Conception détaillée

Objectifs du chapitre

Nous arrivons maintenant à la phase ultime de modélisation avec UML. Après la modélisation des besoins puis l'organisation de la structure de la solution, la conception détaillée consiste à construire et à documenter précisément les classes, les interfaces, les tables et les méthodes qui constituent le codage de la solution. Il s'agit de :

- comprendre le rôle d'UML pour la conception détaillée ;
- savoir appliquer le micro-processus utilisé pour bâtir une conception objet avec UML ;
- apprendre à construire une solution pour : la couche de présentation, la couche d'application et la couche métier distribuée dont l'EAI ;
- savoir transformer un modèle objet en modèle relationnel.

Quand intervient la conception détaillée ?

La conception détaillée est une activité qui s'inscrit dans l'organisation définie par la conception préliminaire. Le modèle logique y est particulièrement important dans la mesure où c'est en conception détaillée que l'on génère le plus gros volume d'informations. Il est ainsi possible de confier les catégories à des personnes différentes, qui pourront travailler indépendamment les unes des autres. La conception détaillée s'appuie donc sur les catégories de conception organisées à la fois suivant les *frameworks* techniques et les regroupements propres au métier. Les concepteurs construisent alors les

classes, les vues d'IHM, les interfaces, les tables et les méthodes qui vont donner une image « prête à coder » de la solution.

En dernier lieu, il convient de préciser le contenu des sous-systèmes de manière à compléter la configuration logicielle. Le niveau d'abstraction visé par l'étape de conception détaillée est la conception des composants. Il s'agit d'avoir une idée la plus précise possible pour la fabrication et l'assemblage des sous-systèmes de configuration logicielle.

La conception détaillée précède la phase de codage. À ce niveau, toutes les questions relatives à l'agencement et aux détails de la solution doivent être modélisées. Ainsi, les interrogations restantes concernent exclusivement la bonne utilisation des langages et des outils de développement.

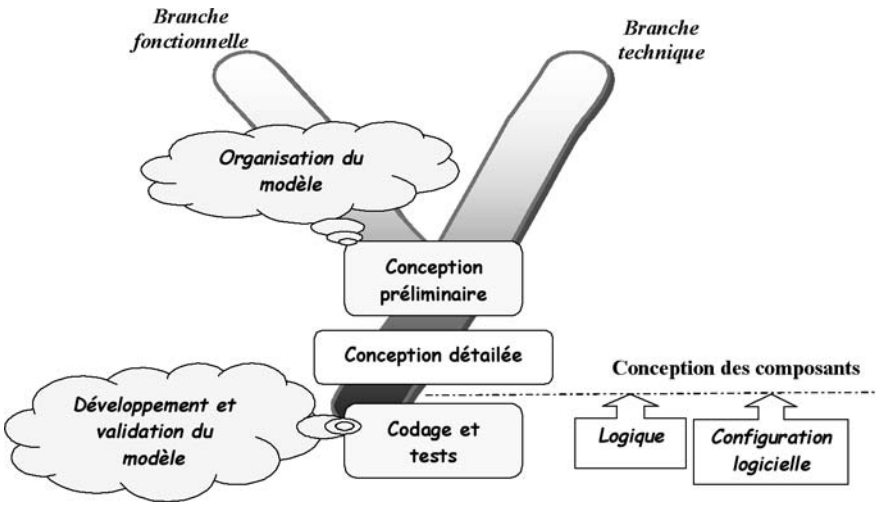

Figure 11-1 : Situation de la conception détaillée dans 2TUP

Éléments mis en jeu

- Micro-processus de conception logique, modèle logique,
- propriétés de conception d'une classe, d'un attribut, d'une association et d'une opération,
- *design patterns* : État, Itérateur, Curseur, Stratégie, Observateur, Référence futée,
- couches de présentation, de l'application, de métier distribué et de stockage des données,

- IHM, distribution RMI, passage d'un modèle objet à un modèle relationnel,
- modèle d'exploitation consolidé et modèle de configuration logicielle détaillée.

Le micro-processus de conception logique

Le micro-processus de conception logique concerne la définition des classes à implémenter. C'est donc une activité centrée sur le modèle logique, qui combine les diagrammes UML suivants :

- principalement les diagrammes de classes pour préciser la structure des classes de développement,
- mais aussi les diagrammes d'interactions pour préciser les communications entre objets,
- et les diagrammes d'activité pour exprimer les algorithmes des méthodes.

Enfin, il est possible de recourir à du pseudo-code pour les algorithmes les plus complexes. Il s'agit en fait d'esquisser le code des méthodes à développer. En l'occurrence, nous utilisons une formulation proche de Java.

Le micro-processus consiste en une itération des cinq activités représentées à la figure 11-2.

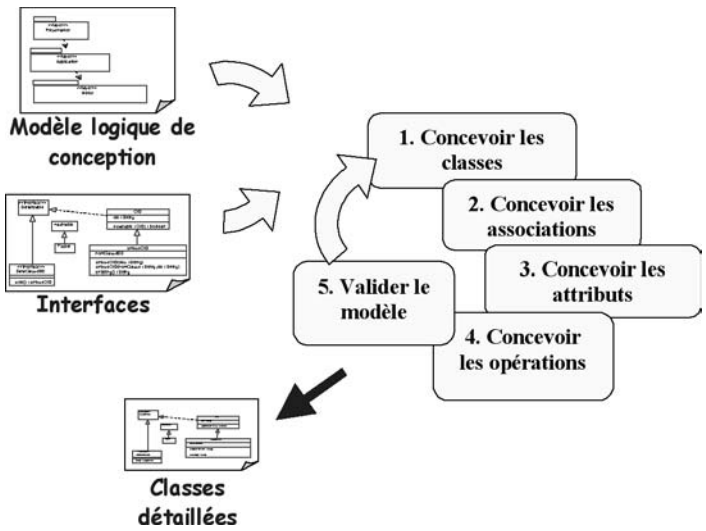

Figure 11-2 : Micro-processus de conception détaillée

- Concevoir les classes consiste à transformer des concepts provenant de l'analyse, tels que les métaclasses ou les classes à états parallèles, en techniques disponibles avec les langages et les outils de développement.
- Concevoir les associations définit la façon d'exploiter chaque association et les techniques qui seront employées dans le codage.
- Concevoir les attributs nécessite essentiellement d'identifier les structures de données, les itérations et d'autres types complexes permettant de représenter les attributs d'analyse avec le langage utilisé.
- Concevoir les opérations permet de déterminer le contenu des méthodes complexes et d'identifier en cascade de nouvelles classes et opérations dans la catégorie.
- Valider le modèle constitue la phase de décision du cycle itératif. Sortir de ce cycle signifie que le modèle donne l'image prête à coder de ses composants de configuration logicielle.

Nous allons étudier tour à tour ces activités, puis voir leur application sur les différentes couches du système SIVEx.

Concevoir les classes

Les classes qui proviennent de l'analyse ne sont pas toujours conformes aux possibilités du langage d'implémentation. Dans certains cas, une analyse orientée objet est réalisée dans un langage non objet. La transformation des classes en codage est alors particulièrement importante pour conserver la trace du passage de l'analyse au code. Java offre bien entendu une transformation beaucoup plus directe. Certaines formes telles que les métaclasses, les états ou les héritages multiples sont cependant tolérées par les analystes mais inconnues de Java. Concevoir les classes consiste tout d'abord à expliciter comment ces concepts devront être traduits dans le code.

Concevoir les classes, c'est aussi en introduire de nouvelles soit pour prendre en charge des responsabilités purement techniques, soit pour décharger une classe d'analyse de certains de ces aspects techniques. Les liens qui rattachent ces classes entre elles doivent le plus souvent correspondre à des *design patterns*. On trouvera à terme des classes comme des « fabriques », des « archiveurs », des « traceurs », des « vérificateurs de cohérence », etc.

Concevoir les classes, c'est enfin redistribuer les messages et les événements du modèle dynamique sur les différentes couches de conception – nous verrons notamment comment réaliser la conception orientée objet des messages identifiés dans les interfaces EAI. Il est probable en effet que ces concepts vus de manière abstraite par l'analyste ne correspondent plus aux principes de conception. Pour les systèmes 3-tiers, nous pensons plus

particulièrement à la redistribution des échanges entre les couches métier et application. Ces échanges s'appuyant sur les capacités d'un réseau physique doivent faire l'objet d'optimisations. Dans cette optique, il convient que les diagrammes d'états soient retravaillés au niveau de la conception.


Définition

LE DESIGN PATTERN ÉTAT

Le *design pattern* État [Gamma 95] est une façon de concevoir le diagramme d'états d'une classe d'analyse. La gestion des états est déléguée de sorte qu'à chaque état corresponde une classe du patron. Une classe gère ainsi les activités et les transitions attachées à l'état qu'elle représente.

Le diagramme d'états du suivi de mission sert à illustrer l'application de ce *design pattern*. Chaque état de la classe correspond à une spécialisation de la classe *SuiviMissionEtat*. Les événements du diagramme d'états deviennent des opérations polymorphes pour les classes *États*. L'interprétation du diagramme d'états de la figure 11-3 donne ainsi une conception de la classe *SuiviMission*, accompagnée d'un environnement de gestion de ses états (voir figure 11-4).

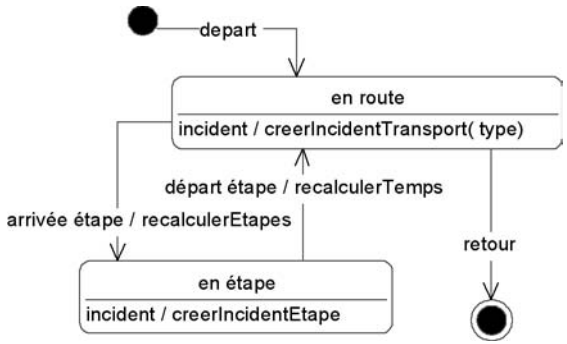

Figure 11-3 : Diagramme d'états de la classe SuiviMission

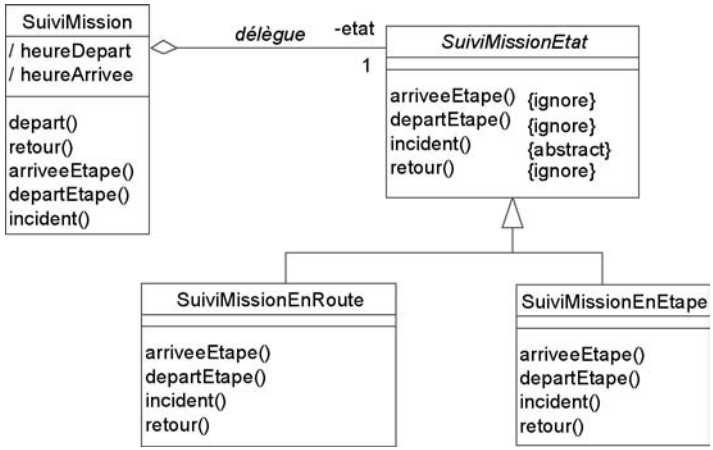

Figure 11-4 : Environnement résultant des états de la classe SuiviMission

La classe *SuiviMission* délègue la gestion de ses états à la classe *SuiviMission Etat* ; en d'autres termes, elle transmet systématiquement les événements qu'elle reçoit. Les états de la classe sont ensuite chargés de déclencher les opérations et d'assurer les transitions. Le diagramme de communication ci-après vous montre comment un tel mécanisme peut être documenté.

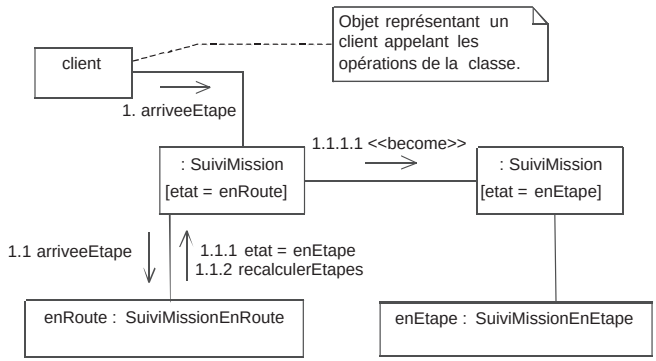

Figure 11-5 : Dynamique d'un changement d'état sur une arrivée d'étape

Le *design pattern* État réduit la complexité des méthodes par extraction des instructions d'aiguillage nécessaires à la gestion des transitions. De ce fait, il facilite l'ajout de nouveaux états. La prise en compte d'un super-état se résout tout aussi facilement par l'introduction d'une super-classe état (voir l'étude de cas ci-après).

Lorsque les classes représentant les états ne possèdent aucun attribut, il est souvent possible d'en faire des singletons. L'instruction associée à une transition d'état prendra alors la forme suivante : *etat = SuiviMissionEnEtape.getInstance()*.

ÉTUDE DE CAS : CONCEPTION DES ÉTATS DE LA CLASSE *MISSION*

La réalisation des états de la classe *Mission* s'établit à partir du diagramme d'états de la classe d'analyse (voir chapitre 8). Le diagramme d'états élaboré par l'analyste regroupe à la fois des aspects de niveau application et métier. En effet, les transitions et les activités qui concernent l'affectation des valeurs sont d'ordre applicatif, tandis que les états de création, validation, annulation et réalisation concernent la couche métier. Le concepteur s'appuie donc sur de nouveaux diagrammes d'états de conception pour les classes *Metier::Mission::Mission* et *Application::Mission::DocMission*. À cet égard, reportez-vous respectivement aux figures 11-6 et 11-8.

Au niveau de la couche métier, les événements d'affectation et de modification sont simplifiés de manière à alléger les échanges RMI lors de la création d'une nouvelle mission. Il n'en résulte donc que deux sous-états, suivant l'état correct ou incorrect déterminé par une opération de vérification.

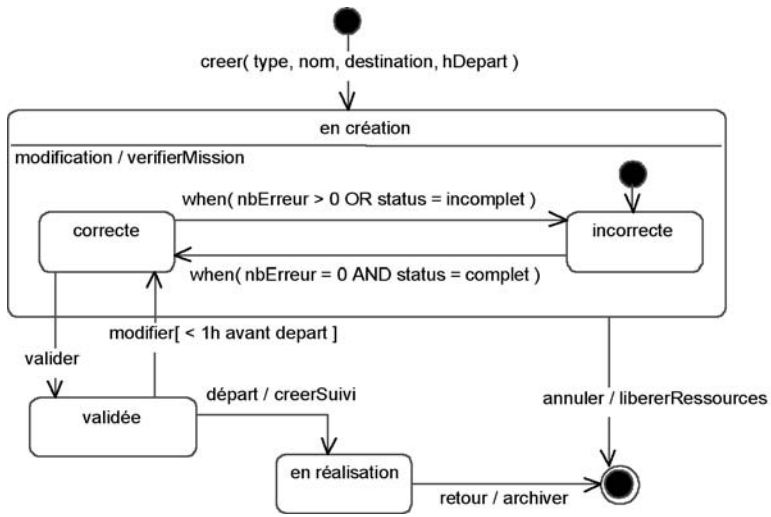

Figure 11-6 : Diagramme d'états simplifié de la classe Mission au niveau de la couche Métier

L'environnement de la classe *Mission* après l'application du design pattern État est donc illustré à la figure 11-7. Vous remarquerez le devenir des sous-états en tant que sous-classes dans la hiérarchie des classes d'état. Toutes les sous-classes finales dans la hiérarchie des états implémentent le design pattern singleton.

