Conception générique

Objectifs du chapitre

Ce chapitre va vous permettre de voir UML en action lors de la conception. Rappelez-vous qu'UML permet de visualiser, de spécifier, de construire et de documenter. Ces quatre aspects vont être maintenant décrits pour l'activité de conception sur la branche droite du processus 2TUP.

Nous allons donc voir comment :

- élaborer l'étape de conception générique ;
- connaître les points de vue de modélisation utilisés pour cette étape ;
- utiliser UML lors de cette étape ;
- organiser le modèle logique en frameworks techniques ;
- utiliser les cas d'utilisation techniques pour démarrer la conception générique ;
- utiliser les design patterns d'architecture ;
- faire bon usage d'un générateur de code.

Quand intervient la conception générique ?

La conception générique consiste à développer la solution qui répond aux spécifications techniques que nous vous avons présentées au chapitre 5. Cette conception est qualifiée de générique car elle est entièrement indépendante des aspects fonctionnels spécifiés en branche gauche. La conception générique reste donc une activité de la branche droite. Cette étape de conception

constitue les préconisations qu'il vous faut inventer, en tant qu'architecte logiciel de SIVEx, pour que la dizaine de développeurs qui y participe, utilise les composants, idiomes et frameworks les plus efficaces.

Identiquement à notre remarque concernant la spécification technique, la standardisation des techniques de développement, à laquelle nous avons assisté ces dernières années, rend cette étape moins conséquente qu'avant. En effet, la diffusion de frameworks et de composants gratuits, tels qu'EJB, Struts ou JDO, propose en quelque sorte des architectures logicielles clés en main, et généralement de qualité, qu'il suffit de réutiliser.

La conception technique constitue le niveau d'abstraction à atteindre. Les points de vue développés sont les suivants :

- le point de vue logique, qui détaille les classes de la solution ;
- le point de vue d'exploitation, car les premiers composants d'exploitation du système sont conçus à ce niveau ;
- le point de vue de configuration logicielle, qui trace les classes et les versions nécessaires pour fabriquer le système.

La conception générique est terminée lorsque le niveau de détail des diagrammes donne une image suffisante des classes et des composants techniques à intégrer dans le système.

Le développement d'un prototype peut succéder à la conception générique de manière à en valider les principes par le codage et le test. Cette phase de prototypage est fortement conseillée, car la qualité d'une conception générique conditionne généralement celle du développement pour le reste du projet.

La conception préliminaire consiste ensuite à appliquer les concepts génériques aux fonctionnalités du système et à intégrer les composants techniques dans le système considéré dans la perspective de son exploitation.

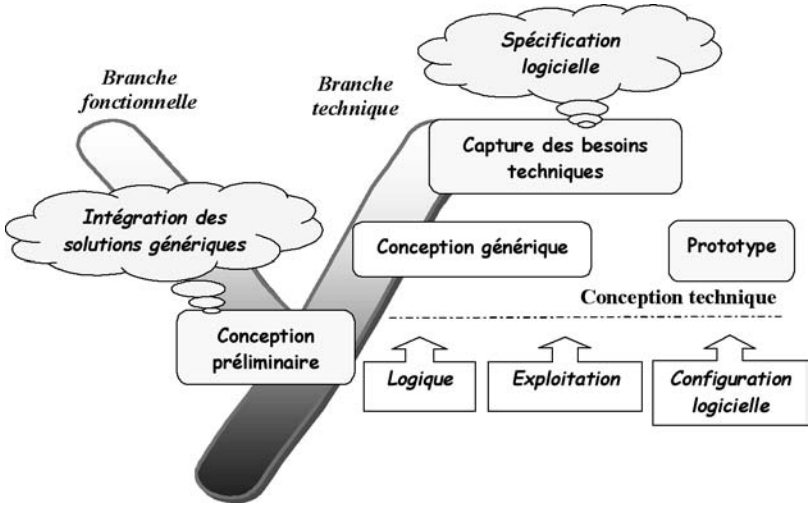

Figure 9-1 : Situation de la conception générique dans 2TUP

En définitive, la conception générique permet de formaliser, sous la forme de classes techniques réutilisables, les règles de conception pour l'ensemble d'un système à développer. Ces règles définissent l'intégrité de conception citée par Brooks [Brooks 90] comme condition de succès d'un projet.

