La programmation transactionnelle au sein d'Eclipse

PHILIPPOT Sébastien

25 Avril 2014

Table des matières

Table des figures

Remerciements

J'adresse mes remerciements aux personnes qui m'ont aidé dans la réalisation de ce mémoire. Particulièrement à Aurélie pour sa patience et sa relecture, Fatima pour sa relecture et ses précieux encouragements.

Je remercie aussi M BRUEL Jean-Michel, Professeur des universités et chercheur à l'IRIT, M LEROUX Renan, ingénieur chez ALTRAN, qui ont su répondre à mes questions.

Enfin, un grand merci à l'équipe pédagogique de l'IPST CNAM Toulouse, pour m'avoir donné les compétences nécessaires pour la réalisation de ce projet.

1 Introduction

Le projet Eclipse

Eclipse a été initialement développé par IBM en 2001, aujourd'hui il est géré par un consortium, en 2003 il est composé de plus de 80 acteurs dont on peut citer IBM, Airbus, BMW, Google, Intel, Oracle, Sony, le CNRS ...

Eclipse est modulaire, tout est plug-ins.

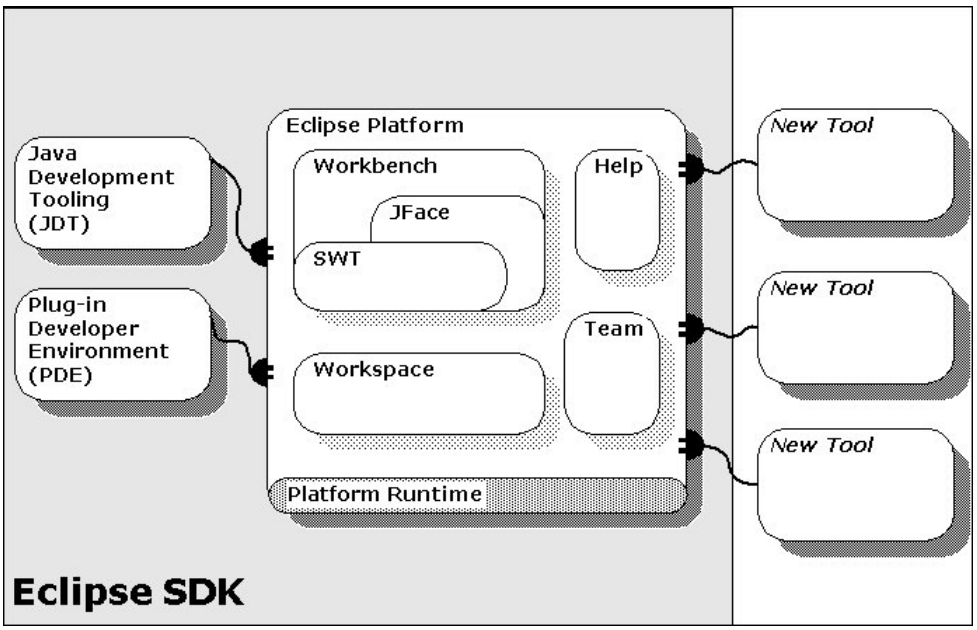


FIG. 1.1 – *Architecture Eclipse*

Il n'est pas question ici de traiter en détail l'architecture d'Éclipse présentée dans la figure 1, toutefois on peut remarquer que la partie centrale *platform runtime* est le coeur d'Éclipse, le *workspace* contient tous les projets réalisés avec Eclipse, JFace est la partie graphique, et surtout on peut voir la possibilité de connecter sur la platform runtime des nouveaux outils.

Il offre une plateforme pour développer des plug-ins. De nombreux logiciels sont donc créés à l'aide du Framework que propose Eclipse, il est évident que de nombreux échanges ont lieu entre ces différents plug-ins, ce dossier portera sur la régulation de ces échanges, afin de garder un ensemble cohérent. La première partie définit les notions de "régulation des échanges" ainsi que

"ensemble cohérent", au travers des transactions. Les accès concurrents seront aussi traités dans cette partie. De nombreux projets sont basés sur EMF, la deuxième partie traitera de l'aspect transactionnel d'EMF.

Il est fréquent d'avoir la nécessité de persister les données d'une application, l'échange a lieu entre l'application et le stockage de l'information. Dans cette seconde partie sera traitée la persistance via le projet EclipseLink.

Première partie

Les transactions

1 Notion de transaction

1.1 Définition

Les transactions appliquées à l'informatique sont le reflet des transactions métier, qui mettent en interaction deux parties prenantes et une tierce partie de confiance, le résultat attendu est garanti. Il s'agit d'une régulation de la fonction d'échange. Dans l'informatique une transaction peut être vue comme une suite d'instruction, c'est-à-dire un ensemble d'éléments finis.

$$T = \{t_1,...,t_n\},\ n < \infty \tag{1.1}$$

Ces instructions peuvent être des opérations de lecture, d'écriture, d'incrémentation ..., dans ce dossier les opérations seront du type lecture (L) et écriture (E), ainsi on a :

$$t_n \in \{L(x); E(x)\} \tag{1.2}$$

x représente une donnée quelconque.

1.2 Des transactions pourquoi faire?

L'informatique est présente dans tout type d'activités, les transactions sont la clé pour rendre toutes ces activités cohérentes [4]. Elles vont mettre en œuvre des règles (propriétés) sur l'échange de données entre deux systèmes. Toutes les instructions d'une transaction doivent être validées, pour que l'échange ait lieu. Ces règles ont été définies par Jim Gray [1] vers la fin des années 1970.