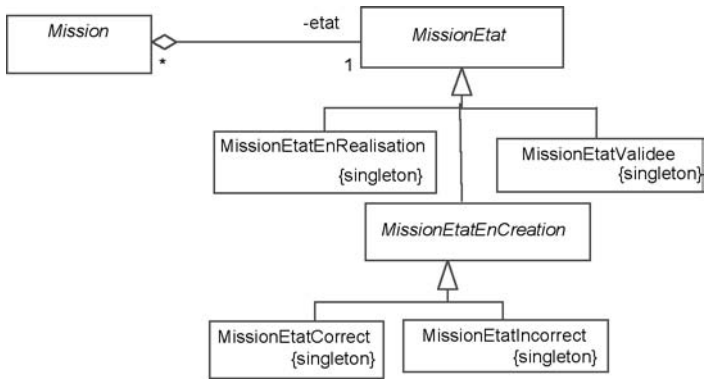

Figure 11-7 : Environnement de gestion d'états de la classe Métier::Mission::Mission

Pour fixer les idées, voici le code Java correspondant à la structure et la distribution des opérations sur les classes *État*. En premier lieu la classe *Mission* délègue à son état le traitement des événements du diagramme d'états.

```
package        SIVEx.metier.mission.mission;
import           SIVEx.metier.mission.mission.etats;
…
public class Mission {
      private MissionEtat _etat;
      …
      // Opération réservée qui permet aux états de transiter :
      public void setEtat( MissionEtat newEtat) {
         _etat = newEtat;
      }
      // réception des événements du diagramme d'états :
      public void modification(this, …) {
         _etat.onModification(this);
      }
      public void erreursOuIncomplete() {
         _etat.onErreurOuIncomplete(this);
      }
      public void justeEtComplete() {
         _etat.onJusteEtComplete(this);
      }
      public void valider() {
         _etat.onValider(this);
      }
      // etc…
}
```

Par la suite, chaque état gère les transitions sur la classe *Mission*. Les classes représentant des super-états fournissent des opérations factices, qui peuvent déclencher un message d'erreur, dans la mesure où un fonctionnement normal ne devrait jamais les utiliser.

```
package        SIVEx.metier.mission.mission.etats;
…
abstract public class MissionEtat {
      public void modification( Mission sujet) {
         FichierTrace.instance().tracer( «ERREUR : états mission : …»);
      }
      // etc…
}
```

```
package     SIVEx.metier.mission.mission.etats;
…
public class MissionEtatEnCreation extends MissionEtat {
      public void onModification( Mission sujet, …) {
         // traitement de l'événement au niveau du super-état.
         sujet.modifier( …);
         sujet.verifier(); // déclenche de nouveaux événements.
      };
}
```

```
package        SIVEx.metier.mission.mission.etats;
…
public class MissionEtatEnCreationCorrecte extends MissionEtatEnCeation {
      // réalisation du singleton :
      private static MissionEtat _instance;
      public static MissionEtat instance() {…}
      // réalisation des événements qui concernent le sous-état :
      public void onValider(Mission sujet) {
         // transition :
         sujet.setEtat( MissionEtatValidee.getInstance());
      }
}
```

En ce qui concerne la couche application, le nouveau diagramme d'états montre la prise en compte des contrôles au niveau du poste client, à charge pour la classe *Mission* de fournir une seule opération de vérification lors de la création ou de la modification d'une mission dans la couche métier. Le design pattern État est également applicable au niveau de la classe *DocMission*.

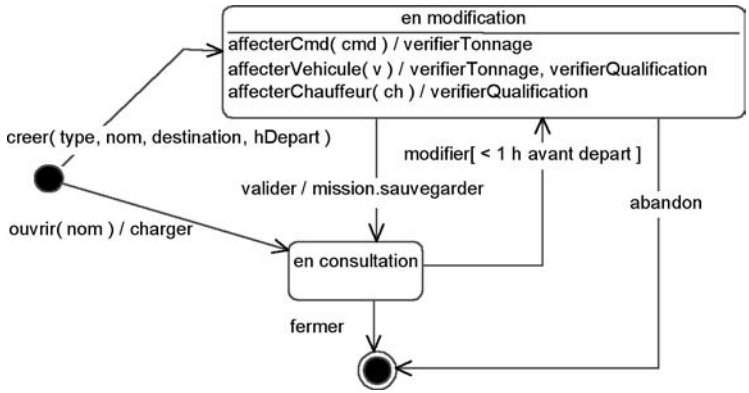

Figure 11-8 : Diagramme d'états de la classe DocMission au niveau de la couche application

L'identification des classes à partir des interfaces EAI illustre également comment les éléments du modèle dynamique, cette fois-ci les messages, alimentent de nouvelles classes. En effet, pour des raisons de maintenance des modèles d'échanges, il est important de donner une tournure orientée objet à l'EAI, même si les outils qui l'implémentent proposent par défaut une séparation forte des données et des fonctions.

Le travail réalisé en conception préliminaire donne déjà une première orientation objet au travers de la matrice qui a servi à identifier les interfaces EAI et par la sémantique « objet.verbe » des messages utilisés dans les diagrammes de séquence.

ÉTUDE DE CAS : CONCEPTION DES MESSAGES EAI « CLIENT »

À partir de l'identification de l'objet client, présent sur plusieurs interfaces d'échanges EAI, le travail consiste à établir la structure d'un client au vu des différentes interfaces auxquelles il participe.

Typiquement, un message EAI est composé d'un en-tête contenant les informations d'identification de l'objet, et ce afin de minimiser le volume d'informations à envoyer lorsqu'une simple référence à l'objet concerné par le message suffit. Par ailleurs, et suivant les différentes interfaces, l'échange d'un objet client est assorti d'adresses, d'informations comptables ou de contacts. En conséquence, l'utilisation d'un diagramme de classes supporte la conception orientée objet des messages EAI en mettant en valeur les agrégations, voire les héritages, nécessaires à la structuration des données échangées.

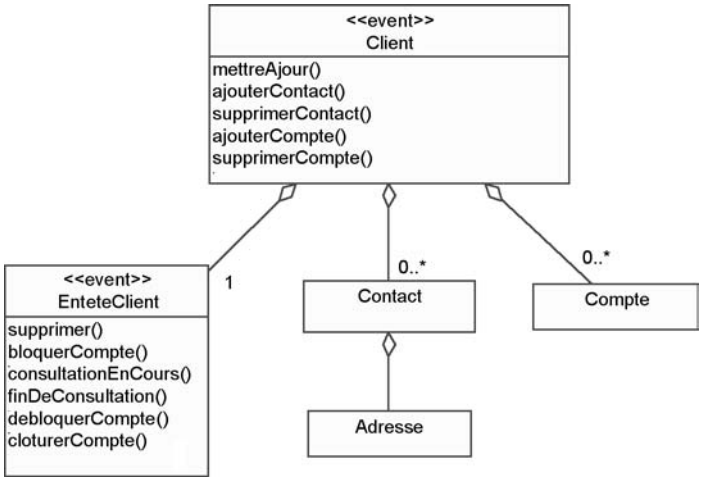

Figure 11-9 : Conception orientée objet des messages identifiés dans les interfaces EAI

Le diagramme ci-dessus montre la conception des messages EAI, remarquez la réutilisation du stéréotype « event » pour différencier les classes qui sont échangées des blocs d'attributs qui les accompagnent. Par ailleurs, la transformation des verbes, provenant des messages identifiés en conception préliminaire, en opérations permet d'identifier rapidement quelle structure de données accompagne le message. Ainsi le message « client.supprimer » transporte la structure de données décrite par la classe *EnteteClient*, tandis que le message « client.mettreAjour » transporte celle qui correspond à la classe *Client*.

Concevoir les associations

L'association est un concept inconnu de la plupart des langages orientés objet. Elle se transforme en attribut ou en tableau d'attributs suivant sa multiplicité. La figure 11-10 montre l'application de ce principe à deux associations de la classe *Métier::MissionTraction*.

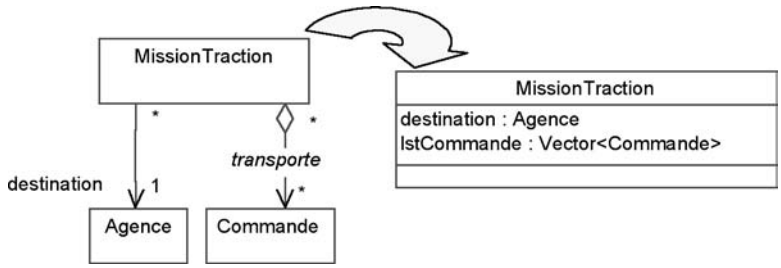

Figure 11-10 : Exemple d'une conception d'associations par des attributs

La figure ci-dessus montre qu'une agrégation, outre la précision sémantique qu'elle apporte en analyse, n'est pratiquement jamais prise en compte en conception. Remarquons également que l'utilisation du tableau générique *Vector* ou *ArrayList* de Java ne permet pas d'exprimer le type d'éléments stockés (nous donnons par la suite indifféremment des exemples avec *Vector* ou *ArrayList*, le principe est le même). L'usage d'une notation empruntée au *template* de C++ permet de conserver cette information sur le diagramme. Remarquons enfin que la conception des associations est facilitée par l'expression de leur navigabilité.

Il serait cependant fastidieux de transformer tous les diagrammes de conception, d'autant que la conception d'une association s'accompagne d'un ensemble d'opérations nécessaires à sa gestion. On utilise donc une valeur étiquetée, *design tip*, pour désigner les mécanismes d'association accompagnés de leurs opérations de gestion. Souvenez-vous, nous avons déjà utilisé la même technique en conception générique. Ce mécanisme doit donc être documenté dans le modèle, comme l'illustre la figure ci-dessous.

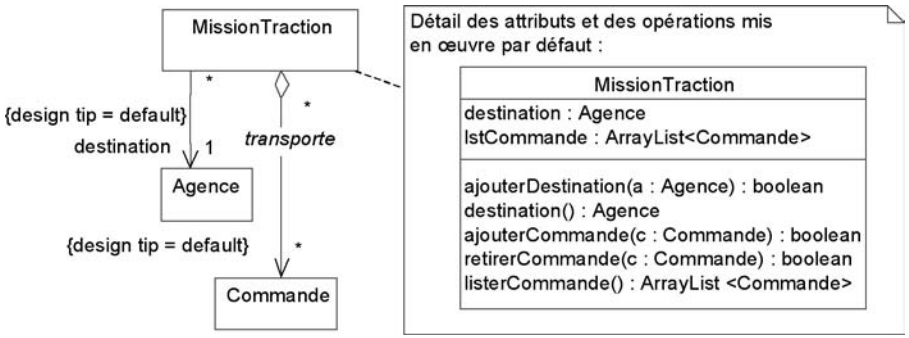

Figure 11-11 : Documentation du mécanisme {design tip = default}

La visibilité des opérations de gestion de l'association étant identique à la visibilité du rôle, il est important d'introduire cette information en conception détaillée.

La conception d'une association se complique lorsqu'elle comporte des contraintes à respecter. On peut considérer deux types de contraintes différentes et adapter le style de conception en conséquence.

- Les contraintes de gestion sont exprimées par une multiplicité minimale supérieure à 0 (obligatoire), une composition, un qualifieur ou les propriétés d'UML {ordered}, {addOnly} et {frozen}. Ces contraintes ont des répercussions sur l'interface et les méthodes de gestion d'une seule association. Par exemple : l'opération de retrait peut disparaître du fait d'un addOnly ou d'un frozen ; le constructeur doit comporter des paramètres

supplémentaires pour initialiser les liens obligatoires. Une composition se traduit par une règle de structure : la destruction du composite implique celle de ses sous-parties ; cette contrainte a toutefois peu d'influence appliquée à Java du fait de son ramasse-miettes. Enfin, les qualifieurs transforment un *Vector* en *Hashtable* et une *ArrayList* en *HashMap* dont la clé est le qualifieur.

- D'autres contraintes structurelles portent sur plusieurs associations : il peut s'agir des contraintes utilisateur, des contraintes prédéfinies {XOR}, {subset} et {AND}, des associations bidirectionnelles ou d'une classe d'association. Ces contraintes induisent des postconditions aux opérations de gestion de l'association. Les associations bidirectionnelles sont conçues exactement comme deux associations réciproques, dont les attributs de part et d'autre sont synchronisés par des règles de gestion. Une classe d'association devient un point central qui gère la relation avec les deux autres classes.

Concevoir les associations consiste enfin à transformer les relations n-aires, tolérées par les analystes, en relations binaires alors plus faciles à concevoir, ou bien à ajouter une nouvelle classe qui pilote la relation complexe.

Dans un second temps, il est possible d'optimiser les méthodes de gestion. Le cas le plus fréquent concerne la consultation d'une liste d'objets en mode client/serveur. D'une part, le transfert d'un groupe d'objets est coûteux pour le réseau, alors qu'il suffit d'afficher un seul libellé caractéristique pour chaque objet. D'autre part, au-delà de 10 à 20 références, un utilisateur reformule la plupart du temps son critère de sélection. Pour éviter des temps d'attente, il est souvent utile de procéder à la gestion d'une liste de libellés caractéristiques, par un curseur ramenant 10 à 20 références à la fois.

ÉTUDE DE CAS : CONCEPTION DES ASSOCIATIONS DANS *MÉTIER::MISSION*

La conception des associations consiste ici à compléter le nom des rôles et la navigabilité des associations. Pour la plupart, une application du mécanisme par défaut suffit à décrire les attributs et les opérations d'implémentation.

Seule la classe d'association liant une mission de tournée à ses étapes doit être explicitée. Dans ce cas, le {*design tip = ignore*} signifie que l'association reste sur le diagramme aux seules fins d'une meilleure lisibilité, mais qu'elle est explicitement conçue par les attributs et les opérations disponibles sur la classe *MissionTournée*.

La figure ci-dessous montre la structure résultant des classes de la catégorie.

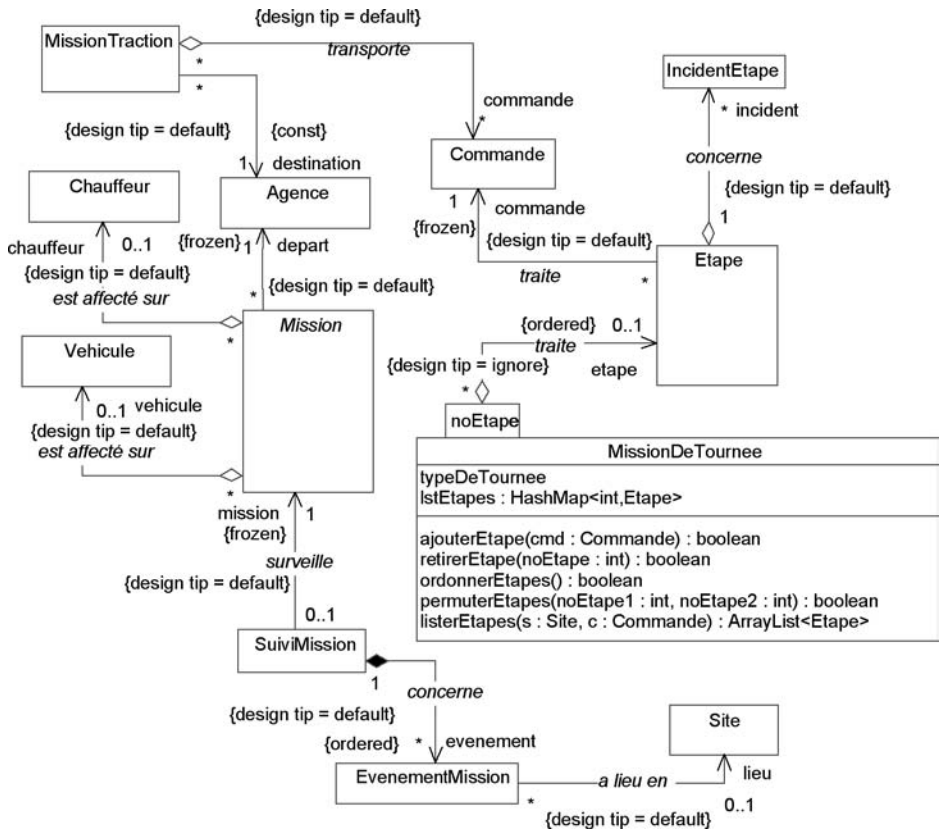

Figure 11-12 : Associations de la catégorie Mission, documentées pour la conception détaillée

Le code Java ci-dessous montre la réalisation par défaut de l'association entre la classe *Etape* et *IncidentEtape*. L'aspect systématique de ce code souligne l'avantage d'utiliser un générateur de code, conformément aux préconisations « Model Driven » de l'OMG

```
package                  SIVEx.metier.mission;
…
public class Etape {
      // réalisation de l'association avec les incidents d'étape :
      private Vector_incident;
      …
      // opérations de gestion :
      public Boolean ajouterIncident( IncidentEtape inc) {
         _incident.addElement( inc);
         return true;
      }
      public Boolean retirerIncident( IncidentEtape inc) {
         int _incident.removeElement( inc);
         return true;
```

```
      }
      public Vector listerIncidents() {
         Vector                              clone = _incident.clone();
         return clone;
      }
}
```

Définition

LE DESIGN PATTERN ITÉRATEUR

Le design pattern Itérateur [Gamma 95] est une façon de concevoir l'accès séquentiel à un ensemble d'objets, sans avoir à exposer la structure interne de l'ensemble. Par ailleurs, l'itérateur offre une interface de parcours standard qui uniformise et facilite la manipulation des différents types de liste d'un même projet. Il offre enfin la possibilité de parcourir simultanément et indépendamment le même ensemble d'objets par des tâches parallèles.

L'objectif de l'itérateur est de déléguer les opérations de parcours d'une association. Par ailleurs, il a pour rôle de gérer l'état d'un élément courant, auquel on peut accéder séquentiellement au précédent ou au suivant. L'itérateur est donc particulièrement approprié aux associations ordonnées (propriété *{ordered}*).

La structure d'un itérateur répond au diagramme de la figure 11-13 ; la classe responsable de gérer l'association sert naturellement de fabrique d'itérateur.