On peut encore considérer que la conception générique développe le squelette technique d'un projet. Si ce squelette est robuste et bien conçu, il pourra soutenir toutes les évolutions fonctionnelles possibles. Si, au contraire, il est mal conçu et qu'il nécessite par la suite de nombreuses retouches, la reprise des erreurs aura des répercussions pouvant se révéler très coûteuses pour la suite du projet.

Conseil

UTILISEZ UML COMME LANGAGE ENTRE CONCEPTEURS

En préambule à ce premier chapitre de conception, nous vous suggérons d'utiliser UML comme langage de conception entre développeurs. En effet, les concepteurs spécifient d'abord les besoins techniques auxquels le système devra répondre, avant d'en développer la solution. UML se révèle un langage de communication efficace pour à la fois spécifier, esquisser et construire une conception.

UML n'entre pas systématiquement en action avec un outil de type CASE (Computer Aided Software Engineering). C'est parfois sur la nappe d'une table de restaurant, lors d'une négociation entre plusieurs concepteurs, que s'élaborent les meilleures solutions ! Aurions-nous pu imaginer un tel support de communication avec du pseudo-code ? C'est par son universalité et sa relative simplicité schématique qu'UML est devenu l'outil des architectes et des concepteurs qui construisent l'intégrité de conception d'un système. Si vous êtes sceptique, nous vous conseillons d'essayer dès maintenant. En tout état de cause, nous vous suggérons de vous entraîner à toute occasion pour acquérir la pratique de la conception objet avec UML.

Éléments mis en jeu

- Diagramme de classes, *frameworks* techniques abstraits et concrets, mécanisme,
- *design patterns*, réutilisation de composants techniques,
- diagramme de composants, composants d'exploitation, composants de configuration logicielle,
- générateurs de code et outils CASE.

Classes et frameworks techniques

L'intégrité de conception s'exprime sous la forme d'un ensemble de classes techniques que les concepteurs du projet vont par la suite réutiliser pour développer les différentes composantes fonctionnelles du système. À titre d'illustration, le mécanisme de contrôle des transactions peut être conçu par un ensemble de classes techniques réutilisées, quelle que soit l'application envisagée dans SIVEx. En d'autres termes, depuis l'application de saisie des commandes jusqu'à l'édition des plans de transport, les applications de SIVEx réutiliseront, grâce à la conception générique, les mêmes classes pour gérer leurs transactions.

On constate cependant qu'une classe technique fonctionne rarement seule, c'est pourquoi le concept-clé de la conception générique est le *framework* technique.


Définition

FRAMEWORK TECHNIQUE

Un *framework* est un réseau de classes qui collaborent à la réalisation d'une responsabilité qui dépasse celle de chacune des classes qui y participent [UML-UG 05]. Un *framework* technique ne concerne que les responsabilités de la branche droite du processus.

À titre d'exemples, les produits Struts et JDO, précédemment cités, sont des illustrations de frameworks techniques, aujourd'hui distribués en Open Source.


Définition

INTERFACE

Au sens UML, une interface est un ensemble d'opérations utilisé pour spécifier le service (ou contrat) d'une classe ou d'un composant [UML-UG 05].

Structurellement, une interface ressemble à une classe avec le mot-clé « interface », qui ne peut ni définir d'attributs, ni définir d'associations navigables vers d'autres classes. Par ailleurs, toutes les opérations d'une interface sont abstraites. Ce concept UML correspond assez directement au concept d'interface dans Java.

Un *framework* peut être abstrait ou concret. Dans le premier cas, il est constitué d'interfaces. La réutilisation du *framework* consiste alors à implémenter ces interfaces, à charge pour le développeur de comprendre et de respecter le domaine de responsabilité technique qui leur échoit. Un *framework* abstrait structure seulement le modèle de configuration logicielle. Il est composé de classes que l'on peut directement réutiliser dans un projet. Un *framework* concret détermine à la fois le modèle de configuration logicielle et le modèle d'exploitation. En réalité, un *framework* n'est pas forcément abstrait ou concret, il est plus souvent une combinaison d'interfaces à implémenter et de classes à réutiliser. Au niveau du modèle d'exploitation, il est livré sous la forme d'un package, d'un fichier JAR, WAR, EAR ou plus largement d'une bibliothèque. Dans le modèle d'exploitation, un *framework* peut donner lieu à des composants distribués ou à des librairies partagées.