1.3 Propriétés ACID

Quatre propriétés essentielles ont été définies par Jim Gray:

- **Atomicité** : Une transaction est exécutée complètement ou pas du tout. Lorsque la transaction atteint le point de validation *commit*, elle est validée. Si une erreur est découverte avant que la transaction atteigne le point de validation, la transaction s'annule intégralement comme si elle n'avait jamais été exécutée.
- **Cohérence** : Le changement d'état d'un système doit être valide et doit respecter les contraintes liées à cet état, les contraintes d'intégrité, par exemple. En conjonction avec la propriété de l'Atomicité, une transaction doit atteindre le point de validation avec toutes les contraintes satisfaites, si ce n'est pas le cas le système fera un appel explicite à la fonction *RollBack*, et la transaction sera annulée selon le principe de l'atomicité.

1. Jim Gray est un chercheur au laboratoire Microsoft Research, il est reconnu pour ses contributions dans les transactions informatiques

– **Isolation** lorsque plusieurs transactions s'exécutent de façon concurrente, chacune est isolée des autres et ne voient que ses données tant qu'elle n'a pas été validée. Ceci permet que des opérations temporairement incohérentes d'une transaction influent sur une autre transaction.
– **Durabilité** Une fois la transaction validée, les données sont persistés et ne peuvent pas être effacés dans la base de données.

La figure 1.3 montre les changements d'état successifs que peut avoir une transaction.

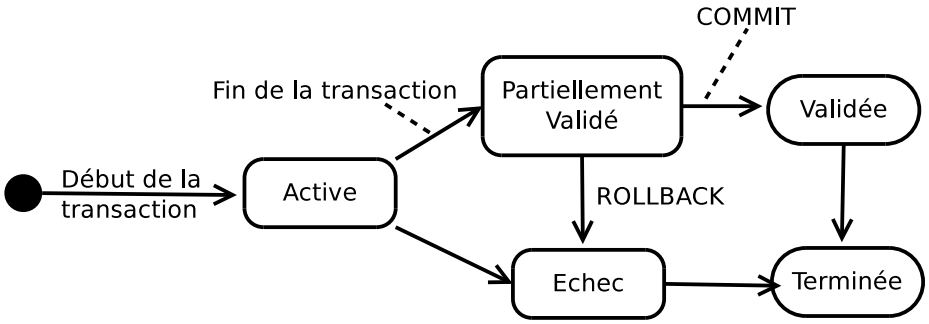


FIG. 1.1 – *diagramme d'état d'une transaction*

2 Concurrence

En français, le mot *concurrence*, signifie selon le dictionnaire Larousse :

Compétition, rivalité d'intérêts entre plusieurs personnes qui poursuivent un même but : Être en concurrence avec quelqu'un pour obtenir un poste.

Cela suggère une compétition entre des personnes, dans le contexte des transactions informatique, cette rivalité a lieu lorsqu'un utilisateur veut lire une donnée alors qu'au même moment un second utilisateur veut écrire sur cette donnée.

Exemple 1. Les deux exemples sont copiés du livre [6].

Une transaction T_1 s'exécute simultanément à une transaction T_2 . T_1 Retire 10 € d'un compte dont le solde est initialement de 100 € tandis que T_2 dépose 100 € sur le même compte. Si ces deux opérations s'exécutent en série l'une après l'autre, sans aucune opération interfoliée, le solde final du compte est de 190 € quelque soit l'ordre d'exécution des deux transactions. Or les transactions T_1 et T_2 démarrent à peu près en même temps et lisent le solde initial de 100 €, ce qui fait 200 € et persiste le résultat. Entre-temps, la transaction T_2 décrémente la copie de Solde de 10 €, ce qui donne 90 € et persiste la valeur finale, écrasant ainsi le solde calculé par T_2. Il s'ensuit une perte de 100 €.

Temps	T_1	T_2	$solde_x$
1		début_transaction	100
2	début_transaction	lire($solde_x$)	100
3	lire($solde_x$)	$solde_x = solde_x + 10$	100
4	$solde_x = solde_x - 10$	écrire($solde_x$)	100
5	écrire($solde_x$)	validation	90
6	validation		90

FIG. 2.1 – *Mise à jour perdue*

Exemple 2. Le problème de la dépendance non validé (ou de la lecture incorrecte, *dirty read* se produit lorsqu'une transaction est autorisé à voir immédiatement les résultats d'une autre transaction, avant que celle-ci n'ait été validée. La figure 2 montre un exemple de dépendance non validée qui provoque une erreur, sur base de la même valeur initiale du $solde_x$ que dans l'exemple précédent. Ici la transaction T_4 modifie $solde_x$ à 200 €, mais la transaction est avortée, de sorte que le $solde_x$ est restauré à sa valeur initiale de 100 €.Or la transaction T_3 a lu la nouvelle valeur de $solde_x$ et l'utilise pour calculer une réduction de 10 €, donnant ainsi une valeur incorrecte de 190 € au lieu de 90 €. La valeur du $solde_x$, lue par T_3 est dites *«donnée sale» [dirty data]* et met en exergue le problème de la lecture incorrecte.