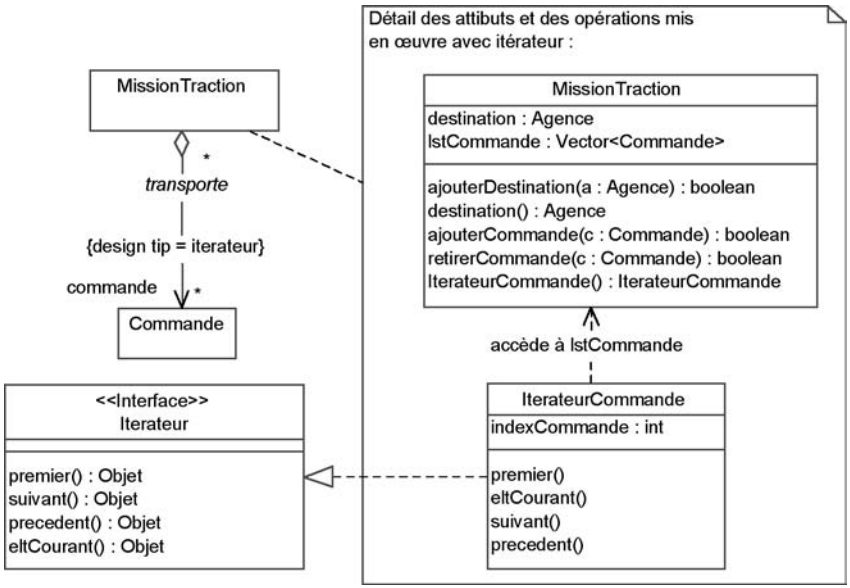

Figure 11-13 : Documentation du mécanisme {designTip=itérateur}

Un curseur est un dérivé d'itérateur qui renvoie à chaque requête un nombre déterminé d'éléments. Nous allons maintenant le mettre en œuvre dans le cadre de l'étude de cas.

ÉTUDE DE CAS : CONCEPTION D'UN CURSEUR ENTRE AGENCE ET MISSION

Nous allons maintenant passer à la phase d'optimisation des associations. L'accès aux missions d'une agence pour une plage de dates donnée est une opération fréquente du répartiteur, des opérateurs de quai et des réceptionnistes. Par ailleurs, cette opération est parallèle, car elle s'appuie sur le réseau et requiert par conséquent plus d'attention en matière de volume et de temps d'accès.

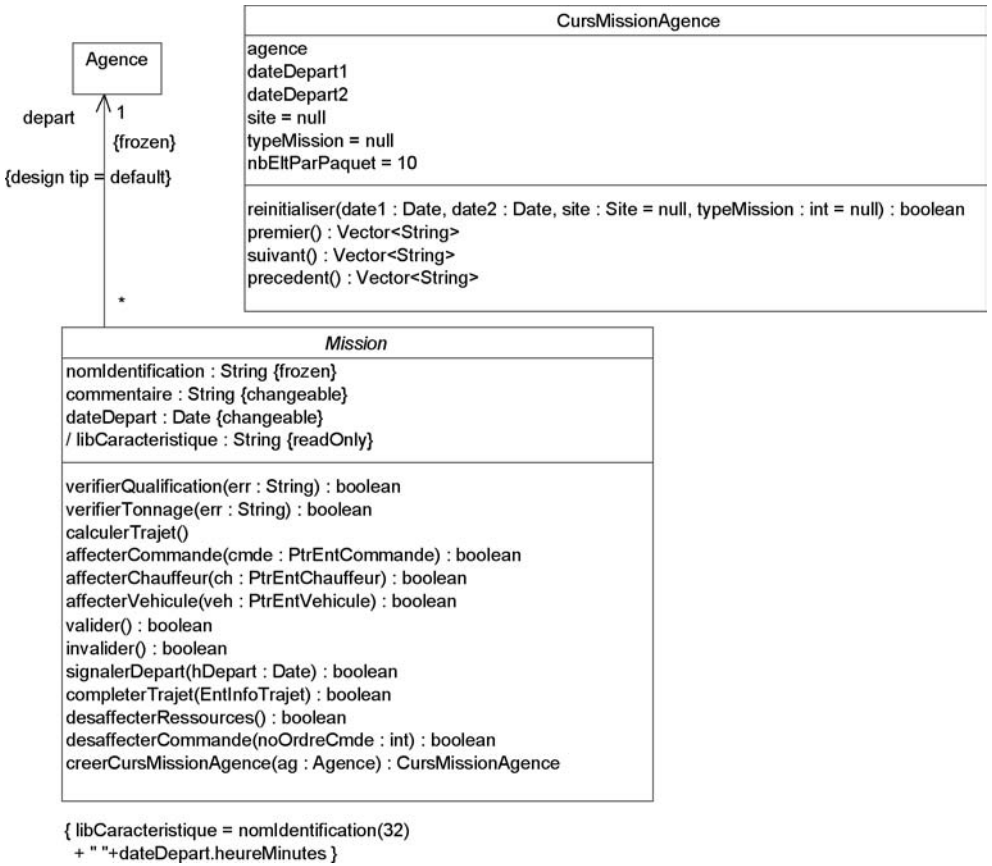

Figure 11-14 : Structure d'un curseur parcourant les missions concernées par une agence

Le développement d'une classe curseur parcourant les libellés caractéristiques des missions se justifie donc en prévision de la distribution aux applications clientes de l'information. En effet, des

listes de sélection de missions sont notamment gérées par les applications ConVEx déployées sur le poste de travail du réceptionniste.

La fabrication d'un curseur est une opération de classe ; elle correspond à une responsabilité sur la classe *Mission*. La construction d'un curseur spécifie une agence, car il s'agit de recenser les missions d'une agence. Cette technique permet de respecter le sens de la dépendance entre la catégorie *Ressource* à laquelle appartient le concept d'agence et la catégorie *Mission*.

Le curseur distribue des tableaux de libellés caractéristiques au travers des opérations *premier()*, *suivant()* et *precedent()*. L'attribut *nbEltParPaquet* permet de redimensionner la taille des tableaux.

Concevoir les attributs

La conception des attributs consiste principalement à définir le type des attributs identifiés en analyse. Bien que la plupart des attributs se satisfassent des types de base de Java, certains attributs d'analyse correspondent à une structure de données qu'il est nécessaire de spécifier. Dans ce cas, nous introduisons le stéréotype *struct* pour distinguer une simple structure de données d'une classe. Cette dernière s'apparente à un type de base du langage ; tous ses attributs sont publics, de sorte qu'elle nécessite rarement d'opérations associées. D'autres attributs induisent des énumérations pour lesquelles nous introduisons également le stéréotype *enum*. En Java, une énumération correspond à une classe composée d'attributs *public static final*.

Concevoir les attributs, c'est également spécifier leur visibilité et leur mode d'accès. Par défaut, un attribut est privé et ce principe reste invariable dans notre conception. La prise en compte des propriétés UML {changeable}, {readOnly} ou {frozen} est donc implicitement gérée par des opérations d'accès.

Pour assurer respectivement la modification et la lecture de la valeur de l'attribut, les attributs {changeable} requièrent deux opérations *set<nom attribut>* et *get<nom attribut>*. La propriété {frozen} implique l'initialisation de l'attribut dans le constructeur de la classe. Cela se traduit le plus souvent par un paramètre d'initialisation supplémentaire. Nous avons ajouté la propriété *{readOnly}* pour indiquer que l'attribut n'est disponible qu'en lecture grâce à l'opération get.

Enfin, concevoir les attributs, c'est spécifier les méthodes qui servent à la mise à jour des attributs dérivés. Il existe à cet effet plusieurs techniques, la plus simple consistant à invoquer une opération de mise à jour lors de l'accès à l'attribut. Certaines méthodes de calcul peuvent cependant être coûteuses ou fréquentes, aussi faut-il gérer un attribut d'état pour recalculer l'attribut dérivé uniquement si l'opération se justifie. La gestion des attributs dérivés dans une

application multitâche nécessite parfois la conception de synchronisations complexes.

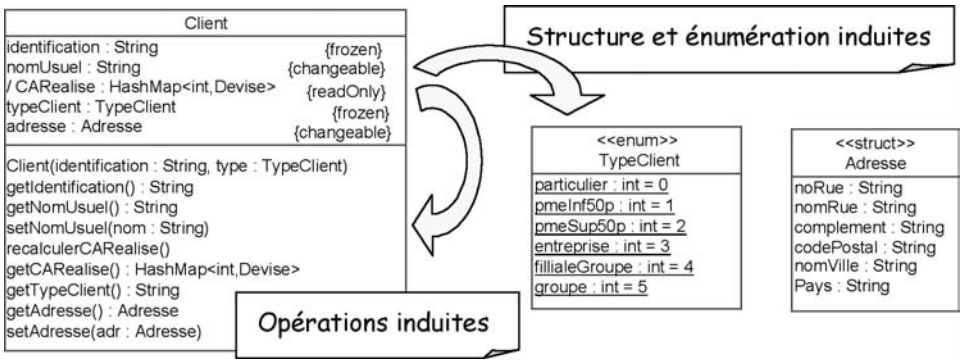

Figure 11-15 : Mécanismes induits par la définition des attributs

ÉTUDE DE CAS : CONCEPTION DES ATTRIBUTS DE *MÉTIER::MISSION*

La conception des attributs des classes de la catégorie *Mission* ne pose guère de problèmes, les méthodes de calcul des attributs dérivés correspondront à la transcription en langage Java des contraintes spécifiées par l'analyste.

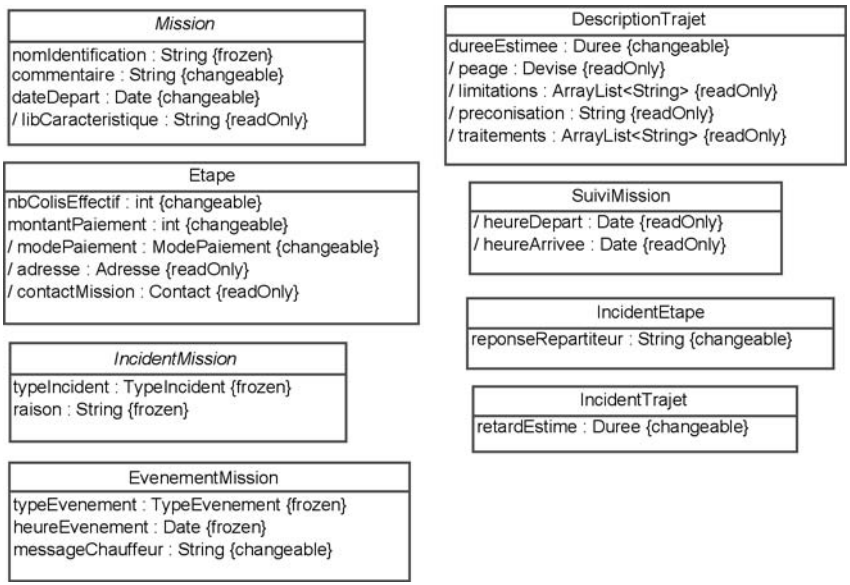

Figure 11-16 : Définition des attributs sur les classes de la catégorie Métier::Mission

À titre d'exemple, les structures et énumérations induites par la conception des attributs sont détaillées à la figure 11-17.

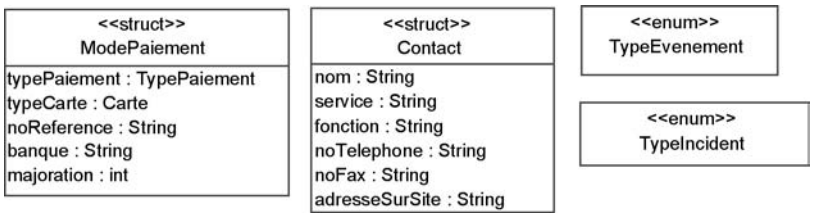

Figure 11-17 : Classes dérivées de la conception des attributs

Les choix de réalisation des attributs induisent donc la réalisation du code suivant. Nous avons pris pour exemple la classe *Mission* qui possède les trois types de propriétés.

```
package      SIVEx.metier.mission;
…
public class Mission {
      // réalisation des attributs :
      private String _nomIdentification;
      private String _commentaire;
      private Date _dateDepart;
      // libCaracteristique pas déclaré car c'est un attribut dérivé
      …
      // Constructeur avec les attributs frozen :
      public Mission( String nomIdentification) {
         _nomIdentification = nomIdentification;
      }
      // Accès en lecture :
      public String getNomIdentification() {
         return _nomIdentification;
      }
      public String getCommentaire() {…}
      public Date getDateDepart() {…}
      // Calcul des attributs dérivés :
      public String getLibCaracteristique() {…}
      // Accès en écriture
      public void setCommentaire( String commentaire) {
         _commentaire = commentaire;
      }
      public void setDateDepart( Date dateDepart) { …}
      …
}
```

Concevoir les opérations

La conception des opérations constitue la dernière étape du micro-processus de conception. Le plus gros du travail est certainement fourni lors de cette étape. Il s'agit en effet de donner une image assez précise du contenu des méthodes du projet.

Pour concevoir ou documenter une méthode, UML met à notre disposition toute la palette des diagrammes du modèle dynamique.

- Un diagramme d'activité permet de décrire une méthode pour laquelle peu de classes différentes interviennent. Ce diagramme se révèle efficace lorsque la méthode met en jeu un algorithme avec des étapes et de nombreuses alternatives.
- Un diagramme d'interactions expose au contraire une méthode faisant intervenir plusieurs classes, mais avec peu d'alternatives.

Certaines méthodes sont conçues avec des états stables de calcul intermédiaire et nécessitent un contexte de variables pour mémoriser leur avancement. Ces méthodes peuvent correspondre à une classe de conception avec des états/transitions. Cette technique de transformation d'une méthode d'analyse en classe est depuis longtemps connue des concepteurs objet ; cela s'appelle la réification.


Définition

RÉIFICATION

Réifier consiste à traiter en objet ce qui n'est pas usuellement considéré comme un objet [UML-RM 04] et [Rumbaugh 91].

Cette technique est particulièrement utile pour transformer des comportements dynamiques, tels que processus, tâches, activités, en classes que l'on peut stocker et associer avec d'autres éléments du modèle. La réification de méthodes permet de réaliser des délégations lorsqu'une classe concentre trop de rôles (syndrome de la classe obèse) ; c'est un principe de conception couramment employé pour apporter plus de souplesse et d'évolution à la solution mise en œuvre.

Dans la mesure où la classe issue d'une réification de méthode décrit un processus, un diagramme d'états peut être utilisé pour décrire les états qu'elle réalise. Ce diagramme montrant des successions d'actions et d'activités s'apparente à un diagramme d'activité.

En pratique, la description des méthodes permet d'identifier de nouvelles responsabilités techniques à assigner aux différentes classes du système. C'est ainsi qu'apparaissent tantôt de nouvelles opérations sur les classes existantes, tantôt de nouvelles classes techniques provenant de l'application de design

patterns. Les nouveaux éléments de modélisation alimentent alors une nouvelle itération sur le micro-processus de construction jusqu'à l'obtention d'un modèle facile à coder. Ce processus suit le principe d'une approche *top-down* (du plus abstrait vers le plus détaillé), mise en place à l'aide des techniques traditionnelles de décomposition fonctionnelle.


Définition

LE DESIGN PATTERN STRATÉGIE

Le *design patten* Stratégie [Gamma 95] consiste à encapsuler une famille d'algorithmes qui s'exécutent dans un contexte identique. La stratégie utilise la réification pour transformer les algorithmes en classes, puis l'héritage d'interface pour organiser ces classes en arbre de spécialisation.

Le diagramme de la figure 11-18 indique le schéma nominal d'une stratégie

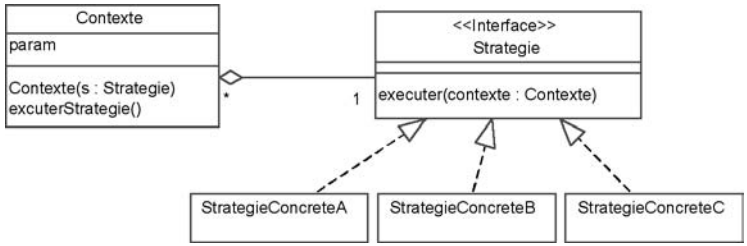

Figure 11-18 : Structure du design pattern Stratégie.

Le contexte d'une stratégie sert à la fois de réceptacle à ses paramètres d'entrée/sortie, et de déclencheur de son comportement. Le mode d'utilisation d'une stratégie est illustré par le diagramme de séquence de la figure 11-19.