Un *framework* représente généralement les mécanismes nécessaires à l'implémentation d'une couche logicielle. Le package Swing de Java, les classes MFC de Microsoft, ou plus récemment le composant *struts* du domaine *www.apache.org* constituent des exemples de *frameworks* qui réalisent la structure technique de la couche de présentation.

Élaboration du modèle logique de conception

Les classes, les interfaces et les *frameworks* techniques représentent les briques de construction d'un modèle logique de conception générique. Les diagrammes de classes en constituent la trame centrale ; autour de celle-ci viennent se greffer différents diagrammes dynamiques en complément de l'étude du fonctionnement de la structure.

La modélisation des classes de conception avec UML ne nécessite pas de reproduire exactement la structure du code qui doit être développé à terme. La conception est avant tout un travail de réflexion et de communication. Il s'agit donc de s'appuyer sur un ensemble de schémas suffisamment précis pour exploiter les possibilités techniques d'une solution et d'en explorer les avantages et inconvénients.


Définition

VALEUR ÉTIQUETÉE

Une valeur étiquetée est une extension des propriétés d'un élément d'UML qui permet d'apporter de nouvelles informations de spécification.

On utilise fréquemment les valeurs étiquetées pour alléger les diagrammes de leurs détails d'implémentation. À titre d'exemple, la valeur étiquetée *serialized* peut être utilisée pour signifier que la classe répond aux mécanismes de sérialisation Java.

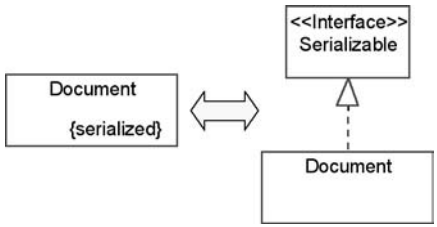

Figure 9-2 : Exemple d'utilisation d'une valeur étiquetée


Conseil

UTILISEZ LES VALEURS ÉTIQUETÉES POUR DOCUMENTER DES MÉCANISMES RÉCURRENTS

D'une part, l'utilisation des valeurs étiquetées allège et simplifie la définition des diagrammes de conception. D'autre part, vous donnerez plus de cohérence à votre conception, car chaque valeur étiquetée standardise, partage et factorise un même mécanisme de conception.

Tout mécanisme introduit doit faire l'objet d'une définition claire dans le modèle. Nous avons introduit un stéréotype de package « mechanism » pour y développer les diagrammes UML de documentation. Vous devrez notamment expliciter l'équivalence de chaque valeur étiquetée comme illustré à la figure 9-2.

ÉTUDE DE CAS : CONCEPTION D'UN *FRAMEWORK DE JOURNALISATION*

Illustrons maintenant la conception d'un *framework* technique avec UML. Nous prendrons pour cela un problème simple, à savoir l'audit des opérations effectuées sur les postes clients et serveurs. La spécification des besoins d'audit précise les contraintes suivantes :

- la mise en œuvre de journaux synchronisés entre les serveurs et les applications en activité-notamment dans le cas des clients lourds ;
- la possibilité de surveiller les erreurs par récupération d'événements SNMP (Simple Network Management Protocol), utilisé ici pour la surveillance des pannes sur le réseau ;
- le respect de formats standard pour faciliter l'exploitation des systèmes de l'entreprise ;
- l'accès concurrent au même journal sur le serveur, qui ne doit pas ralentir les tâches appelantes.

Les schémas obtenus pour décrire les besoins en phase de spécification technique sont décrits à la figure 9-3 et 9-4. Comme nous vous l'avons signalé au chapitre précédent, ces schémas ne peuvent être repris en l'état pour la conception.