Temps	T_3	T_4	$solde_x$
1		début_transaction	100
2		lire[$solde_x$]	100
3		$solde_x = solde_x + 100$	100
4	début_transaction	écrire[$solde_x$]	200
5	lire[$solde_x$]	...	200
6	$solde_x = solde_x - 100$	annulation	100
7	écrire[$solde_x$]		190
8	validation		190

FIG. 2.2 – *Lecture sale*

2.1 Méthode de gestion de la concurrence

Il existe deux méthodes principales pour gérer la concurrence, la première dite *pessimiste* dans laquelle on va supposer qu'un conflit peut arriver à tout moment, la seconde est appelée méthode *optimiste*, on suppose dans celle-ci qu'un conflit est rare.

2.1.1 Méthode pessimiste

Dans cette approche lorsqu'un utilisateur veut un accès en lecture ou en modification sur une donnée, les autres utilisateurs ne peuvent plus faire d'actions qui peuvent créer des conflits. Pour ce faire on peut exécuter les transactions en série c'est-à-dire que tant qu'une transaction n'a pas été validée, aucune autre transaction ne pourra commencer. Cette sérialisation garantit qu'aucun conflit ne peut avoir lieu.

Exemple:

Temps	T_1	T_2
1	$L(x)$	
2	$R(x)$	
3	validation	
4		$L(x)$
5		$R(x)$
6		validation

FIG. 2.3 – *Planification sérielle*

Cet ordonnancement des opérations s'appelle planification [Schedule] sérielle. Toutes les opérations ne sont pas concurrentes entre elles, deux lectures par exemple n'amènent pas à un conflit.

Temps	T_1	T_2
1	$L(x)$	
2		$L(x)$
3	$R(x)$	
4	validation	
5		$R(x)$
6		validation

FIG. 2.4 – *Planification non sérielle*

Dans cet exemple les données du système restent cohérentes et l'ordonnancement donne le même résultat que la planification sérielle 2.3, on dit qu'elle est *sérialisable*.

Il existe deux techniques de contrôle de concurrence, les méthodes de verrouillage et la méthode d'estampillage.

Méthode de verrouillage Dans ce cas une transaction avant de s'exécuter va demander un verrou sur la donnée, dont l'action porte, elle ne sera ainsi utilisable par aucune autre transaction, si l'action est une opération de lecture alors le verrou est dit *partagé*, s'il s'agit d'une opération d'écriture le verrou est dit *exclusif*, il est possible que plusieurs transactions détiennent un verrou partagé, puisque les opérations de lecture ne génèrent aucun conflit. Si une transaction T_1 détient le verrouillage exclusif sur une donnée a, alors aucune transaction ne peut lire ni modifier cette donnée, tant que T_1 ne libère pas a. Pour chaque objet O, une table d'allocation des verrous est maintenue.

- $O.ecrivain$ = ensemble des écrivains pour l'objet O
- $O.lecteurs$ = ensemble des lecteurs pour l'objet O
- $O.attente$ = ensemble des transactions en attente pour l'objet O

Avec la technique du verrouillage pour maintenir la sériabilité on fait appel au protocole de verrouillage en deux phases (V2P ou 2PL)

Verrouillage en deux phases Le principe de ce protocole est de diviser chaque transaction en deux phases, une d'acquisition des verrous et une seconde de libération dans laquelle la transaction libère les verrous. Une fois que la transaction libère un verrou, elle ne peut plus en demander d'autres.

Inter-blocage Si une transaction T_1 attend qu'une transaction T_2 libère une ressource alors que T_2 attend que T_1 libère cette même ressource, alors il y a un interblocage, aussi appelé *DeadLock*.

Temps	T_1	T_2
1	debut_transaction	
2	verrou_ecriture[$solede_x$]	$L(x)$
3	lire[$solede_x$]	
4	$solde_x = solde_x - 10$	
5	écrire($solde_x$	$R(x)$
6	verrou_écriture($solde_y$)	validation
7	ATTENTE	validation
8	ATTENTE	ATTENTE
9	ATTENTE	ATTENTE
10	|	|

FIG. 2.5 – *Inter-blocage entre deux transactions*

Cet exemple est extrait de [6]. Pour éviter ce type de conflit le système à deux solutions

Prévention des verrous indéfinis

- **Approche Wait and Die :** Si une transaction T_1 veut accéder à une donnée déjà verrouillée par une transaction T_2, T_1 peut attendre T_2 que si elle est plus ancienne sinon elle "meurt".
- **Approche Wound and Die :** Si une transaction T_1 veut accéder à une donnée déjà verrouillée par une transaction T_2, T_1 peut attendre T_2 que si elle est plus récente sinon elle est annulée.

Détection d'un interblocage Si une transaction T_2 verrouille un objet o et qu'une transaction T_1 souhaite accéder à cet objet, on dit que T_1 dépend de T_2, on peut représenter cette situation dans un graphe appelé *graphes des attentes*, dans lequel une transaction serait représentée par un noeud N, et la "relation T_1 dépend de T_2" serait représentée par une flèche de T_1 vers T_2, si dans ce graphe on détecte un circuit alors il y a un interblocage.

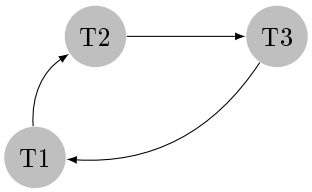


FIG. 2.6 – *Graphes d'attente avec un cycle*

Estampillage Une autre technique se rajoute aux protocoles de verrouillage, celui des estampilles, un identifiant est donné aux transactions de façon chronologique, on peut considérer une estampille comme étant la date de début d'une transaction, ainsi elle est repérée de façon unique, le système peut trier les estampilles. Chaque donnée contient deux estampilles, une de lecture et une d'écriture, qui correspondent aux anciennes transactions.