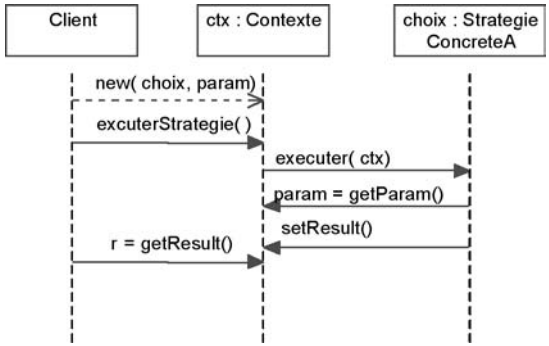

Figure 11-19 : Dynamique de fonctionnement du design pattern Stratégie

On recourt à une stratégie lorsqu'on a besoin de différentes variantes de comportements s'exécutant en fonction de conditions extérieures à une classe. On

utilisera généralement une stratégie pour développer des variantes dépendantes d'un paramétrage ou d'un choix utilisateur. Une stratégie sert également à organiser les différents comportements d'une hiérarchie de classes dotées de la même opération.

ÉTUDE DE CAS : CONCEPTION D'OPÉRATIONS DANS MÉTIER::MISSION

Opération Mission::signalerDepart :

Le signal de départ d'une mission entraîne une chaîne de messages dans le système qui permet la mise à jour des informations d'encours de commande. Un diagramme de séquence est le plus approprié pour documenter ce type de méthode mettant en jeu plusieurs classes différentes.

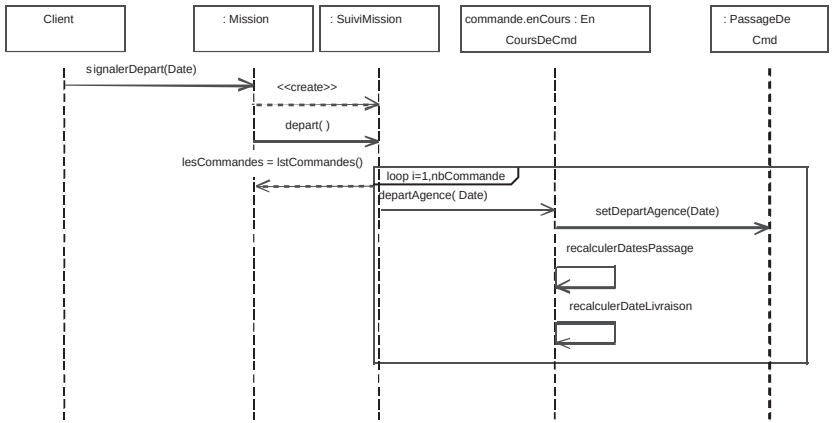

Figure 11-20 : Conception de Mission::signalerDepart au travers d'un diagramme de séquence

Cette séquence de messages correspond aux méthodes décrites ci-après pour la classe *SuiviMission* et l'opération *EnCoursDeCmd ::DepartAgence.*

```
package   SIVEx.metier.mission;
…
public class SuiviMission {
      …
      public SuiviMission( Mission mission, …) {
            _mission = mission;     // définit le lien avec la mission
            …
      }
      public void depart() {
            int    i = 0;
```

```
            Vector       lesCmd = _mission.lstCommandes();
            Commande             cmd;
            EnCoursCommande      ecmd;
            for( i = 0; i < lesCmd.size(); i++ ) {
               cmd = (Commande)lesCmd.elementAt( i);
               ecmd = cmd.encours();
               ecmd.setDepartAgence( _mission.getDateDepart());
            }
      }
      …
}
```

```
package SIVEx.metier.commande;
…
public class EnCoursCommande {
      …
      public void setDepartAgence(Date date) {
         _departAgence = date;
         recalculerDatesPassages();
         recalculerDatesLivraison();
      }
}
```

Opération MissionDeTournee::ordonnerEtapes :

La fonction Ordonner les étapes offre au répartiteur un calcul d'agencement des étapes sur un parcours en fonction de critères de rapidité d'exécution, de distance minimale ou d'urgences à traiter. C'est une méthode construite sur un algorithme de recherche ; elle comporte des étapes d'avancement et des alternatives de calcul dépendant d'urgences à positionner sur le parcours. Cette méthode ne fait appel à aucune autre opération d'autres classes et sa description fait apparaître de nouvelles opérations privées de gestion interne. Un diagramme d'activité est dans ce cas le plus approprié pour représenter cette méthode.

Par ailleurs, le calcul d'ordonnancement réclame un environnement composé de différents tableaux de classement et de diverses opérations spécifiques. C'est pourquoi nous avons choisi de réifier l'opération en une classe *OrdonnateurDEtapes* qui a pour attributs les variables du contexte de calcul et pour opérations les activités de l'algorithme. Les diagrammes ci-après documentent la structure statique et fonctionnelle de cette classe.

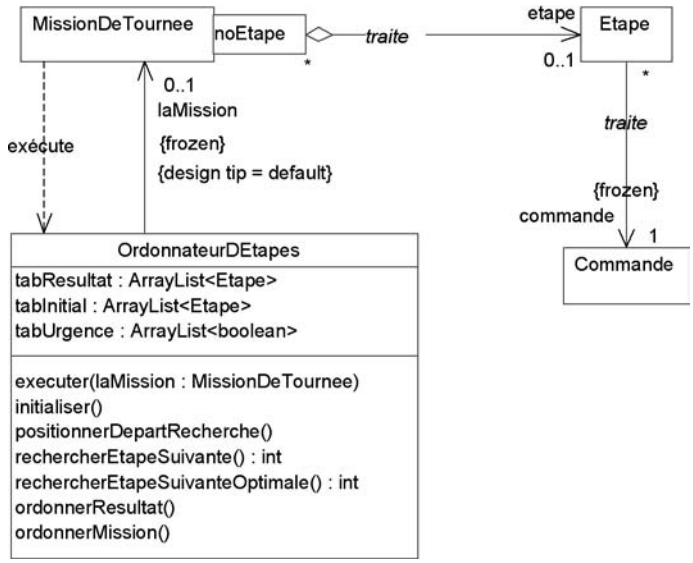

Figure 11-21 : Structure de l'opération réifiée OrdonnateurDEtapes

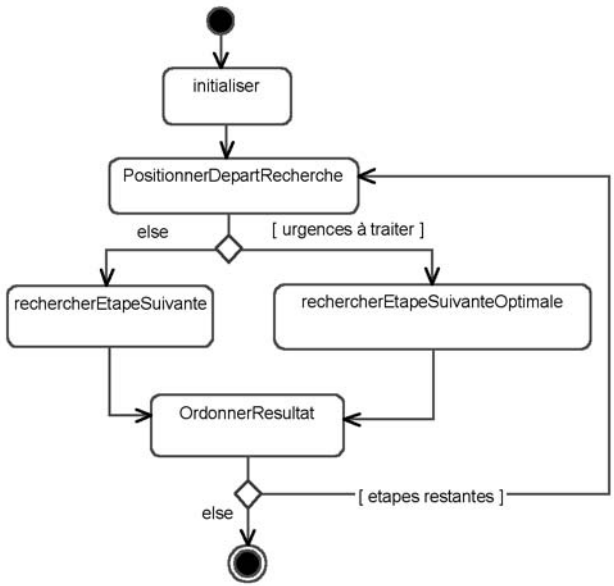

Figure 11-22 : Activités de l'opération OrdonnateurDEtapes::executer()

Le diagramme d'activité correspond au code Java suivant pour la classe *OrdonnateurDEtapes*

```
package  SIVEx.metier.mission;
…
class OrdonnateurDEtapes {
      …
      public OrdonnateurDEtapes( Mission laMission) {
         int          i;
         _laMission = laMission;
         _tabInitial = laMission.lstEtapes();
         _tabUrgence = new Boolean[_tabInitial.size()];
         for ( i = 0; i < _tabInitial.size(); i++ ) {
            Etape etape = (Etape)_tabInitial.elementAt(i);
            _tabUrgence[i] = etape.commande.estUrgente();
         }
      }
      public void executer() {
         int err = 1;
         _initialiser();
         // tant qu'il reste des étapes à traiter :
         while( _tabResultat.size() < _tabInitial.size() && err > 0) {
            _positionnerDepartRecherche();
            if ( _tabUrgence.size() > 0 ) // reste des urgences
               err = _rechercherEtapeSuivanteOptimale();
            else
               err = _rechercherEtapeSuivante();
            _ordonnerResultat();
         }
      }
      private void _initialiser() {…}
      private void _positionnerDepartRecherche() {…}
      private int _rechercherEtapeSuivanteOptimale() {…}
      private int _rechercherEtapeSuivante() {…}
      private void _ordonnerResultat() {…}
}
```

Opération SuiviMission::recevoirEvenement :

Notre dernier exemple concerne la réception d'un événement de mission dont le traitement revient au suivi de mission en cours. La réception d'un événement s'exécute toujours dans un même contexte, mais donne lieu à des comportements totalement différents suivant le type d'événement.

Nous avons en effet distingué en analyse, les simples événements des incidents d'étape et de trajet. Suivant leur nature, les incidents engendrent une série de messages susceptibles de provoquer une réponse du répartiteur ou de recalculer les encours de mission.

En raison de cette divergence de comportements, nous avons choisi de mettre en œuvre une stratégie dont la hiérarchie est documentée dans le diagramme de la figure 11-23.

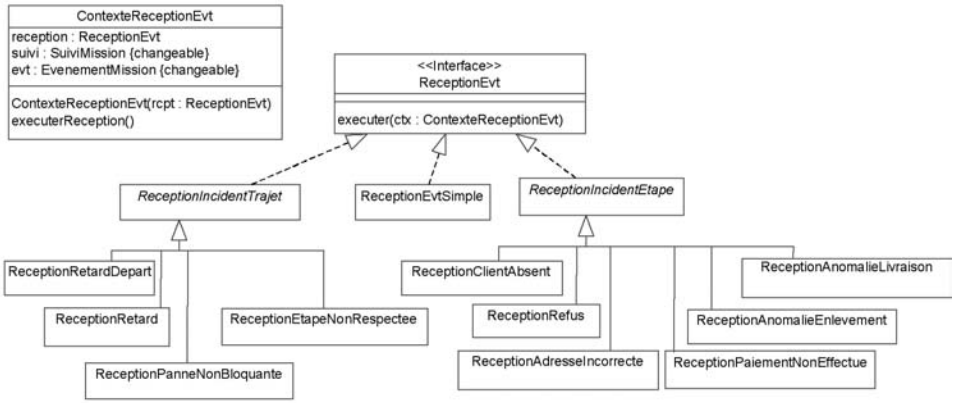

Figure 11-23 : Structure des stratégies de réception d'un événement

La fabrication de la stratégie dépend du type de l'événement de mission reçu par le système. Pour cela, une fabrique cataloguée isole et encapsule l'aiguillage switch que nécessite la construction des différents types de stratégies. Le diagramme de séquence ci-dessous illustre la réception d'un refus client lors d'une étape.

Une fois les mécanismes de la stratégie définis, il convient bien entendu de décrire la méthode *executer(ctx)* de tous les types de stratégies possibles. On aura recours pour cela aux diagrammes d'interactions et d'activités.

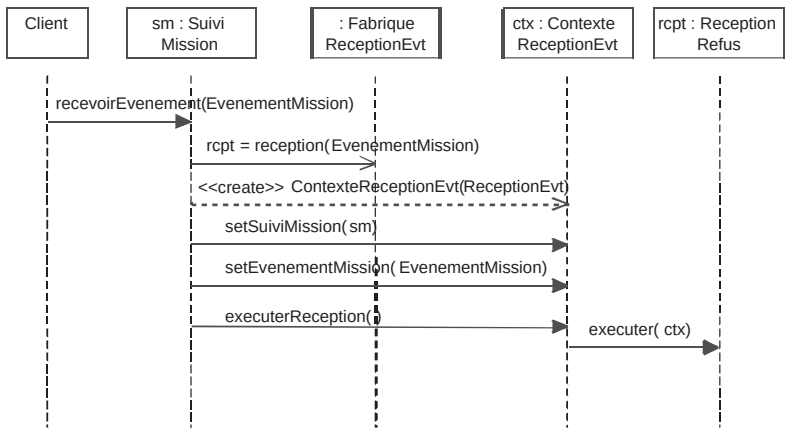

Figure 11-24 : Dynamique d'exécution sur la réception d'un événement d'étape

Nous vous présentons ci-après la structure du code correspondant à la fabrique cataloguée chargée de construire les stratégies.

```
package SIVEx.metier.mission.evenementsMission;
…
public class FabriqueReceptionEvt()
      // mise en œuvre du catalogue :
      Map      _catalogue;
      ReceptionEvt      instance( Class cl) {
         // la classe de l'événement sert de clé :
         Object recept = _catalogue.get( cl);
         if ( recept == null ) {
            // swith de fabrication en fonction des types d'événements :
            if ( cl.getName().equals( «IncidentTrajetRetardDepart») )
               recept = new ReceptionRetardDepart();
            else if ( cl.getName().equals(«IncidentTrajetRetard») )
               recept = new ReceptionRetard();
            else if ( cl.getName().equals(«IncidentTrajetPanneBloquante»))
               recept = new ReceptionPanneBloquante();
            //etc…
            _catalogue.put( cl, recept)
         }
         return (ReceptionEvt)recept;
      }
      …
      public void reception( EvenementMission evt, SuiviMission sm) {
         ReceptionEvt         strategie = instance( evt.getClass());
         ContexteReceptionEvt ctx = new ContexteReceptionEvt( strategie);
         ctx.setSuiviMission( sm);
         ctx.setEvenementMission( evt);
         ctx.executerReception();
      }
}
```

Conception de la couche de présentation

La couche de présentation ou IHM se limite à la partie visible d'une application. Les environnements avec fenêtrage (type Windows) ou pages HTML équipées de mécanismes réflexes en JavaScript, correspondent aux choix technologiques les plus courants. Dans ces environnements, un utilisateur est face à trois grandes familles de concepts.

- les fenêtres (ou pages) et leur contenu qu'il voit, redimensionne, bouge et réduit, font partie des concepts visuels ;

- les actions qu'il peut déclencher et les changements d'aspects, font partie du comportement de la présentation ;
- les flux de données qu'il transmet *via* des listes de choix, des champs d'édition font partie de l'échange d'informations avec l'application.

Concevoir ou documenter une couche de présentation revient à passer en revue ces trois aspects : le visuel, le comportemental et le fonctionnel.

Grâce aux environnements de développement actuels, la conception d'une IHM s'effectue le plus souvent conjointement avec sa construction visuelle. Cette capacité est liée à l'existence d'éléments standard qui peuvent se composer à souhait pour former l'application de notre choix. Une IHM se caractérise également par la notion de fenêtre ou de page qui représente un élément insécable de présentation. Cette notion présente en effet une unité visuelle, comportementale et fonctionnelle, caractéristique très utile pour organiser le modèle UML de conception. Par ailleurs, une fenêtre ou une page correspond à une vue de la couche applicative, comme nous l'aborderons au paragraphe suivant.

Il serait en fait rarement utile de documenter avec UML la structure statique d'une IHM, si ce n'est pour en étudier le comportement et les informations échangées. C'est en effet à la fois la force et la faiblesse des environnements modernes que de permettre la construction immédiate de l'IHM statique : force pour la productivité que cela procure au développement, faiblesse pour l'apparente facilité qui masque les contraintes d'architecture d'une conception. On constate ainsi que les développeurs entreprennent la construction immédiate des comportements et des flux de données, sans avoir pris la peine de dénouer les mécanismes de comportements et d'échanges nécessaires. Le manque de modélisation et de réflexion sur l'intégrité conceptuelle engendre fréquemment un imbroglio de code difficile à relire et coûteux à maintenir. Le manque de clarté est généralement lié à l'amalgame de responsabilités de niveau comportements, fonctionnels et applicatifs, au sein de la même classe d'IHM. Nous allons justement étudier ici comment séparer ces responsabilités pour gagner en lisibilité et en facilité de maintenance.

La première étape de conception d'une IHM concerne la définition visuelle des fenêtres ou des pages. L'existence d'un modèle objet d'analyse permet d'influencer cette conception : à partir d'un diagramme de classes, un générateur de code pourrait générer des fenêtres ou des pages pour l'affichage et la saisie de chaque élément du modèle :

- une fenêtre ou plusieurs pages pour chaque classe afin d'en éditer les instances : création, modification des attributs et des relations, simple consultation et suppression ;
- une fenêtre ou plusieurs pages pour certaines associations complexes afin d'en éditer les liens.

La figure 11-25 vous donne un aperçu de cette idée en montrant des fenêtres d'édition des sites et des parcours du réseau. L'équivalent en pages HTML est

aussi facile à imaginer ; notons cependant qu'il faut parfois doubler les pages car l'affichage et la saisie procèdent de techniques HTML différentes.

