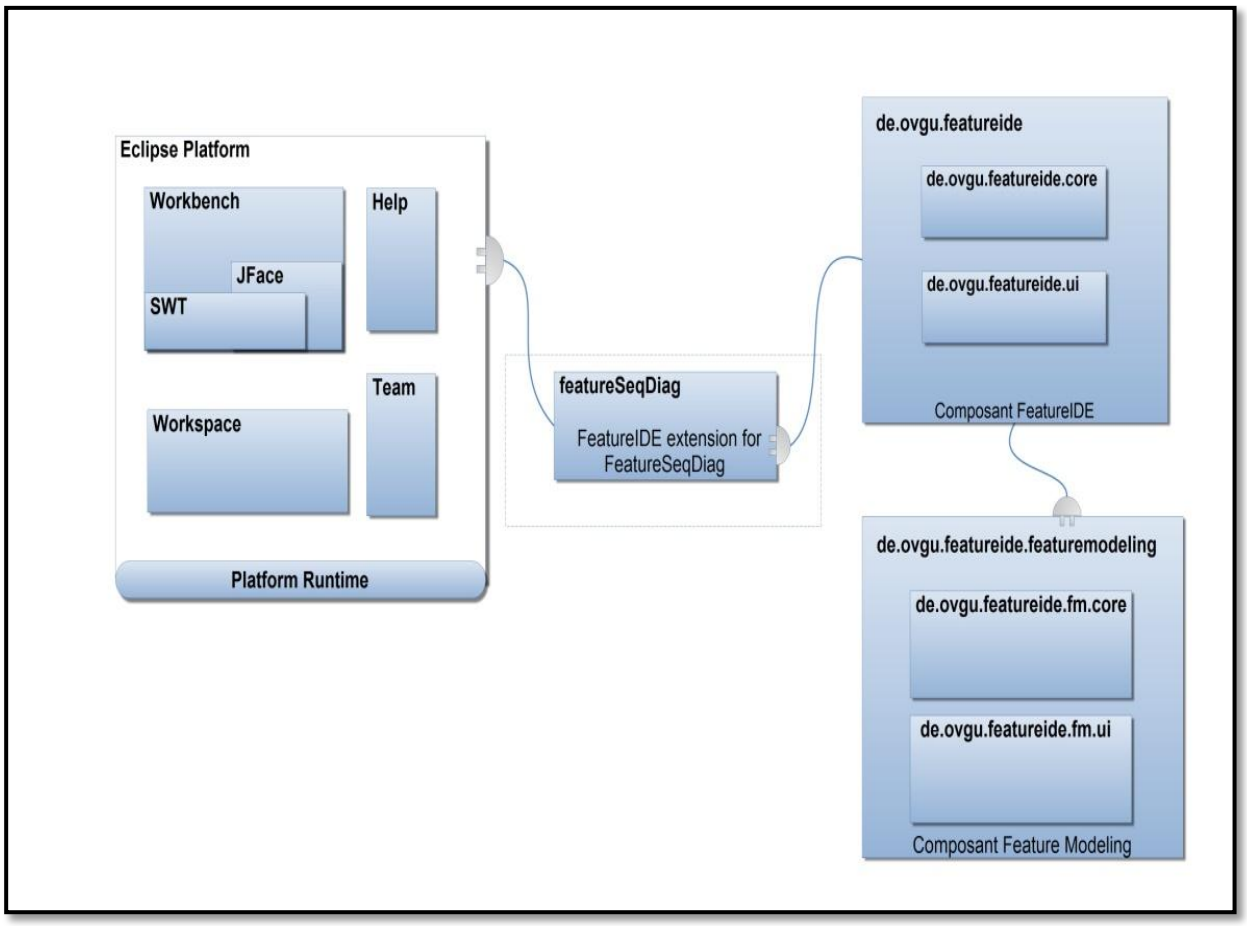
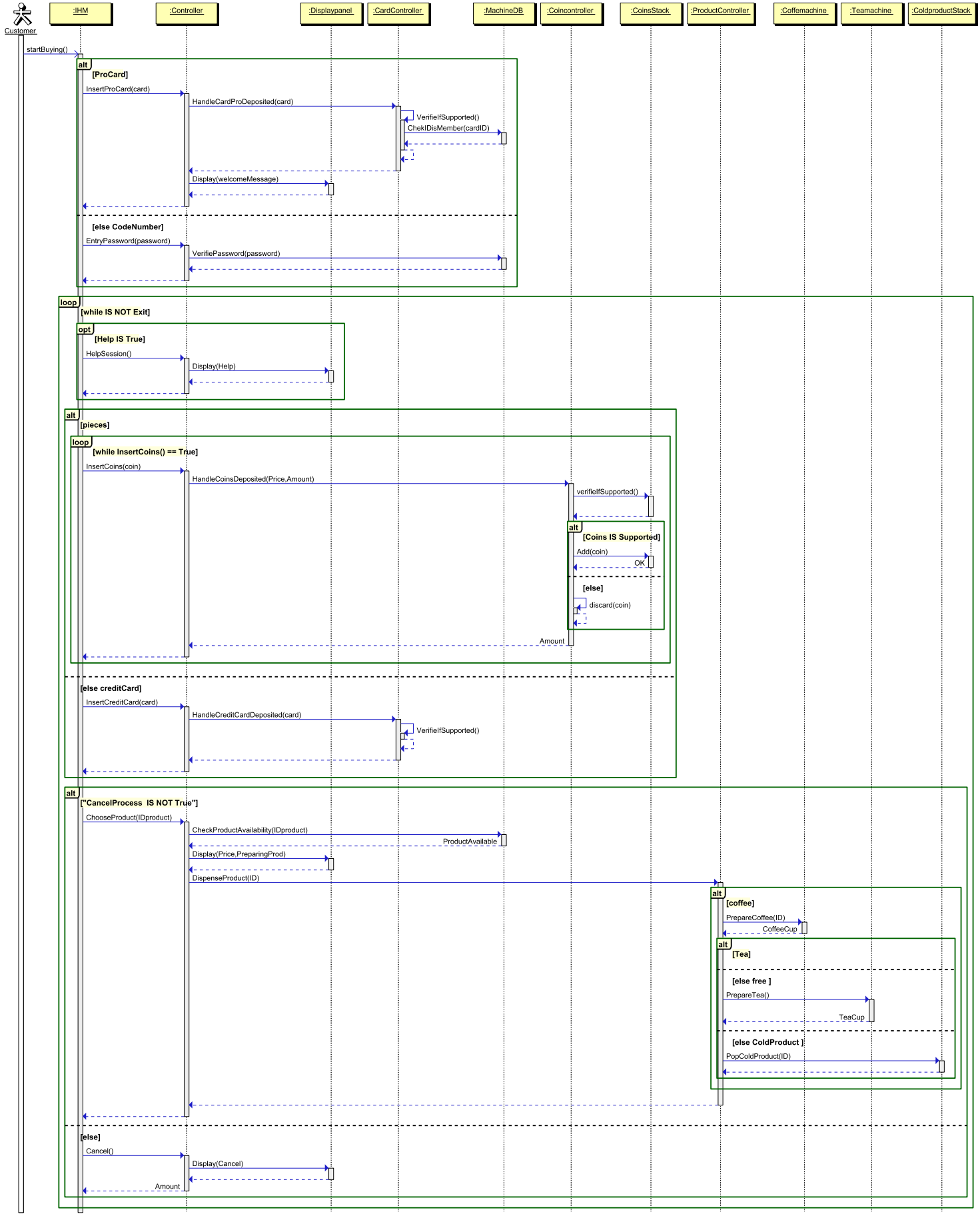


Figure 20:L'architecture globale de la plateforme eclipse


Customer
:IHM
:Controller
:Displaypanel
:CardController
:MachineDB
:Coincontroller
:CoinsStack
:ProductController
:Coffemachine
:Teamachine
:ColdproductStack
startBuying()
alt
[ProCard]
InsertProCard(card)
HandleCardProDeposited(card)
VerifieIfSupported()
ChekIDisMember(cardID)
Display(welcomeMessage)
[else CodeNumber]
EntryPassword(password)
VerifiePassword(password)
loop
[while IS NOT Exit]
opt
[Help IS True]
HelpSession()
Display(Help)
alt
[pieces]
loop
[while InsertCoins() == True]
InsertCoins(coin)
HandleCoinsDeposited(Price,Amount)
verifieIfSupported()
alt
[Coins IS Supported]
Add(coin)
OK
[else]
discard(coin)
Amount
[else creditCard]
InsertCreditCard(card)
HandleCreditCardDeposited(card)
VerifieIfSupported()
alt
["CancelProcess IS NOT True"]
ChooseProduct(IDproduct)
CheckProductAvailability(IDproduct)
ProductAvailable
Display(Price,PreparingProd)
DispenseProduct(ID)
alt
[coffee]
PrepareCoffee(ID)
CoffeeCup
alt
[Tea]
[else free]
PrepareTea()
TeaCup
[else ColdProduct]
PopColdProduct(ID)
[else]
Cancel()
Display(Cancel)
Amount