Exemple . La transaction T_1 à une estampille notée E_T1. x représente une donnée avec deux estampilles eL_x pour l'estampille de lecture et eE_x pour l'estampille d'écri-

ture. Si T_1 souhaite écrire sur x, le système va comparer eE_x avec E_T1, si $E_T1 < eE_x$ alors T_1 sera annulée, redémarrée avec une autre estampille. En effet $E_T1 < eE_x$ montre qu'une transaction plus récente à mis à jour x.

2.1.2 Méthode optimiste

Dans cette approche les conflits sont rares, on va laisser les opérations s'exécuter telles quelles sans intervention. Au moment de la validation du commit on vérifie s'il y a un conflit, dans ce cas la transaction est annulée et redémarrée. Généralement les méthodes optimistes suivent trois phases :

- **Phase de lecture** : Durant cette phase les transactions lisent les données dont elles ont besoin et les stockent dans des variables locales.
- **Phase de validation** : des contrôles y sont menés pour vérifier que la sérialisation n'est pas transgressée [6] s'il n'y a aucun conflit la transaction est validée, sinon elle est annulée et redémarrée.
- **Phase d'écriture** : les modifications apportées à la copie locale sont appliquées à la base de données [6].

Seconde partie

Les transactions dans Eclipse

1 Eclipse Modeling Framework

1.1 Le projet EMF

Le projet EMF est un framework de modélisation maintenu par la fondation Eclipse, il a été initié par IBM, ce projet est sous licence EPL[1].

Il n'existe pas vraiment de différence entre la modélisation et la programmation avec EMF, il réunit ces deux parties de façon bien intégrées. On peut voir EMF comme un compromis entre ces deux parties [5].

1.1.1 EMF pourquoi faire?

Un domaine métier peut-être représenté de différentes manières, au travers d'un modèle UML, avec des classes (interfaces) JAVA, ou bien avec des fichiers de type XML par exemple. Ces représentations définissent strictement la même chose, d'où l'idée de générer à partir d'une représentation les deux autres, et c'est ce que propose EMF, qui se veut unificateur de JAVA, XML, UML [5].

Ce framework offre des outils pour manipuler des modèles et générer du code, les modèles peuvent être sauvegardés dans différents standards (XML, XMI, JAVA avec annotation ...)[5], une chose intéressante est la capacité à accepter plusieurs formats en entré (Java, XML ..) Les modèles EMF sont décrits par des modèles, un modèle qui décrit un modèle s'appelle un métamodèle.

1.2 Notion de métamodélisation

La métamodélisation fait partie de l'ingénierie dirigée par les modèles (IDM). L'idée est de placer les modèles au centre du cycle de vie d'un logiciel, et de leur donner un sens productif, un interêt de cette méthode est de pouvoir adapter les applications aux nouvelles technologies, sans avoir à réécrire le code de cette application. L'OMG a proposé MDA (Model Driven Architecture), qui sépare l'aspect métier de l'aspect technique. MDA est architecturé en quatre niveaux.

1. Eclipse Public Licence se veut plus souple que la licence Gnu GPL, dans le sens où elle n'oblige pas de contribuer au projet

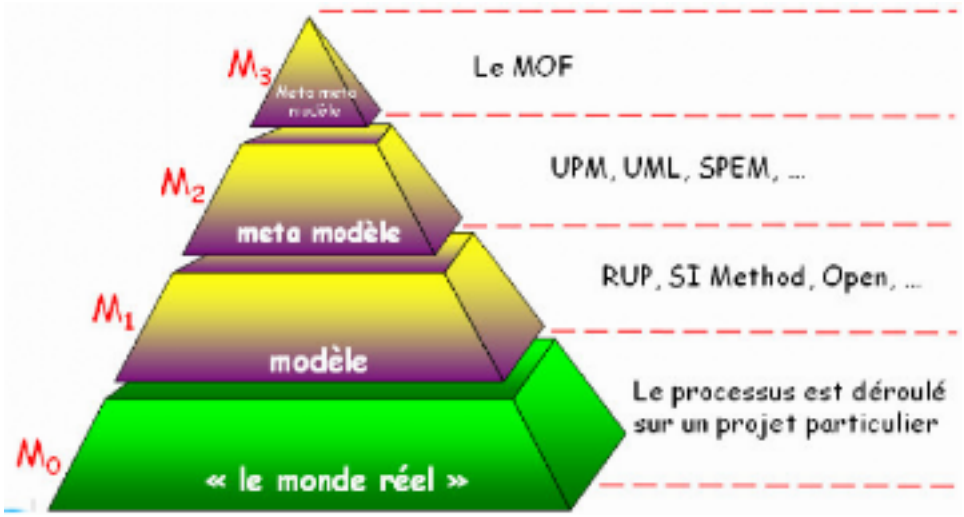


FIG. 1.1 – *Représentation des niveaux définis par le MOF*

Dans la figure 1.1, on a une pyramide représentant les quatre niveaux d'abstraction. Le sommet représente le niveau le plus abstrait et la base le niveau le moins abstrait, ainsi plus on monte dans cette pyramide plus on rend notre système abstrait. Au sommet on trouve la méta-métamodélisation, c'est-à-dire que les métamodèles sont définies dans ce niveau, un méta-métamodèle est réflexif dans le sens qu'il se définit lui même, par exemple le MOF (Model Object Facilities), qui est un langage de définition des métamodèles de niveau M2.