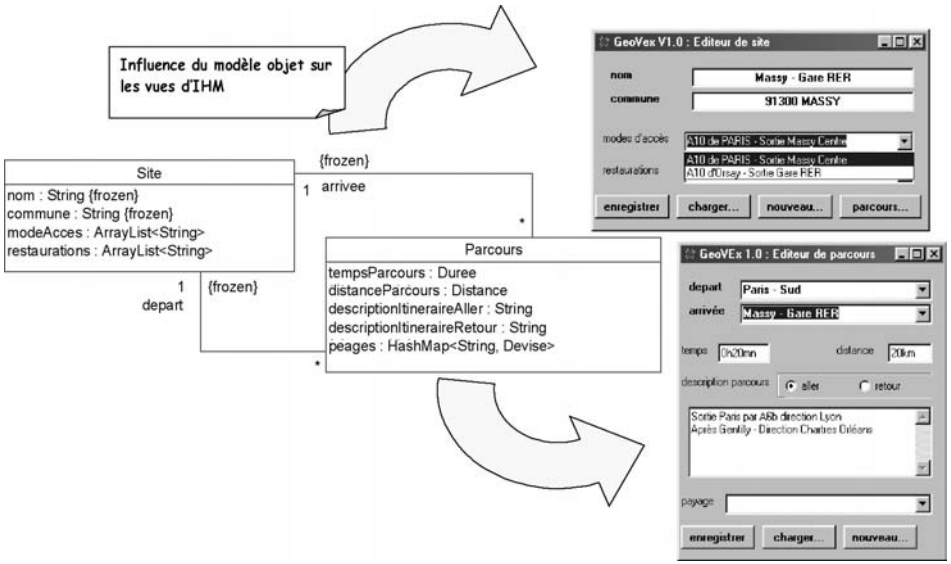

Figure 11-25 : Développement d'une IHM calquée sur le modèle objet

La conception d'une IHM provenant d'un modèle objet n'est pourtant pas toujours aussi simple. Comme l'illustre le schéma de la figure 11-26, l'IHM doit en effet tenir compte des synoptiques attendus par les utilisateurs, or ces derniers font rarement partie du modèle objet d'analyse.

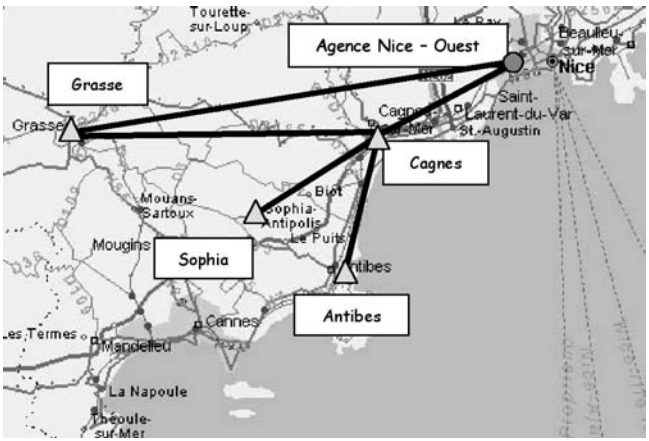

Figure 11-26 : Exemple de synoptique attendu mais non représenté dans le modèle objet d'analyse

L'utilisation d'UML pour documenter la structure statique d'un IHM permet de répertorier les fenêtres ou pages puis d'identifier, le cas échéant, les composants réutilisés d'une vue à l'autre. Dans le diagramme de la figure 11-27, nous avons représenté les fenêtres de gestion du réseau ainsi qu'une liste déroulante ou *drop list* présentant une sélection de choix de sites, utilisée à la fois dans la fenêtre d'édition des parcours et dans la fenêtre de sélection des parcours.

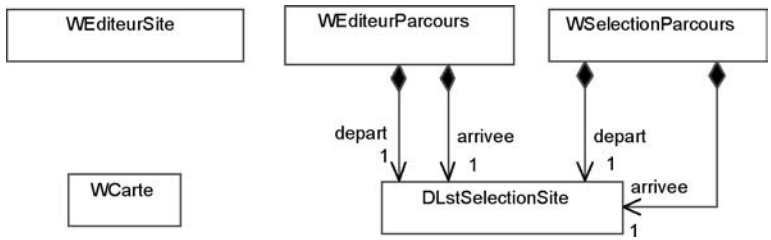

Figure 11-27 : Structure des classes graphiques d'édition du réseau

La seconde étape de conception consiste à définir les comportements des fenêtres. Une modélisation UML s'impose alors pour concevoir la dynamique des IHM. En effet, à partir d'une fenêtre ou d'une page, un utilisateur déclenche des actions ou contrôle des activités. À chaque instant, la vue doit refléter ce qu'il a le droit ou non de faire, en désactivant des boutons, des champs, des menus ou tout autre élément visuel de contrôle.

La plupart des vues ont en fait un comportement de machine à états ; le développement d'un diagramme d'états est alors très utile à la conception de leur comportement. D'autant plus que certains frameworks applicatifs, tels que le composant *struts* du projet Apache, réalisent explicitement le déroulement d'une machine à états en associant une page et une classe Java à chaque état et en décrivant la dynamique des transitions séparément dans un fichier XML.

L'étude de l'éditeur de site permet d'identifier trois états distincts ainsi que de trouver les attributs, les événements et les actions associés à la fenêtre ou à la page correspondante.

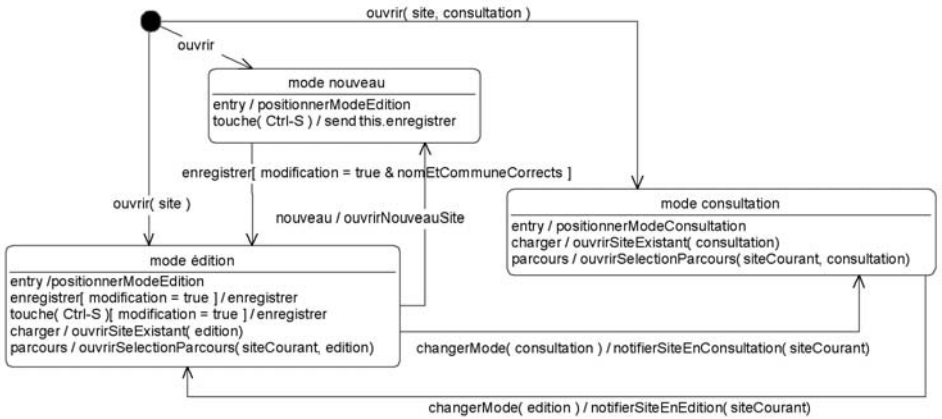

Figure 11-28 : États de la fenêtre Éditeur de site

En conception orientée objet, il est d'usage d'associer à chaque vue une classe dite contrôleur, dont l'objectif est de piloter le comportement interactif avec l'utilisateur. Tandis que les événements transmis par l'utilisateur sont reçus et traduits techniquement par la fenêtre, les activités et les états concernent majoritairement le contrôleur. Si, en outre, le contrôleur est conçu à l'aide du *design pattern* état, le code lié à l'IHM devient extrêmement souple et facile à maintenir et ce même pour les fenêtres aux comportements les plus complexes. Le diagramme de la figure 11-29 montre la structure de la classe contrôleur ainsi que les états qui en résultent.

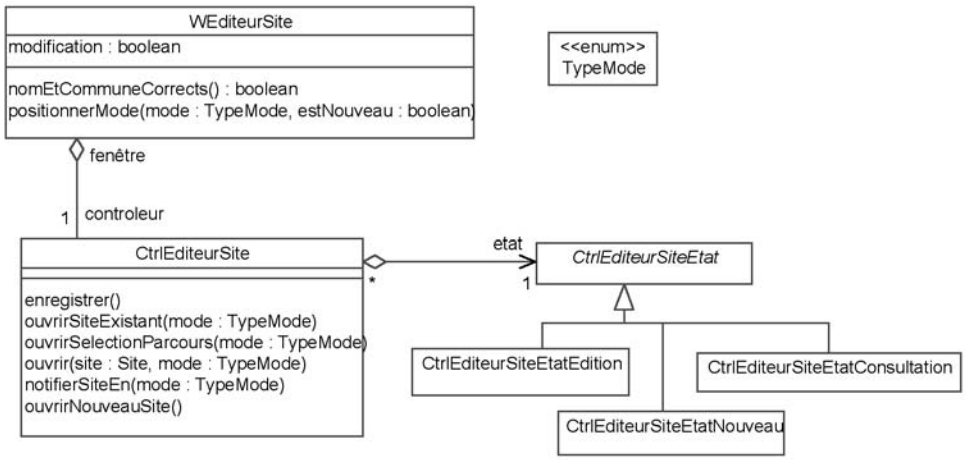

Figure 11-29 : Structure Fenêtre - Contrôleur – États, de la fenêtre Éditeur de site

Le contrôleur prend en charge les activités impliquant une coordination avec la couche de l'application, tandis que la vue reste responsable des opérations gérant l'aspect graphique.

La conception de l'IHM se termine enfin par l'étude des flux de données. Dans le modèle vue-contrôleur que nous développons ici, la vue communique avec l'image mémoire des données qu'elle affiche. Lors de la conception générique, nous avons qualifié de document (voir chapitre 9) cette image qui, tout en appartenant à la couche applicative, fait le lien entre le contenu affiché d'une vue et les objets métier.

Si la couche application est responsable des documents, les vues de présentation sont généralement responsables du transfert de format entre les informations du document et celles qui sont affichées. Des opérations d'accès sur toutes les données modifiables prennent en charge les transformations nécessaires. Ces opérations permettront au contrôleur de modifier la présentation ou de récupérer les informations introduites par l'utilisateur. Le diagramme de la figure 11-30 vous donne en final la structure de la vue *WEditeurSite*.

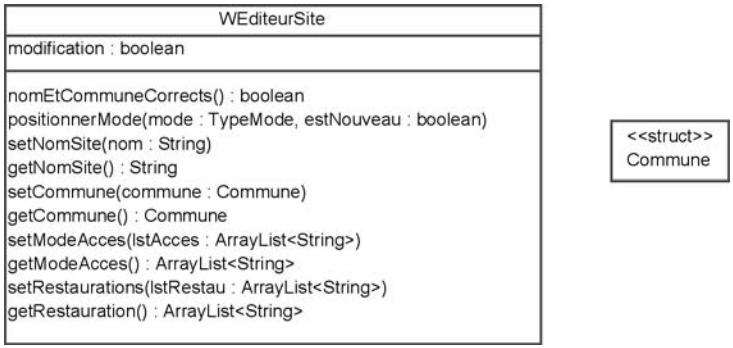

Figure 11-30 : Attributs et opérations de la fenêtre Éditeur de site

Nous n'enchaînerons pas le micro-processus de conception sur l'étape concernant les opérations, car dans ce cas, la formalisation des opérations de transfert de format est superflue. Mais, pour bien en comprendre l'importance, imaginez les opérations que nécessiterait la vue représentant le réseau sous forme cartographique (voir figure 11-26) : les coordonnées géographiques de chaque site doivent en effet être converties en coordonnées image en fonction du niveau de zoom et de centrage choisi par l'utilisateur.

ÉTUDE DE CAS : CONCEPTION DE LA VUE D'ÉDITION D'UNE MISSION

La fenêtre d'édition et de consultation des missions est un peu plus sophistiquée. Elle correspond à la classe *WEditeurMission*.

- La partie supérieure comprend les attributs qui définissent une mission. Ces derniers sont obligatoirement renseignés lors de la création d'une nouvelle mission.
- La partie intermédiaire représente à droite les attributs de la mission, à gauche les relations avec les ressources chauffeur et véhicule.
- La partie inférieure donne les informations de chargement de la mission, le bouton Commandes permet d'affecter des commandes via une fenêtre complémentaire.
- Une barre d'état restitue enfin l'état de l'objet métier contenu dans la fenêtre.

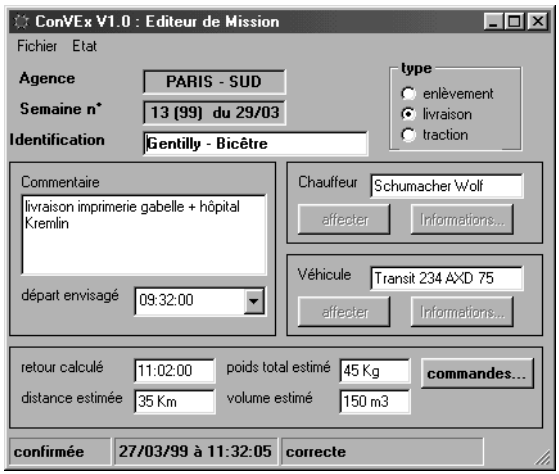

Figure 11-31 : Fenêtre d'édition d'une mission, telle que présentée dans la maquette d'IHM

Le comportement de cette fenêtre est sensiblement identique à la fenêtre d'édition d'un site. Le menu *Fichier* permet d'accéder aux activités de niveau application : enregistrer, charger et création d'une nouvelle mission, tandis que le menu *État* accède aux activités de niveau métier : valider, annuler et recalculer pour forcer la mise à jour des attributs de chargement. L'IHM doit restituer l'état métier de l'objet représenté, ce qui influe sur les états applicatifs de la fenêtre WEditeurMission.

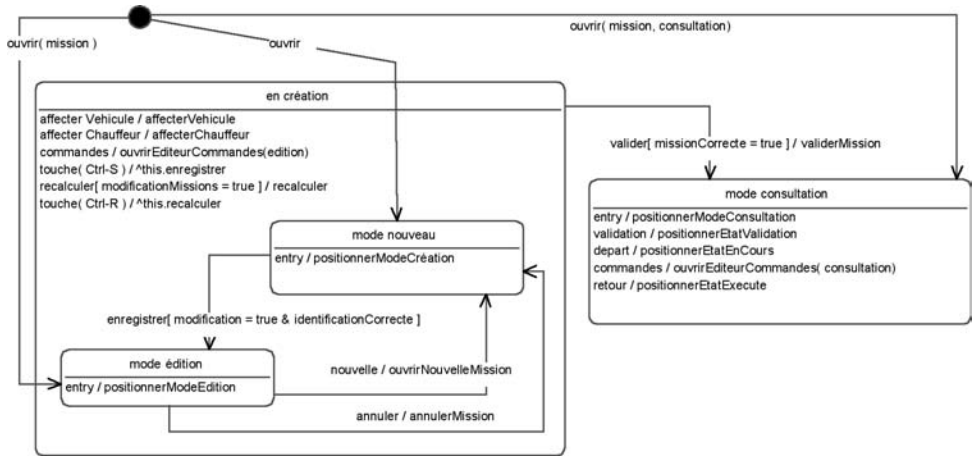

Figure 11-32 : États de la fenêtre d'édition d'une mission

La distribution des activités sur les classes *Vue* et *Contrôleur* ainsi que les opérations de transfert des données aboutissent au diagramme de classes de la figure 11-33. Les contrôles IHM correspondant aux associations et aux attributs dérivés du modèle d'analyse sont introduits dans le document par des moyens complémentaires, il n'y a donc pour ces attributs que des opérations *set*, permettant au contrôleur de positionner la valeur d'affichage.

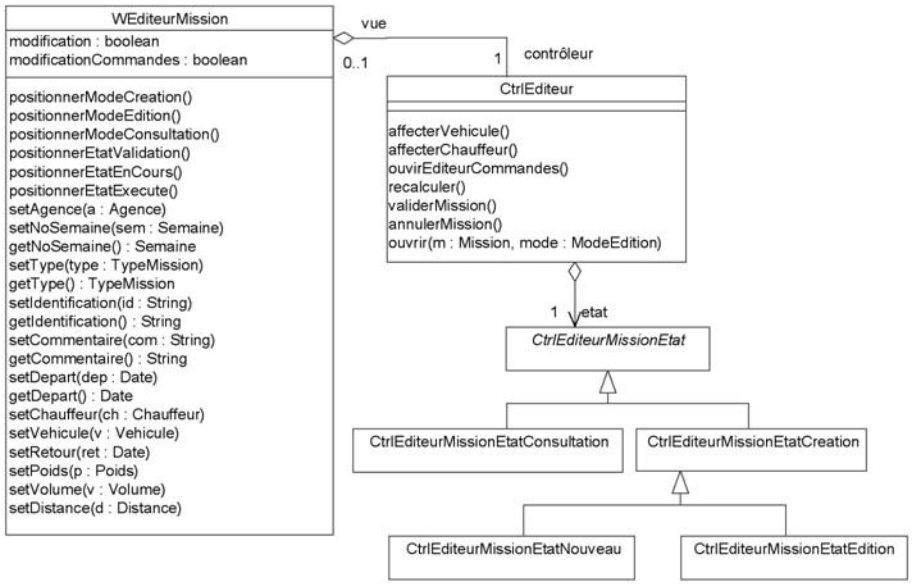

Figure 11-33 : Structure Vue-Contrôleur-États de la fenêtre Éditeur de mission

LE DESIGN PATTERN OBSERVATEUR

Le *design pattern* Observateur [Gamma 95] consiste à synchroniser des objets en minimisant les dépendances qui devraient s'établir entre eux. Chaque objet n'a cependant pas un rôle symétrique : nous y distinguons le sujet et les observateurs.