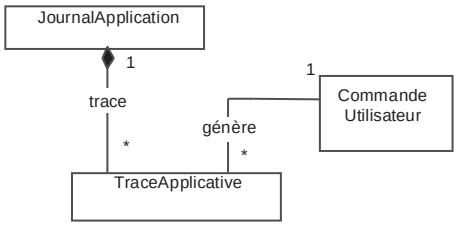

Figure 9-3 : Spécification des besoins d'audit au niveau de la couche de présentation

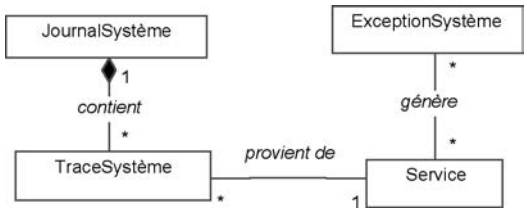

Figure 9-4 : Spécification des besoins d'audit au niveau de la couche métier

Les concepteurs désirent en l'occurrence disposer de la même interface, que l'appel soit sur le poste client ou sur le serveur. Par ailleurs, le fait d'accéder aux journaux de traces homogènes sur les différents serveurs nécessite de distribuer cette interface.

Le déploiement du service en EJB 2.0 a été choisi pour la simplicité des mécanismes qu'il propose dans le cadre de cet exposé sur la conception avec UML. Les techniques de modélisation utilisées dans cet ouvrage restent néanmoins applicables quelle que soit la technologie de distribution employée.

Pour chaque composant EJB, la déclaration d'une distribution passe par la définition d'interfaces dérivant de l'interface standard *Remote*. L'utilisation d'une valeur étiquetée {EJB} sur l'interface et d'une propriété {remote} sur les opérations distribuées permet d'abréger les mécanismes qui seront réellement mis en œuvre dans le code.

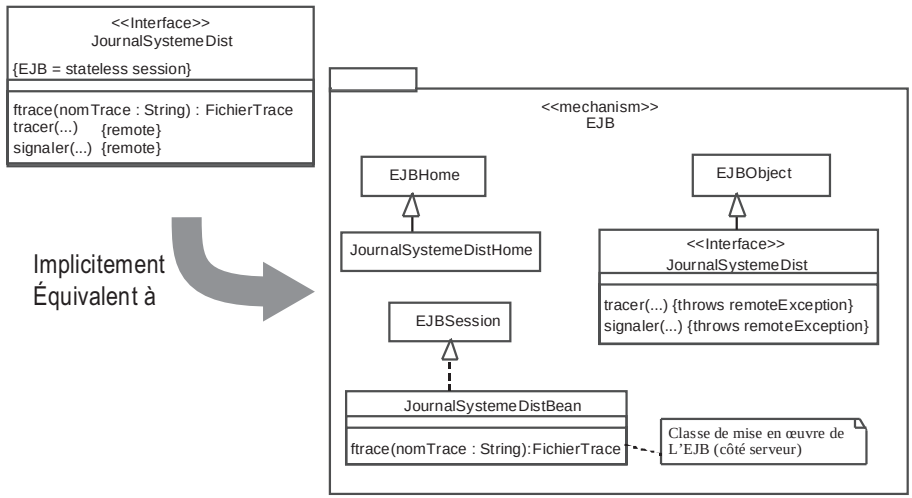

Figure 9-5 : Structure du mécanisme EJB

Pour être plus explicites, nous avons développé la trame du code Java que le diagramme de la figure 9-5 implique :

```
package  SIVEx.frameworks.journalisation.interface;
import   javax.ejb.*;
public interface JournalSystemeDist extends EJBObject {
      void tracer( String nomTrace, String message)
                                    throws RemoteException;
      void signaler(String message) throws RemoteException;
}
```

```
package  SIVEx.frameworks.journalisation.serveur;
import   SIVEx.frameworks.journalisation.interface;
import javax.ejb.*;
public class JournalSystemeDistBean implements EJBSession {
      void tracer( String nomTrace, String message)
                                    throws RemoteException {
      …
      }
      void signaler(String message) throws RemoteException {
      …
      }
      FichierTrace fTrace( String nomTrace) {
      …
      }
}
```

Introduction aux design patterns

Les *design patterns* sont apparus dans les années 1990. Les travaux de la « bande des 4 [1]» [Gamma 95] en ont produit le document fondateur. Cet ouvrage ainsi que des publications ultérieures constituent depuis des catalogues de référence pour les concepteurs objet.