Le niveau juste en dessous est le niveau des métamodèles, ce sont des instances des méta-métamodèles du niveau M3. Dans le niveau M1 on a les modèles, ce sont des instances des métamodèles de niveau M2.

L'OMG a défini trois modèles principaux pour MDA :

- **CIM** *Computational Independant Model :* Ce modèle décrit les exigences fonctionnelles du système.
- **PIM** *Plateform Independant Model :* Ce modèle représente la logique métier.
- **PSM** *Plateform Specific Model :* il dépend de la plateforme utilisé, il est très proche du code, par exemple on peut avoir PSM-EJB.

1.2.1 La transformation de modèle

Le code source de l'application est généré, par transformation de modèle. Le CIM va être transformé vers le PIM, et le PIM vers le PSM. L'inverse est vrai aussi.

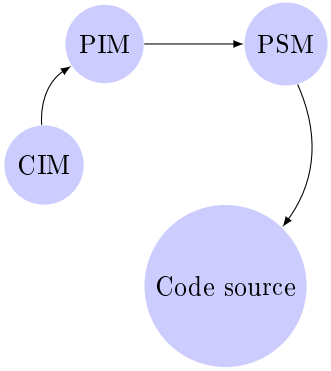


FIG. 1.2 – *Transformation de modèle*

Puisque le code est généré, il est le reflet des modèles.

1.3 EMF et les métamodèles

Le métamodèle Ecore décrit les modèles EMF, il est lui même un modèle EMF, et dans ce sens c'est un méta-métamodèle [5].

Un modèle Ecore est composé de :

- **Eclass** correspond à un concept de base.
- **EAttribute** correspond aux propriétés du concept de base.
- **EReference** correspond aux relation entre les concept de base.
- **EOperation** correspond aux opérations des concepts de base.

Ce modèle peut-être sérialisé dans un fichier XMI, il peut être crée à partir de trois formats d'entrée (JAVA, XML, UML), le code JAVA est généré à partir du modèle Ecore. Pour chaque Eclass, EMF crée une interface et une classe qui l'implémente c'est un choix d'EMF [5], chaque interface hérite de EObject, cette dernière hérite d'une autre interface Notifier, ainsi un objet EMF peut être *écouté* ou *observé*, pour mettre à jour les vues, par exemple. Les écouteurs sont appelés Adapter.

Remarque . EMF n'utilise pas le Design Pattern Observer/Observable car JAVA n'autorise pas l'héritage multiple, et donc si on utilise Observer, on va devoir hérité de la classe Observer et donc perdre tout autre héritage, en utilisant Adapter, on résout ce problème.

1.4 EMF et les transactions

Le contenu de cette section est directement inspiré de l'aide officielle fournit par Eclipse, les exemples et le code JAVA sont copiés depuis ce site internet [2].

EMF permet à plusieurs Threads de manipuler un domaine, pour éviter la concurrence de ces Threads, ce framework fournit une API pour la gestion des transactions. Ainsi on peut contrôler l'intégrité d'un domaine, et de bénéficier du contrôle de concurrence des transactions comme on a vu au chapitre 2.

Un identifiant est donné à un domaine d'édition transactionnel. Chaque instance d'un modèle est placée dans un conteneur nommé Resource, l'interface ResourceSet est utilisée comme conteneur de Resource. Pour créer un domaine d'édition transactionnel, on peut utiliser deux solutions : le pattern Factory ou l'interface Registry.

- **Factory**

```
ResourceSet rset = getResourceSet();

TransactionalEditingDomain domain =
        TransactionalEditingDomain.Factory
                        .INSTANCE.createEditingDomain(rset);
```

  C'est la solution, si l'on ne souhaite pas partager le domaine d'édition transactionnel.

- **Registry**

```
TransactionalEditingDomain.Registry.INSTANCE
                        .add("org.eclipse.example.MyDomain", domain);
```

  Cette solution est utile lorsque l'on souhaite partager le domaine d'édition transactionnel avec plusieurs clients, qui peuvent être des plugins par exemple.

On a vu qu'EMF fournissait des listeners en 1.3, pour les objets du modèle. L'API pour les transactions fournie elle aussi des listeners qui sont attachés automatiquement au domaine en cours d'édition. La planification dans EMF est strictement sérielle. Deux modes d'accès à l'interface ResourceSet.

1.4.1 Lecture

Le thread T_l demande un verrou exclusif sur la Resource, l'écriture est interdite, les autres Threads sont bloqués jusqu'à ce que T_l libère la Resource. L'opération *lecture*, doit être encapsulée dans une interface RunnableWithResult qui hérite de l'interface Runnable, mais qui permet d'avoir certain paramètre en retour comme le statut par exemple. Pour lire un object dans le domaine d'édition transactionnel, on doit exécuter la méthode runExlusive, de la classe TransactionalEditingDomain, qui prend en paramètre un objet de type RunnableWithResult.Impl, qui est une implémentation de l'interface RunnableWithResult.

Exemple . Recherche de "Richmond branch" dans le système.