Le sujet centralise les données et il est unique. Il comprend des opérations permettant aux observateurs d'accéder à ses données.

L'observateur restitue les données du sujet auquel il est abonné, et plusieurs peuvent se synchroniser sur le même sujet.

Le diagramme de la figure 11-34 représente la structure nominale de l'observateur.

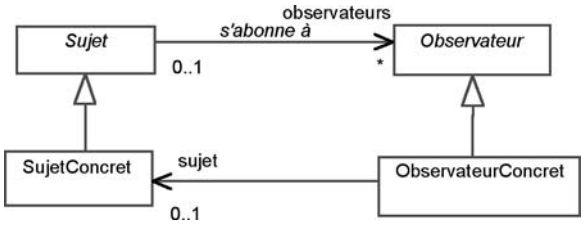

Figure 11-34 : Structure du design pattern Observateur

La dynamique d'échange entre le sujet et ses observateurs abonnés s'établit à partir d'une notification indiquant une modification du sujet. Ce dernier en avise ses observateurs qui questionnent en retour le sujet pour obtenir les informations nécessaires à leur mise à jour. Le diagramme de communication de la figure 11-35 décrit une notification concernant un sujet et ses deux observateurs.

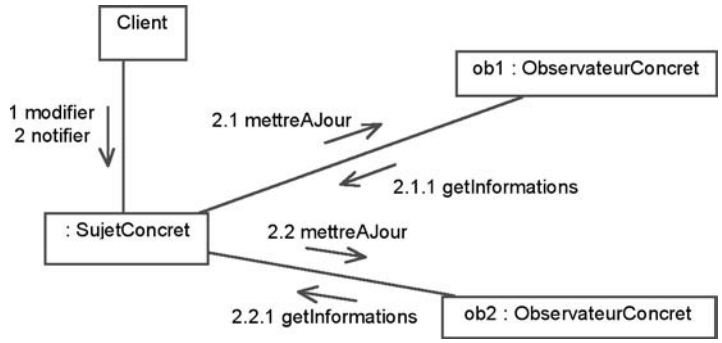

Figure 11-35 : Dynamique du design pattern Observateur

L'observateur est également connu comme modèle document-vue : terminologie que nous avons justement retenue lors de la phase de conception générique de la couche de l'application.

Dans ce cadre, chaque fenêtre ou page de présentation implémente une vue (observateur) et s'abonne par ce biais aux modifications du document associé. De leur côté, les documents ne connaissent des fenêtres ou des pages que leur interface *Vue*. Par cet artifice, nous conservons l'indépendance logicielle de la couche de l'application vis-à-vis de la couche de présentation, bien que les mécanismes de rafraîchissement leur imposent une collaboration étroite.

Ce *design pattern* est intégré à Java au travers des classes *Observable* et *Observer* du package *Java.util*. Par souci de standardisation, la conception du *framework* de la couche de l'application n'a pas manqué d'intégrer ces classes pour concevoir les notions de Document et de Vue.

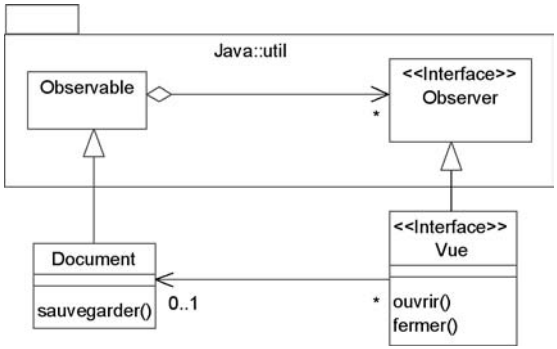

Figure 11-36 : Utilisation des définitions standards de Java

Conception de la couche Application

Le rôle de la couche de l'application consiste à piloter les processus d'interactions entre l'utilisateur et le système. Cette notion peut paraître abstraite, mais il s'agit généralement de mettre en œuvre toutes les règles nécessaires au maintien d'une application cohérente et à l'optimisation des échanges client/serveur et/ou des requêtes http.

De manière plus précise, les mécanismes d'une application assurent :

- l'existence de différentes fenêtres ou pages synchronisées sur les mêmes données ;
- la cohérence entre les objets distribués et les multiples façons de les représenter au travers des IHM ;
- l'optimisation des chargements au niveau des *servlets* pour un déploiement en client léger ou sur le poste client pour un déploiement client/serveur ;

- le respect des habilitations des différents acteurs ;
- l'obligation pour l'utilisateur d'abandonner ou de mener jusqu'au bout un changement commencé ;
- la mise en œuvre des concepts applicatifs : typiquement les sorties de rapports sur imprimante.

Pour toutes ces capacités, la conception d'une application s'appuie sur la définition des documents représentant en fait l'image mémoire des fenêtres ou des pages de présentation. Par définition, il existe un document par vue de l'application. Chaque document définit ensuite les opérations permettant aux pages ou fenêtres d'accéder aux informations nécessaires à l'affichage.

La synchronisation concerne la relation entre un document et ses vues, mais également entre plusieurs documents. Un objet *Site* peut être à la fois visible *via* la page d'édition d'un site, mais également par l'intermédiaire d'une représentation géographique. Le document associé à la carte géographique est qualifié de document composite, en ce sens qu'il rapporte implicitement l'information de plusieurs documents Site et Parcours. En conséquence, le document composite devient implicitement un observateur d'autres documents. Cette caractéristique se traduit par des relations d'agrégation entre documents, comme illustré dans le diagramme de la figure 11-37.

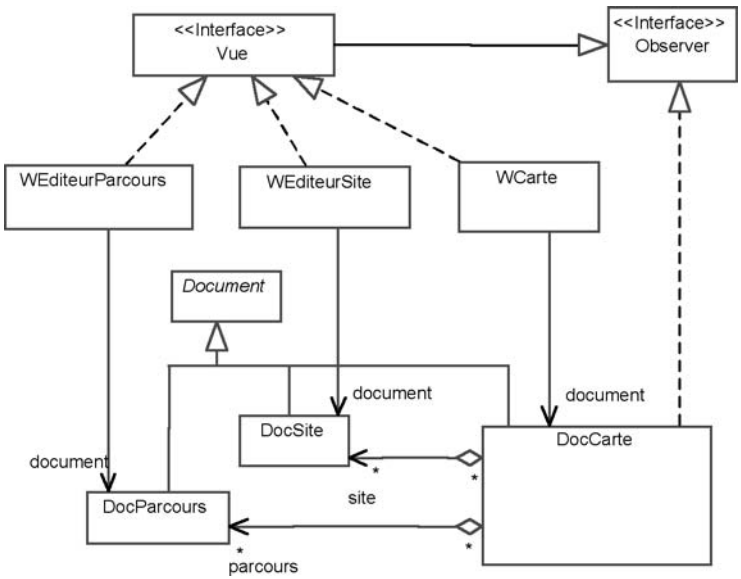

Figure 11-37 : Structure du document composite associé à une carte géographique

Le diagramme de communication de la figure 11-38 montre comment la modification d'un site se propage parallèlement à la fenêtre d'édition et à la carte.

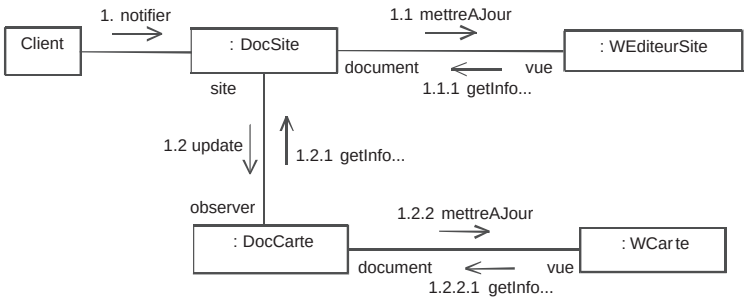

Figure 11-38 : Propagation d'un rafraîchissement entre documents et vues

On trouve généralement des documents composites lorsqu'il faut représenter des cartes, des synoptiques, des schémas, des courbes, des tableaux, des arborescences ou des diagrammes. Une liste d'objets présente également le comportement d'un document composite lorsque celle-ci doit réagir au changement d'un libellé caractéristique ou à la suppression d'un objet.

L'optimisation des échanges consiste à mémoriser ce qui a déjà été chargé, de telle sorte que l'application puisse minimiser les appels aux objets distribués.

Charger un document revient donc à demander le transfert des entités correspondantes ; la mémorisation de ces chargements s'effectue à l'aide d'un catalogue de références OID. À des fins d'optimisation, le catalogue est organisé en sous-catalogues, classés par nom de classe, qui retrouvent eux-mêmes les objets par OID. Comme le montre le diagramme de la figure 11-39, les attributs qualificatifs d'UML servent à spécifier ces clés de rangement.

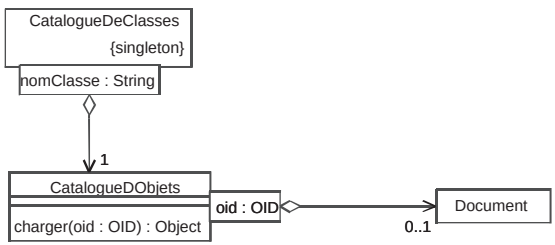

Figure 11-39 : Structure du référencement des documents chargés dans la couche application

Le diagramme précédent schématise la structure du catalogue de référence des documents. L'interface *EntComposant* représente toutes les entités provenant des composants distribués. Le chargement d'une entité est donc systématiquement soumis au catalogue, de manière à retrouver les données précédemment chargées.

Si cette technique minimise les échanges sur le réseau, elle en complique le protocole de synchronisation. Il faut en effet s'assurer par d'autres mécanismes que les données précédemment chargées n'ont pas été modifiées parallèlement par une autre application. Cependant, nous arrêterons là la réflexion de conception car ce n'est pas le but de l'ouvrage ; sachez que ce problème lié à l'intégrité des données distribuées peut être géré de différentes manières avec différents coûts de développement. Nous conseillons dans ces cas de procéder à une analyse de la valeur et de fournir uniquement le mécanisme de synchronisation nécessaire, sans rechercher de sophistications coûteuses.

Nous traitons enfin la cohérence des processus de l'application en recourant au design pattern Commande. Cette technique nous permet en effet de garantir un comportement homogène face aux actions de l'utilisateur, en termes de traces produites et de vérification d'habilitations. Une commande complexe peut enchaîner des sous-commandes et assurer l'atomicité transactionnelle des interactions de l'utilisateur, à savoir que toute interruption d'une sous-commande implique l'interruption de la commande complexe.

Définition

LE DESIGN PATTERN COMMANDE

Le *design pattern* Commande [Gamma 95] consiste à réifier un groupe d'opérations afin de pouvoir les traiter comme les ressources d'une application. On peut mémoriser de la sorte les dernières commandes effectuées par un utilisateur ou bien leur associer dynamiquement une séquence de touches clavier.

La structure du *design pattern* Commande inclut les éléments suivants (voir figure 11-40) :

- une interface d'exécution générique (appelée ici « commande applicative », *CommandeApp* pour la différencier de l'objet métier *Commande*) ;
- un récepteur correspondant à l'objet porteur de l'opération effectivement exécutée par la commande ;
- un invocateur chargé de gérer et d'enchaîner l'exécution des commandes.

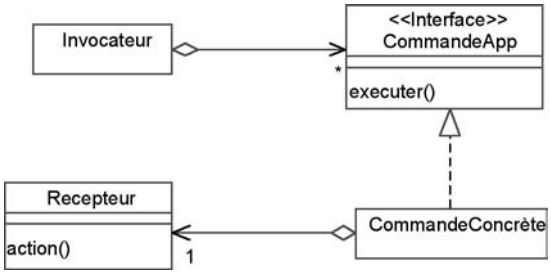

Figure 11-40 : Structure du design pattern Commande

Ce *design pattern* permet de :

- standardiser l'exécution d'une même famille d'opérations, en produisant des vérifications d'habilitation et des traces d'exécution ;
- contrôler le déroulement et l'atomicité d'une action composée d'un enchaînement de sous-actions.

Nous illustrons ces deux avantages dans la conception des commandes d'édition d'une mission. Mais ce *design pattern* permet aussi de :

- différer l'exécution d'actions lorsque l'invocateur doit assurer, par exemple, la disponibilité d'un équipement tout en maintenant la disponibilité de l'interface utilisateur ;
- mémoriser les commandes exécutées et d'offrir le cas échéant le service de défaire (*undo*) ou de reprise de crash ;
- associer dynamiquement des actions à des contrôles de l'utilisateur, pour réaliser des macros ou bien pour personnaliser des touches du clavier.

Dans notre conception, l'interface *CommandeApp* est potentiellement composée d'autres commandes. Les commandes représentant par ailleurs les actions de l'utilisateur sur une application, le récepteur est naturellement le document sous-jacent à la vue d'où est déclenchée l'action. L'invocateur coordonne enfin les commandes, les documents et les informations de l'utilisateur connecté pour générer les traces et vérifier les habilitations.

Le diagramme de communication de la figure 11-41 résume les rôles et responsabilités des différents objets intervenant dans l'exécution d'une action de l'utilisateur.

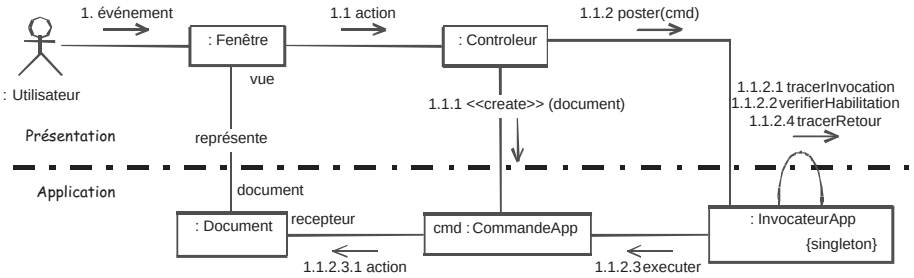

Figure 11-41 : Dynamique d'échange d'une commande entre les couches présentation et application

Dans la couche de présentation :

- la fenêtre ou la page transforme les événements de l'utilisateur en action, elle restitue par ailleurs les informations du document si ce dernier a été modifié ;
- le contrôleur centralise les actions déclenchées depuis l'IHM, il crée la commande correspondante à l'action et la place en file d'exécution auprès de l'invocateur applicatif ;

Dans la couche de l'application :

- l'invocateur exécute les commandes et, suivant un protocole standard, assure les traces et les vérifications d'habilitation ;
- la commande encapsule l'action sur un document et permet de les enchaîner dans une même unicité d'actions de l'utilisateur ;
- le document encapsule les données nécessaires à la présentation d'une fenêtre ou d'une page. Il gère la cohérence des différentes vues sur ses données et prend en charge les opérations nécessaires à l'exécution des commandes.

On retrouvera le concept de commande au travers de la classe Action du framework Struts.

ÉTUDE DE CAS : CONCEPTION DU DOCUMENT D'ÉDITION DE MISSION

La conception des documents est, en termes d'attributs et d'opérations, le reflet des vues de présentation.

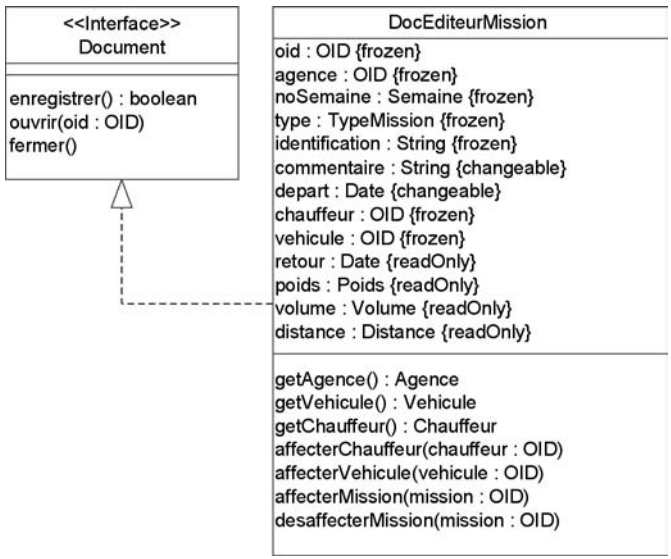

Figure 11-42 : Définition du document d'édition de mission

La conception des classes de type document doit prendre en compte les états applicatifs, ce qui permet d'affiner ses opérations par l'application du *design pattern* État. On se servira donc ici de la technique déjà présentée dans le paragraphe consacré à la conception des classes.

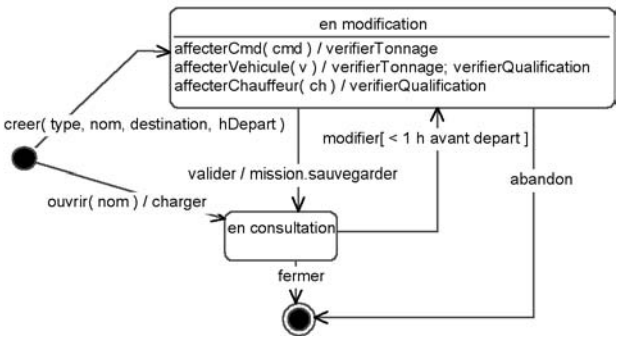

Figure 11-43 : Conception des états du document d'édition de mission

Il convient dans un second temps de concevoir les commandes associées au document. Il n'y aurait bien entendu aucun intérêt à traiter toutes les actions possibles de l'utilisateur, c'est pourquoi nous prenons, à titre d'exemple, la commande applicative d'affectation d'une Commande métier à une mission[1] . Le processus consiste à ouvrir une vue permettant l'édition des objets Commandes de la mission, puis à itérer sur une ou plusieurs sélections de l'utilisateur, avant de terminer par une validation ou un abandon. La commande applicative pilote donc elle-même d'autres sous-commandes qui peuvent interrompre la transaction en cours. Ce type d'enchaînement illustre parfaitement ce que nous appelons un processus applicatif.

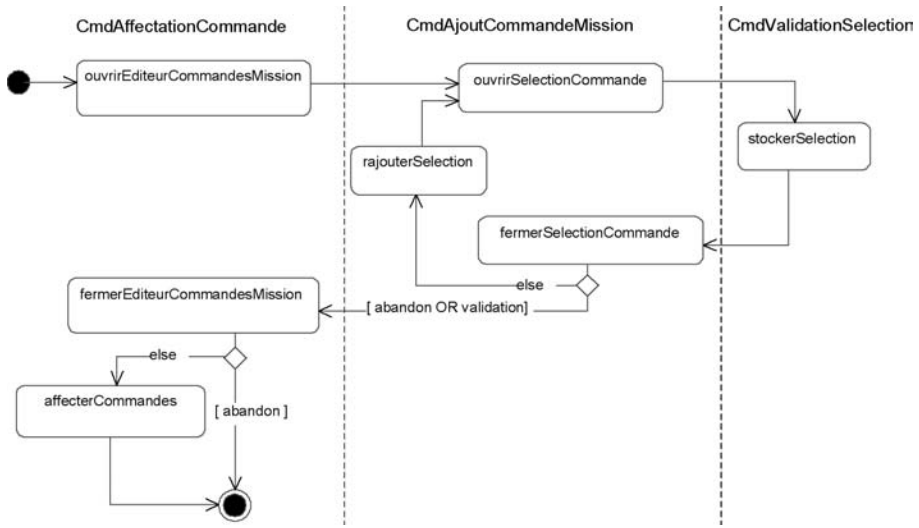

Figure 11-44 : Processus d'affectation d'objets Commande à la Mission

Un diagramme d'activité doté de couloirs de responsabilité (ou partitions) constitue la formalisation la plus appropriée pour représenter ce type d'enchaînement. L'aperçu que vous en donne la figure 11-44 montre la séparation des responsabilités et l'imbrication des déclenchements entre les commandes applicatives.

Conception de la couche métier distribuée

Maintenant que vous avez un bon aperçu de la conception des couches de présentation et de l'application, nous allons aborder l'étude de la couche métier. Conformément aux préconisations du style d'architecture 3-tiers, la couche métier distribue ses interfaces *via* le middleware RMI.

1.Il se trouve que nous appliquons le Design Pattern commande aux concepts métier de Missions et de *Commande*. Pour ne pas risquer de confondre les deux notions une Commande commençant par une majuscule fait référence au concept métier.

Une conception simpliste de la distribution consiste à projeter toutes les classes de la couche métier en interfaces EJB. Cette approche naïve a cependant les inconvénients suivants :

- toute consultation ou modification sur les attributs d'une classe métier entraîne un échange de services RMI/IIOP (protocole utilisé par les EJB). On imagine le volume de transactions nécessaires à l'édition d'une mission comportant une dizaine d'attributs et les temps de réponse qui en découlent ;
- toutes les instances en cours d'édition dans l'application donnent chacune lieu à une tâche et à une connexion d'échanges RMI/IIOP. Pour un système de l'ampleur de SIVEx, cela exige des ressources CPU, mémoire et socket très importantes de la part des serveurs ;
- toutes les instances distribuées ont une adresse IIOP et saturent le gestionnaire de références.

Pour éviter ces handicaps également valables pour CORBA, DCOM ou RMI, il est d'usage de concevoir une distribution en tenant compte des principes suivants. Bien entendu, ces principes ne sont pas à prendre en dépit du bon sens.

- une classe représente un ensemble d'attributs insécables pour la distribution. Dans ce cas, il vaut mieux échanger en une seule transaction l'ensemble des informations d'une instance ;
- lorsque l'exploitation d'une classe ne concerne que des aspects CRUD (Create, Retrieve, Update & Delete) et que la fréquence d'interactions avec le serveur est faible (inférieure à 20 transactions par minute), il est d'usage de ne définir qu'un seul objet distribué par classe. Cet objet, de type EJB session et non EJB entity, représente successivement différentes instances dont les états sont conservés au travers des structures d'informations échangées.

Ces principes respectent les contraintes qu'impose un middleware objet : réduire la fréquence des transactions, le nombre de références et la multiplication des tâches sur le serveur. Pour appliquer ces conseils, nous utilisons les deux types d'objets distribués de la norme EJB.

- Les premiers, de type session, constituent les instances qui échangent les instances par structure de données ; nous en avons conçu le concept au travers d'une interface capable de fournir un OID et un ensemble de clés métier, nécessaires à l'identification par les utilisateurs.

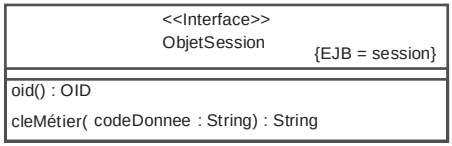

Figure 11-45 : Définition de l'interface entité de composant dans la conception générique

- Les seconds, de type entity, correspondent aux objets que l'on désire réellement représenter avec un état distribué. Chaque objet session, échangé par valeur, peut être associé à l'entité qui correspond au même objet métier distribué et qui peut être utilisé conjointement suivant les deux modes, dans le cas où on attend une forte fréquence de consultation sur une même instance de cette classe. Le diagramme ci-après montre la structure d'une entité d'une session couplée. On recourt à une technique de représentation UML souvent utilisée en conception détaillée, mais qui n'est pas complètement conforme à UML : nous savons en effet qu'une interface ne peut comporter ni attribut, ni association. Les attributs et les associations de l'interface représentent en fait les opérations d'accès (*get* et *set*) qu'ils génèrent par application des règles de conception des attributs et des associations que nous avons étudiées dans ce chapitre.

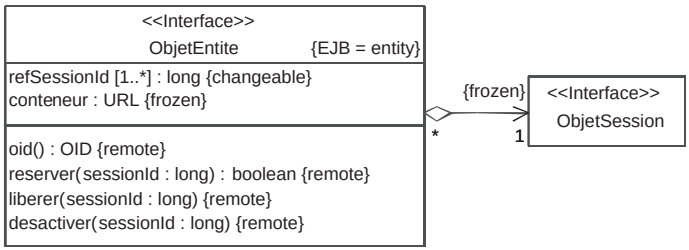

Figure 11-46 : Définition d'une entité couplée à un objet session au niveau de la conception générique

La conception de la couche métier consiste à identifier les objets entités et sessions qu'il convient de développer au vu des classes et des opérations métier à distribuer.

Le concept d'objets distribués pose également le problème de la distribution des liens d'un graphe d'objets. Pour cela deux techniques s'opposent :

- soit les objets sont distribués unitairement, et la demande d'un objet lié ou de sa référence déclenche une nouvelle demande de transaction. Cette pratique convient bien à l'édition d'objets métier telle que celle d'une mission : lorsque l'utilisateur veut obtenir des informations sur le chauffeur associé à la mission en cours d'édition, une nouvelle transaction rapporte l'entité de chauffeur correspondante dans le contexte mémoire de la *servlet* ou de l'application cliente ;

- soit un graphe complet d'objets liés est préalablement chargé dans le contexte mémoire de la couche de présentation. Cette technique s'utilise de préférence pour réaliser la présentation de documents composites, à savoir l'édition d'une carte du réseau qui requiert le rapatriement des informations de tous les sites et parcours à afficher. La difficulté consiste cependant à définir la frontière du graphe, de sorte qu'il n'est peut être pas nécessaire de rapporter tous les sites et parcours d'Europe, si la carte ne concerne que la région Côte d'Azur.

Pour répondre à cette problématique de conception, nous avons introduit le *design pattern* Référence futée, inspiré du *smart pointer* : l'idiome de programmation C++ [Lee 97]. La gestion des liens par référence futée va en effet permettre le maintien des objets liés dans le contexte mémoire de l'application tout en masquant au développeur la décision de chargement et en assurant que seul ce qui sera réellement utilisé sera transféré sur le réseau.

LE DESIGN PATTERN RÉFÉRENCE FUTÉE

Ce *design pattern* a pour objectif de réaliser la navigation au travers d'une association, en masquant les mécanismes de mise à jour du graphe d'objets en mémoire. Cette technique est motivée par le besoin de synchroniser deux graphes d'objets qui, résidants dans deux espaces mémoire distincts, sont image l'un de l'autre. C'est le cas lorsque l'on souhaite distribuer un graphe d'objets, le graphe existe à la fois sur le serveur et sur le client. Cette technique peut également être utilisée entre un graphe d'objets persistants et son équivalent dans la base de données.

Vouloir charger une représentation du graphe dans le contexte client ne signifie pas forcément charger le graphe tout entier. Cependant, l'utilisateur peut demander à naviguer sur une association qui sort des limites de ce qui a été préalablement chargé. La référence futée s'occupe alors du chargement de l'instance demandée, et masque au développeur les mécanismes pour se synchroniser avec l'image.

La structure de la référence futée est détaillée dans le diagramme suivant. Une classe *Référence* vient s'immiscer dans la relation entre le client et son sujet.

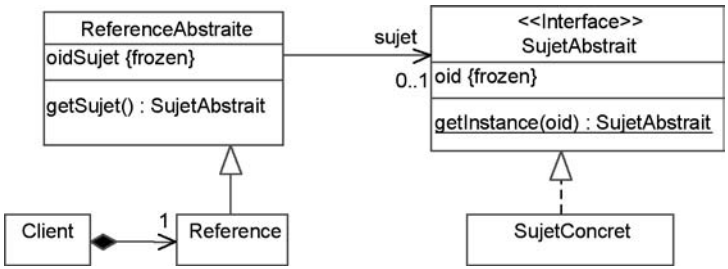

Figure 11-47 : Structure du design pattern Référence futée

Le mécanisme mis en jeu par la référence futée est analogue au mécanisme d'un singleton. Le diagramme de collaboration de la figure 11-48 en montre la dynamique de déclenchement.

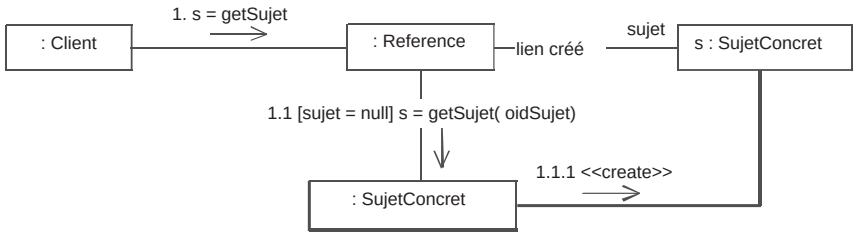

Figure 11-48 : Dynamique du design pattern Référence futée

La référence futée nous permet ainsi de distribuer les associations par l'utilisation des objets sessions ; le diagramme ci-après illustre son utilisation sur la classe Parcours dans la mesure où la classe *PtrSite* réalise une référence futée sur les sites de départ et d'arrivée du parcours.

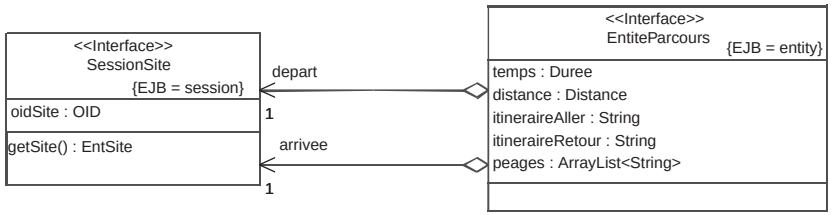

Figure 11-49 : Application d'une référence futée aux liens entre Site et Parcours

Nous allons maintenant récapituler tous les éléments de conception que nous avons introduits pour réaliser une couche métier distribuée au travers de l'étude de cas.

ÉTUDE DE CAS : CONCEPTION DE LA CATÉGORIE *MÉTIER::MISSION*

La distribution des classes métier tient compte des résultats de la conception générique pour organiser techniquement la distribution, mais également des résultats de la conception préliminaire pour structurer les services distribués en interfaces. Dans ce cadre, des EJB sessions distribuent des structures de données pour mettre en œuvre les mécanismes CRUD, mais réalisent en plus les opérations distribuées que l'on a déjà formalisées sous forme d'interfaces en phase de conception préliminaire (voir chapitre 10).

Le diagramme de la figure 11-50 schématise la structure des classes de distribution pour la catégorie Mission. Nous y trouvons un objet session des missions qui est principalement dédié à leur édition CRUD et une classe d'entité des suivis de mission, servant à distribuer l'état d'avancement d'une mission. Le suivi de mission implémente d'une part un état distribué, mais impose d'autre part des mises à jour fréquentes pour les répartiteurs et pour les clients en recherche

d'information sur l'encours de leurs commandes. Nous prévoyons en conséquence qu'un objet session soit également construit pour le suivi de mission.

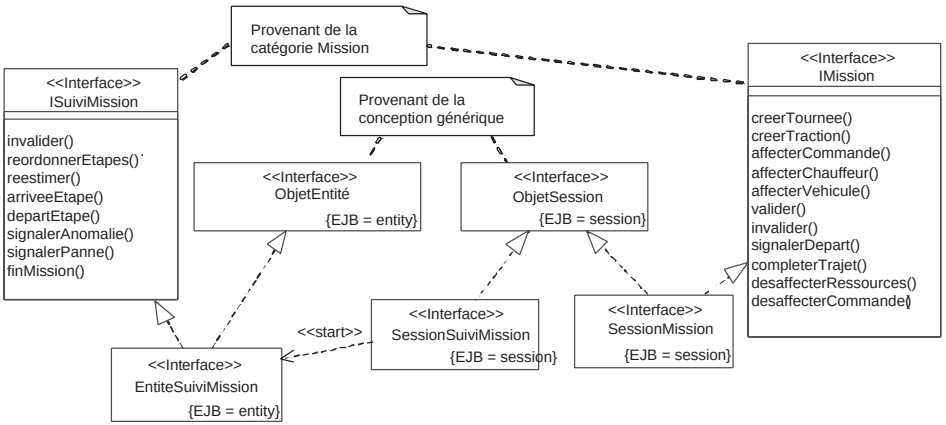

Figure 11-50 : Structure de distribution de la catégorie Métier::Mission

La conception détaillée d'une distribution ne s'arrête pas à la définition de sa structure. L'ensemble des opérations distribuées définissent également des signatures nécessitant le passage d'informations autres que celles contenues dans la structure d'instance représentant la classe. Nous devons donc y ajouter la définition de toutes les structures de données correspondant aux paramètres des opérations.

Conception du stockage des données

La réalisation d'un stockage des instances varie suivant le mode de stockage retenu. Dans tous les cas, la réalisation d'un modèle objet facilite la maintenance des données stockées. Il existe aujourd'hui plusieurs modes de stockage possibles.

- Le système de fichiers est actuellement le moyen le plus rudimentaire de stockage. Avec le mécanisme de sérialisation, Java a fortement simplifié la technique de stockage d'objets dans des fichiers. Le stockage en fichiers ne coûte donc pratiquement rien. Cependant, il ne permet que de lire ou d'écrire une instance par des moyens externes à l'application et il n'a aucune capacité à administrer ou à établir des requêtes complexes sur les données.
- La base de données relationnelle ou SGBDR est un moyen déjà plus sophistiqué. Il existe aujourd'hui une large gamme de SGBDR répondant à des besoins de volume, de distribution et d'exploitation différents. Le SGBDR permet d'administrer les données et d'y accéder par des requêtes

complexes. C'est la technique la plus répandue, que nous avons retenue pour SIVEx. Notre conception aborde donc, à la fin de ce paragraphe, les principes de rapprochement objet-relationnel.