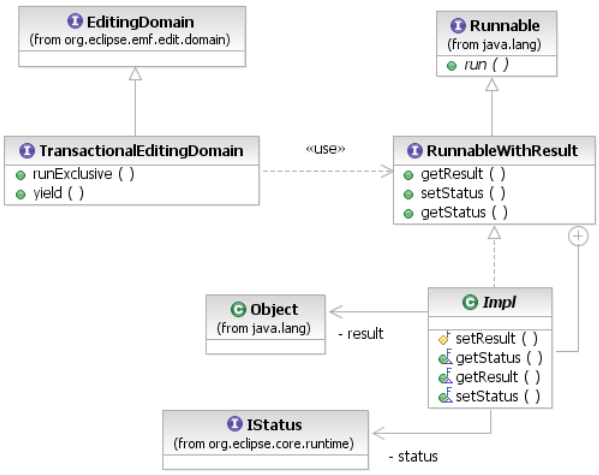
```
TransactionalEditingDomain domain;
Library richmond = (Library) domain.runExclusive(new RunnableWithResult.Impl() {
    public void run() {
        TreeIterator iter = resource.getAllContents();

        while (iter.hasNext()) {
            Object next = iter.next();

            if (next instanceof Library) {
                Library lib = (Library) next;

                if ("Richmond Branch".equals(lib.getName()) {
                    setResult(lib);  // found it
```

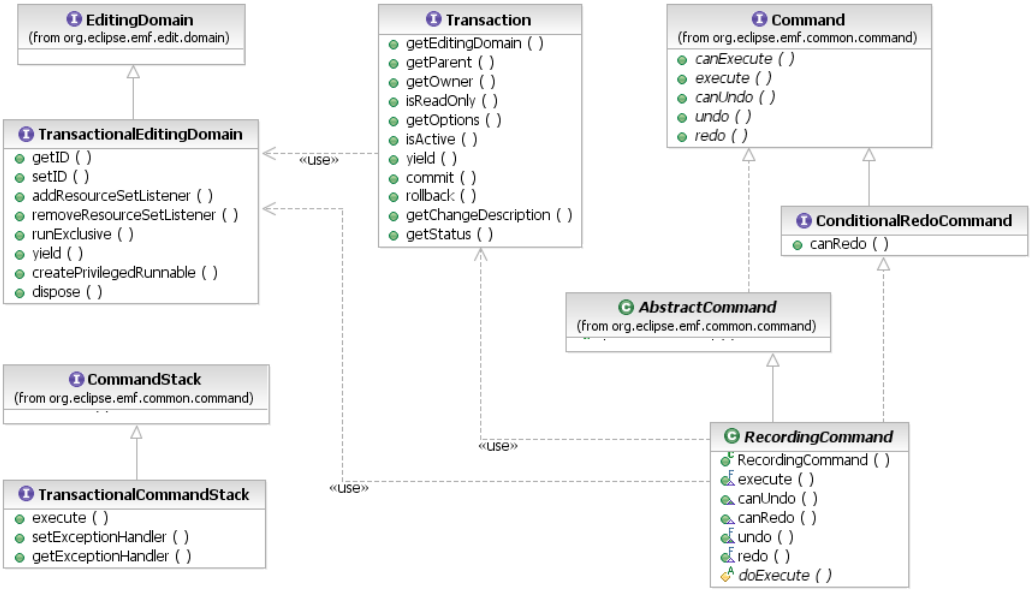
```
                    break;
                }
            }
        }
    }});
System.out.println("Found: " + richmond);
```



1.4.2 Rédacteurs

Les modifications sont réalisées au travers du Design Pattern *command*

Le Design Pattern *command* Ce pattern décrit des techniques pour garder en mémoire des séquences d'opérations, ainsi on peut annuler une opération en cas d'échec, et revenir dans un état cohérent.



Les modifications effectuées dans le domaine sont obtenues par des commandes.

- L'interface **Command**, est le cœur de ce Design Pattern, elle définit les méthodes execute(), Undo() et Redo(), la méthode canExecute permet de tester si la commande est exécutable, souvent elle est utilisée pour contrôler l'activation des actions liées à cette commande[5]. La méthode canUndo permet de vérifier si une commande peut-être annulée.
- L'interface **Transaction** définie une transaction, dans cette interface on peut voir la méthode commit() qui permet de valider la transaction, la méthode rollBack() qui permet d'annuler en cas d'échec, is ReadOnly qui permet de savoir si un Thread est en lecture sur une Resource.
- L'interface **TransactionalCommandStack** permet de maintenir une pile de *command*, toute modification dans le modèle transaction est stockée dans une pile de command, elle est le plus souvent utilisée pour bénéficier de Undo/Redo pour les *command*.
- L'interface **TransactionnalEditingDomain** permet de définir un domaine d'édition transactionnel, il est le support de l'édition d'un domaine, on peut voir les méthodes getID() et setID(), qui sont les identifiants du domaine d'édition transactionnel.

1.4.3 Partager le domaine d'édition transactionnel

EMF permet de partagé, un domaine d'édition transactionnel, avec des plug-ins au travers de l'interface Registry.

Exemple . GMF (Graphical Modeling Framework) Une fois le modèle crée avec EMF, on a besoin d'un éditeur graphique pour manipuler des modèles conformes au métamodèle Ecore, c'est le rôle de GMF. Ce FrameWork est basé sur EMF et GEF (Graphical Editing FrameWork). Il offre la possibilité de modifier les modèles EMF sous-jacents, dans ce cas GMF va se servir de l'API des transactions qu'offre

EMF. Ils utilisent le même domaine d'édition transactionnel, ainsi que la même pile de *command*. L'application de modélisation Payrus est basée sur EMF et GMF par exemple. La première chose à faire est de récupérer le domaine d'édition transactionel à l'aide de l'identifiant donné dans le registre

```
TransactionalEditingDomain shared = TransactionalEditingDomain
.Registry.INSTANCE                                    .getEditingDomain("org.e
```

Pour toutes modifications du modèle EMF, il suffit de faire :

```
transactionalEditingDomain.getCommandStack().execute(cmd);
```

Pour toutes modifications ayant un impact sur le diagramme GMF il suffit de faire :

```
getDiagramEditDomain().getDiagramCommandStack().execute(cmd);
```

"cmd" représente une modification de type Command.

1.4.4 Les écouteurs

Dans le cas où une transaction échoue, le mécanisme de notification d'EMF ne fonctionnera pas, et le client n'obtiendra aucune notification, puisqu'il n'y a pas eu de modification éffectuée, cela satisfait la partie atomicité des propriétés ACID vu en 1.3 . Pour répondre à ce problème l'API des transactions d'EMF fournie deux types d'écouteurs, *pre-commit*, exécuté avant la fin de la transaction elle notifie que la transaction a été terminée avec succès et *post-commit*, qui sera exécuté après la fin de la transaction.

Le type d'un évènement reçu est ResourceSetChangeEvent, cette classe permet de récupérer le domaine d'édition en cours, des informations sur la transaction, ainsi que la liste des notifications. Par exemple le site [5] donne cet exemple.

```
public void resourceSetChanged(ResourceSetChangeEvent event) {
        System.out.println("Domain " + event.getEditingDomain().getID() +
                " changed " + event.getNotifications().size() + " times");
    }
```

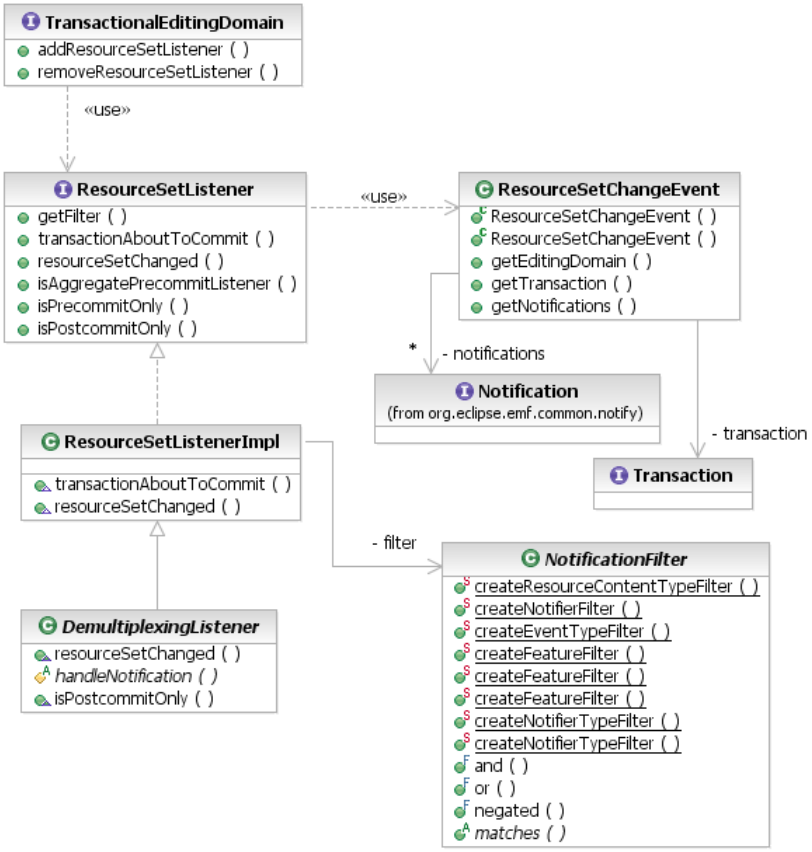


FIG. 1.3 – *Listeners EMF*

2 Les transactions dans la couche de persistance

2.1 Qu'est-ce que EclipseLink ?

EclipseLink est un ORM (Object Relationnel Mapping) proposé par la fondation Eclipse, pour JAVA SE ou JAVA EE, il implémente les spécifications de JPA et de JAXB. Il est basé sur TopLink qui est développé par la société Oracle.