- La base de données objet ou SGBDO constitue la méthode la plus élaborée de toutes. Cette technique élude la conception d'un stockage des données puisqu'elle permet de stocker et d'administrer directement des instances de classe. Cette technique n'a pourtant pas connu un grand succès sur le marché des bases de données.
- La base de données XML ou SGBDX est un concept émergeant qui répond au besoin croissant de stocker des documents XML sans risque d'altération de ces derniers. Dans le cadre de développement orienté objet qui nous occupe, cela signifierait une translation intermédiaire entre nos objets Java et un format XML. Bien que certains composants standards (JDO ou Xerces) facilitent ce travail, il ne nous a pas paru opportun de recourir à ce type de technologie pour SIVEx.

La conception du stockage des données consiste à étudier sous quelle forme les instances sont sauvegardées sur un support physique. Elle s'accompagne également d'une conception de la couche d'accès aux données, qui explicite comment se réalise la transformation du modèle de stockage en modèle mémoire. On peut citer ici le composant *open source* Castor/JDO qui fournit la conception très complète d'une couche d'accès aux données, et tout particulièrement dans un cadre objet-relationnel.


Étude

PASSAGE DU MODÈLE OBJET AU MODÈLE RELATIONNEL

L'utilisation d'un SGBDR impose un changement de représentation entre la structure des classes et la structure des données relationnelles. Les deux structures ayant des analogies, les équivalences [Blaha 97] exprimées au tableau 11-1 sont utilisées pour en réaliser le rapprochement.

Une classe définit une structure de données à laquelle souscrivent des instances ; elle correspond donc à une table du modèle relationnel : chaque attribut donne lieu à une colonne, chaque instance stocke ses données dans une ligne (*T-uplet*) et son OID sert de clé primaire.

Certains attributs de type complexe ne correspondent à aucun des types de SQL ; on rencontre fréquemment ce cas pour les attributs représentant une structure de données. Un type complexe peut être conçu ;

- soit avec plusieurs colonnes, chacune correspondant à un champ de la structure ;
- soit avec une table spécifique dotée d'une clé étrangère pour relier les instances aux valeurs de leur attribut complexe.

Modèle objet	Modèle relationnel
Classe	Table
Attribut de type simple	Colonne
Attribut de type composé	Colonnes ou clé étrangère
Instance	T-uplet
OID	Clé primaire
Association	Clé étrangère ou Table de liens
Héritage	Clé primaire identique sur plusieurs tables

Tableau 11-6 : Équivalences entre les concepts objets et relationnels

Le diagramme suivant (figure 11-51) illustre la conception du stockage de la classe *Commande* dans la table *TblCommande* correspondante. UML définit spécifiquement un stéréotype *table* pour représenter la table d'un schéma relationnel. Nous avons ajouté le stéréotype *join* pour exprimer les jointures que définissent les clés étrangères entre tables.

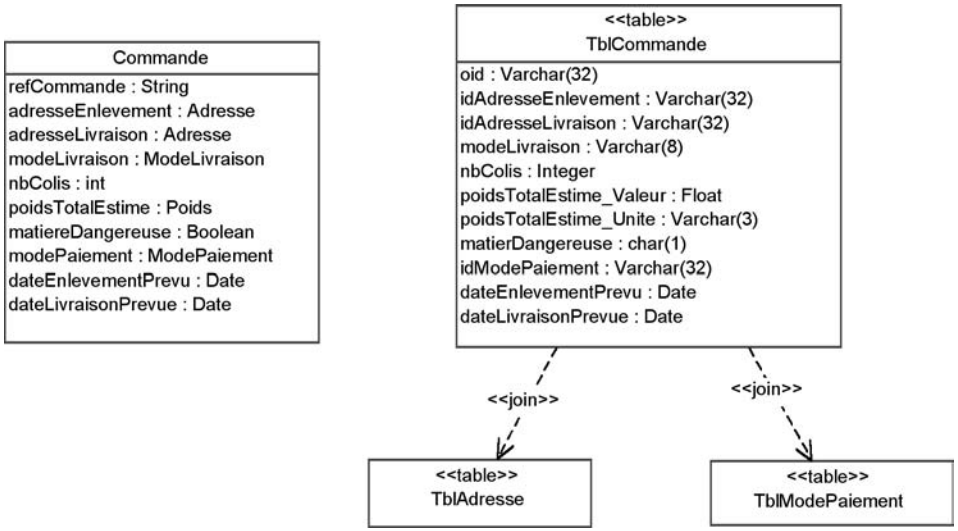

Figure 11-51 : Conception du stockage de la classe Commande avec une table relationnelle

Il est à noter que le schéma relationnel ne permet pas de différencier les associations des agrégations et des compositions. Quel qu'en soit le type, les relations correspondent en effet à une clé étrangère lorsque la multiplicité le permet. Une association multiple, plusieurs à plusieurs, nécessite en revanche la définition d'une table de liens supplémentaire. Cette dernière stocke des couples de clés étrangères provenant de chacune des deux classes de l'association. Le diagramme de la figure 11-53 schématise la conception des associations, simple et multiple, de la classe *EnCoursDeCmd*.

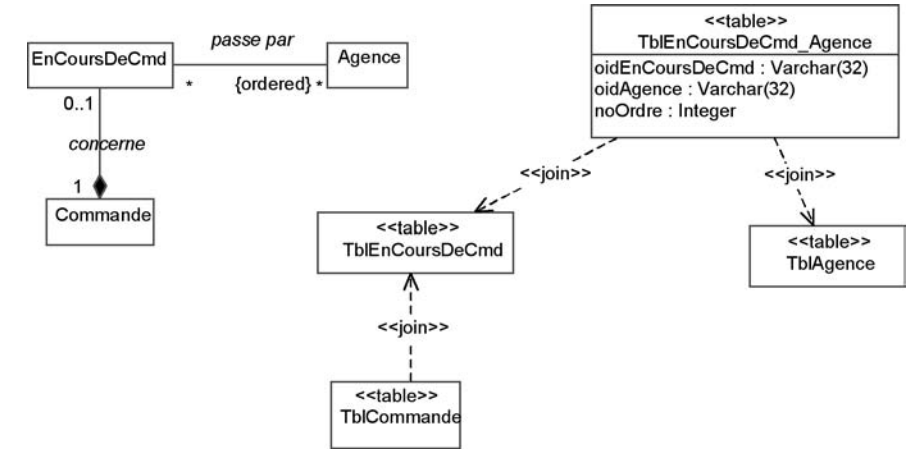

Figure 11-52 : Illustration d'une table servant à stocker une association multiple

La relation d'héritage se définit par le partage du même OID entre les tables provenant d'une même hiérarchie de classes. Dans l'exemple de la figure 11-53, une mission de tournée stocke ses données dans les tables *TblMission* et *TblMissionDeTournee*, tandis qu'une traction conserve les siennes dans les tables *TblMission* et *TblTraction*. Dans les deux cas, c'est une jointure sur les tables appropriées qui permet de reconstituer une instance complète.

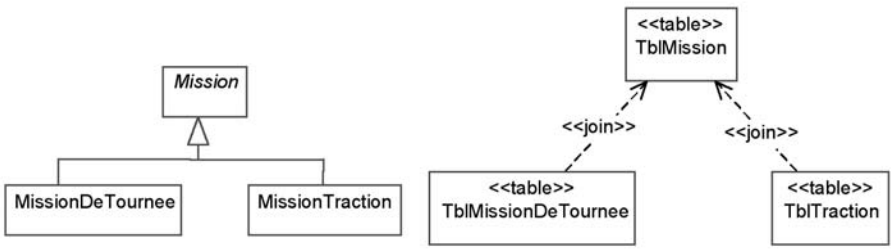

Figure 11-53 : Illustration du stockage d'une relation de généralisation

ÉTUDE DE CAS : CONCEPTION DU STOCKAGE DES CLASSES *MISSION*

Il s'agit ici de définir les tables correspondant à la classe *Mission* et à ses spécialisations. Une fois comprise, la conception du modèle relationnel ne pose aucune difficulté. C'est pourquoi les générateurs de code SQL fournis par les outils CASE apportent généralement une aide facile à mettre en œuvre et appréciable. Dans le diagramme suivant, nous avons simplement appliqué les techniques exposées au paragraphe précédent.

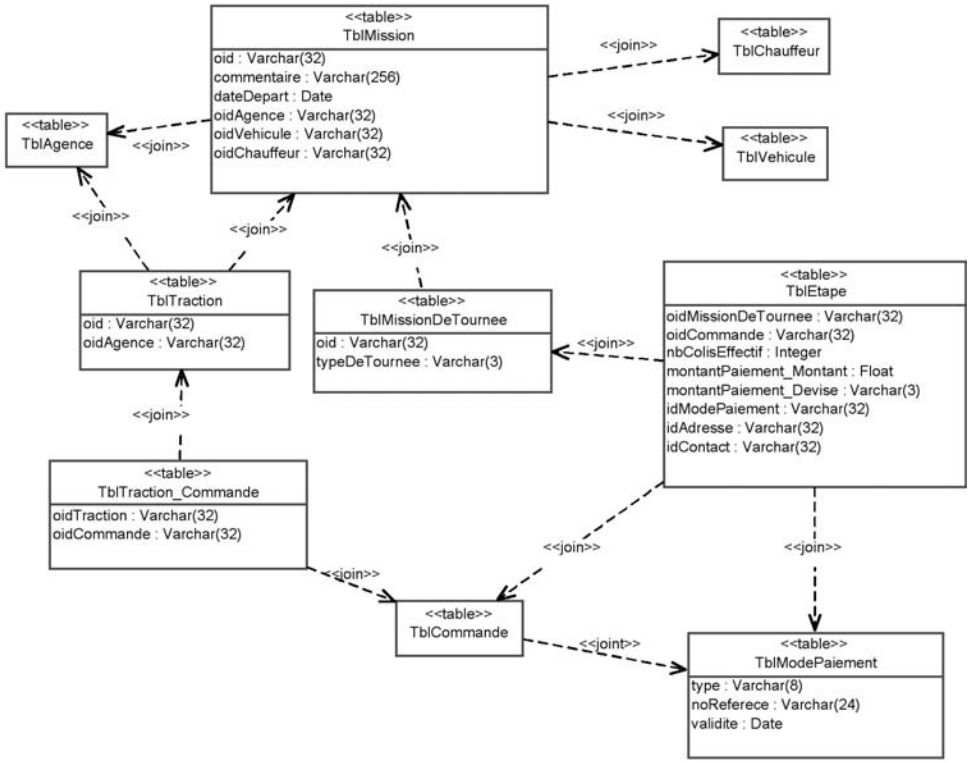

Figure 11-54 : Structure des tables relationnelles pour stocker les missions

Nous avons maintenant terminé l'étude de la conception détaillée et abordé tous les aspects du développement depuis l'analyse jusqu'à la conception détaillée. Vous pouvez mesurer ici le volume d'informations relativement important que produit la phase finale de conception détaillée, et comprendre de ce fait l'importance que nous accordons à la décomposition en catégories et en sous-systèmes ; cette décomposition permet en effet de structurer, d'organiser, et donc de faciliter la maintenance du modèle.

Une fois le modèle de conception terminé, il peut être encore opportun de développer le modèle de configuration logicielle. Il s'agit de préciser comment les différents produits doivent être assemblés à partir des classes, des interfaces et des tables du modèle logique.

Développer la configuration logicielle

Rappelons que la conception préliminaire a défini une structure de configuration logicielle en packages ou sous-systèmes. C'est lorsque toutes les classes sont détaillées à un niveau proche du code que chaque sous-système de configuration logicielle peut être défini.

La technologie Java a grandement simplifié cette dernière étape de conception puisque à chaque classe de conception correspond par défaut une classe Java, qui est elle-même codée dans un fichier source identifiable par le nom de la classe. La structure d'une configuration logicielle peut cependant être améliorée ; il faut à cet effet regrouper les classes en packages Java, lesquels correspondent alors à des packages du modèle de configuration logicielle. Pour plus de lisibilité, il est d'usage d'associer les catégories de conception détaillée aux packages Java. Certaines règles d'organisation du code peuvent également intervenir : il est parfois opportun de séparer les interfaces dans des packages Java spécifiques.

Pour documenter et concevoir la configuration logicielle, il peut être utile de représenter la configuration logicielle à l'aide d'un ou plusieurs diagrammes de composants UML. Dans cette optique, nous avons développé, à titre d'exemple, un extrait de la structure du sous-système de configuration logicielle *Composant Mission*, en soulignant les dépendances nécessaires aux classes *Java Mission* et *SuiviMission*.

Notez que cette dernière étape ne représente pas un intérêt incontournable pour la conception détaillée.

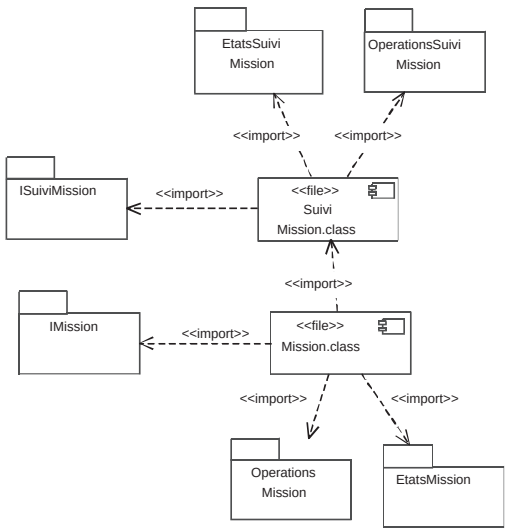

Figure 11-55 : Exemple de configuration logicielle

Phases de réalisation en onception détaillée

La conception détaillée consiste à concevoir et documenter précisément le code qui va être produit. Dans cette phase, toutes les questions concernant la manière de réaliser le système à développer doivent être élucidées. Le produit d'une conception détaillée consiste en l'obtention d'un modèle prêt à coder. Lorsque l'on utilise des langages orientés objet : C++, Java, VB6 ou C#, le concept de classe UML correspond exactement au concept de classe du langage concerné. Cette propriété facilite la compréhension des modèles de conception et donne encore plus d'intérêt à la réalisation d'une conception détaillée avec UML.

L'activité de conception détaillée utilise beaucoup de représentations UML, sans dégager spécialement de préférences entre les différents diagrammes de modélisation dynamique. On retiendra cependant les rôles suivants :

- le diagramme de classes centralise l'organisation des classes de conception, c'est lui qui se transforme le plus aisément en code ;
- les diagrammes d'interactions (séquence, communication et interaction globale) montrent la dynamique d'échanges entre objets en mettant en valeur l'utilité des différentes opérations. On notera une petite préférence pour le diagramme de communication lorsqu'il s'agit de détailler la réalisation d'une méthode ;
- le diagramme d'activité sert spécifiquement à détailler une méthode dont l'algorithmique complexe met en jeu de nombreuses alternatives ;
- les diagrammes d'états permet d'étudier les mécanismes d'une classe à états. On a vu que son emploi est courant lors de la conception des couches de présentation et de l'application. Le diagramme d'états se transforme en code, par l'application du *design pattern* État ;
- enfin, le diagramme de composants sert optionnellement à établir la configuration logicielle des sous-systèmes.

La conception détaillée met en œuvre itérativement un micro-processus de construction, qui s'applique successivement aux différentes couches logicielles du système. En dernier lieu, la conception détaillée précise les modes de fabrication de chacun des sous-systèmes définis lors de la conception préliminaire.

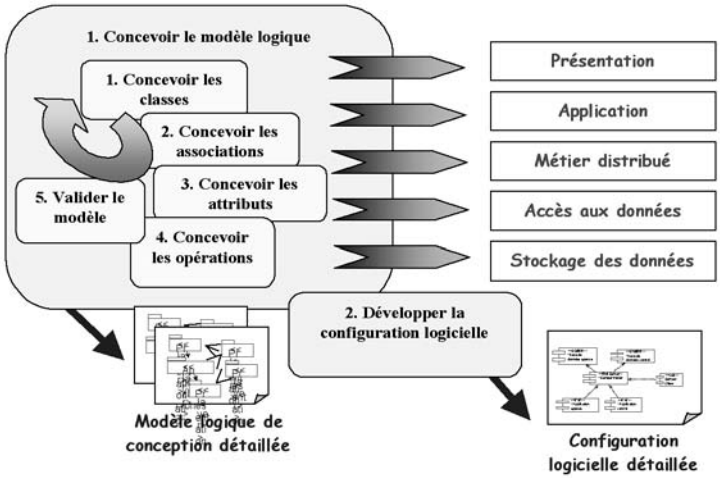

Figure 11-56 : Construction de l'étape de conception détaillée