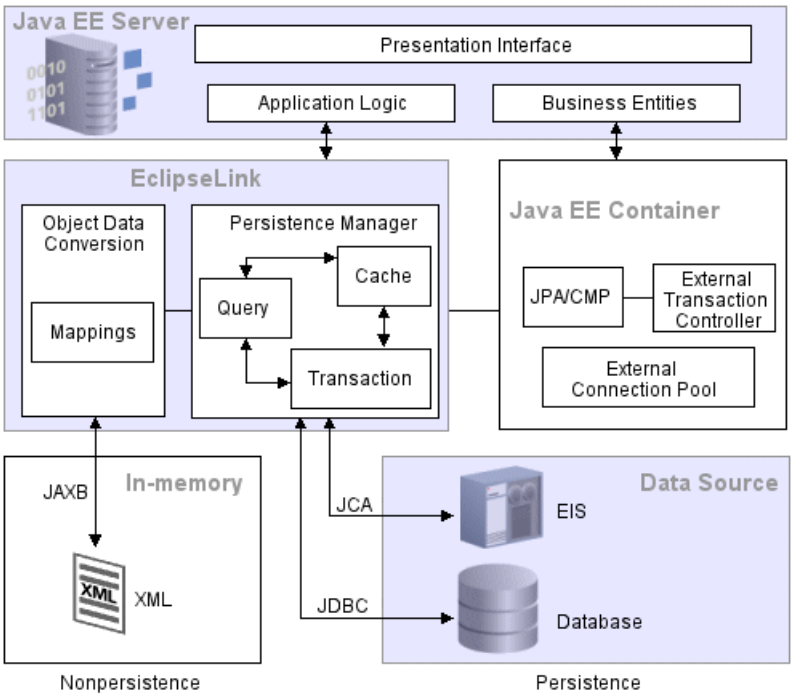


FIG. 2.1 – *EclipseLink dans un environnement JEE*

2.2 Les transactions avec EclipseLink

Dans un environnement JEE, les transactions peuvent-être gérées par le serveur d'application via l'API JTA, ou bien par l'unité de travail. Chaque transaction est

isolée des autres transactions, elles ont leur propre espace mémoire. EclipseLink supporte les modes optimistes et pessimistes.

2.2.1 Mode optimiste

Il suffit de mettre une annotation particulière sur un attribut qui deviendra un attribut de version de type long, int ..., cet attribut sera incrémenté à chaque fois, EclipseLink va mettre à jour cet attribut après chaque modification. Si une transaction T_1 veut accéder en écriture sur une entité contenant un attribut de version, EclipseLink va comparer les valeurs de cet attribut avec celle qu'il a conserver en mémoire. Si les valeurs sont les mêmes EclipseLink valide les modifications, si les attributs n'ont pas les mêmes valeurs, EclipseLink considère qu'il s'agit d'une tentative d'écriture sur cette entité et interdit l'accès, ceci n'est utilisable que pour un seul attribut, cette solution est fournie en standard par JPA [3]. EclipseLink permet d'autres types de mode optimiste [1].

- **ALL_Column :** Ce type compare chaque champ de la clause WHERE
- **CHANGED_COLUMN :** Ce type compare uniquement les champs modifiés dans la clause WHERE
- **SELECTED_COLUMN :** Ce type compare uniquement les champs sélectionnés de la clause WHERE, ils ne doivent pas être des clés primaires.

2.2.2 Mode pessimiste

A la section 2.1.1 on a vu que dans le mode pessimiste toutes transactions voulant écrire sur une donnée, pose un verrou exclusif sur cette donnée. D'une manière générale, on ajoute l'instruction FOR UPDATE à la requête, cette instruction va permettre à la transaction de poser un verrou en même temps qu'elle lit la donnée, lorsque dans JPA on choisit l'instruction LockModeType.PESSIMISTIC_WRITE, JPA ajoute à la requête l'instruction FOR UPDATE. EclipseLink procède différemment, il va redéfinir les paramètres de verrouillage. Il n'existe pas vraiment de techniques pour réaliser un verrouillage pessimiste, JPA recommande d'utiliser le mode optimiste, de plus le mode pessimiste peut engendrer des problèmes de concurrences, par exemple l'interblocage.

3 Conclusion

On a pu voir tout au long de ce dossier, ce que Eclipse proposait, dans le domaine des transactions, pour créer des applications et persister les données. Ces deux aspects peuvent se relier au travers de plusieurs projets, CDO, utilisé pour du travail collaboratif. Il permet d'enregistrer des modèles EMF au travers d'un ORM de type EclipseLink, ou Hibernate. Un autre projet Texo, anciennement Tenneo, qui génère à partir des diagrammes Ecore, du code JAVA avec la gestion de persistance. Ces deux projets utilisent aussi la programmation transactionnelle. Éclipse offre comme on la vu la possibilité de mettre en œuvre les transactions à partir de la modélisation jusqu'à la persistance d'objet. C'est un point que j'ai trouvé très intéressant.

Le sujet fait appel à de nombreuses notions, et dans le temps qu'il m'a été alloué je n'ai pu que survoler ces notions. Mon travail a été de relier tous ces concepts entre eux. Je pense à EMF qui est très riche en fonctionnalité dispose de nombreux outils supplémentaires. J'ai beaucoup apprécié de travailler sur ce sujet pour cette richesse culturelle, ainsi que toutes les compétences qu'il m'a permis d'acquérir. Ce travail a modifié mon point de vue sur l'importance d'avoir un système cohérent et intègre. Mon point de vue a aussi changé dans des domaines collatéraux au sujet, comme la place des modèles dans le cycle de vie d'une application. Ma vision d'Eclipse a elle aussi été modifiée, puisque je le voyais juste en tant qu'IDE JAVA avec quelques plug-ins, suite à ce projet, je m'aperçois que c'était une vision bien restrictive.

Éclipse est riche en fonctionnalité et propose des outils dans tous les domaines, avec la possibilité d'en créer aux besoins, comme on a pu l'entrevoir. La programmation transactionnelle au sein d'Éclipse fait partie de ces possibilités.

Bibliographie

[1] Fondation Eclipse. Eclipselink solutions guide for eclipselink release 2.5.

[2] Fondation Eclipse. Emf model transaction developer guide.

[3] Fondation Eclipse. Understanding eclipselink, 2.5.

[4] Jim Gray Andrea Reuteur. *Transaction Processing: Concepts and Techniques Management*. Morgan Kaufmann.

[5] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley, Boston, MA, 2 edition, 2009.

[6] Carolyn BEGG Thomas CONNOLY. *Systèmes de base de donnés, approche pratique de la conception de l'implémentation et de l'administration*. 2005.