Définition

DESIGN PATTERN

Un *design pattern* est une solution de conception commune à un problème récurrent dans un contexte donné [UML-UG 05].

Dans les faits, les *design patterns* recensent les problématiques communément rencontrées lors des conceptions orientées objet. À titre d'exemple, on peut citer les problématiques suivantes :

- diminution du couplage, en vue de faciliter l'évolution du code,
- séparation des rôles,
- indépendances vis-à-vis des plates-formes matérielles et logicielles,
- réutilisation de code existant,
- facilité d'extension.

L'usage des *design patterns* apporte donc évolutivité, lisibilité et efficacité aux développements. C'est pourquoi leur emploi améliore sensiblement le respect des prescriptions d'architecture [Bushmann 96]. Par ailleurs, ils offrent un transfert de compétence rapide en conception orientée objet, dans la mesure

1. Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides, les quatre auteurs de [Gamma 95], souvent référencés en tant que GoF (Gang of Four).

où ils représentent pour les néophytes un catalogue des meilleures pratiques à adopter. À titre d'illustration, mais aussi parce que nous allons en faire usage dans la conception de SIVEx, nous présentons deux *design patterns* fréquemment utilisés en conception objet.

Le design pattern « singleton »

Le « singleton » [Gamma 95] est l'une des techniques les plus utilisées en conception orientée objet. Il permet de référencer l'instance d'une classe devant être unique par construction. Certains objets techniques prennent en effet une responsabilité particulière dans la gestion logique d'une application. C'est par exemple le cas d'objets comme le « contrôleur des objets chargés en mémoire » ou le « superviseur des vues », qui sont les seuls et uniques représentants de leur classe. Ces objets sont le plus souvent publiquement accessibles. De tels cas de figure sont extrêmement fréquents en conception objet, et le singleton est requis pour les concevoir.

Le singleton repose sur l'utilisation d'une opération de classe, *getInstance() : Instance*, chargée de rapporter à l'appelant la référence de l'objet unique. De plus, le singleton se charge automatiquement de construire l'objet lors du premier appel. Le diagramme UML ci-dessous présente la forme générique d'une classe implémentant un singleton. Vous remarquerez l'usage de la notation UML pour un attribut et une opération de classe.

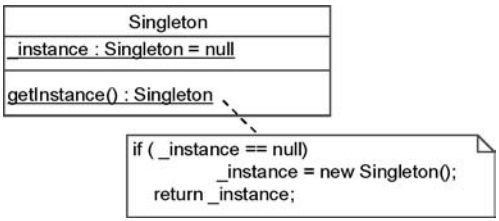

Figure 9-6 : Structure du design pattern singleton

Le singleton est utile au concepteur de SIVEx qui désire créer en local un seul et unique fichier de traces pour toute l'application. De plus, le singleton qui repose sur une méthode de fabrication implicite d'objet pour l'appelant, peut être étendu et fournir un moyen de contrôle sur la création d'instances. Dans l'exemple du *framework* de journalisation, un seul et même fichier de traces doit être créé par jour. Le concepteur peut donc enrichir la méthode de fabrication *getInstance()* pour ajouter un contrôle sur la date de création comme dans le code ci-après. Vous remarquerez que la classe standard *java.util.Calendar* utilise également un singleton pour renvoyer la date courante.

```
package          SIVEx.frameworks.journalisation.serveur;
import              java.util.*;
…
public class FichierTrace {
      private Date             _dateCreation;
      // réalisation de l'attribut de classe du singleton
      private static           FichierTrace _instance
      …
      // réalisation de l'opération de classe du singleton avec contrôles
      public static FichierTrace getInstance() {
         // la classe Calendar utilise un singleton pour la date courante
         int               day = Calendar.getInstance().get( Calendar.DAY_OF_WEEK);
         if ( _instance == null || _dateCreation.getDay() != day ) {
            _instance = new FichierTrace();
         }
         return _instance;
      }
      // le constructeur est privé car le singleton doit être le seul
moyen
      // d'accéder à une instance de la classe.
      private FichierTrace() {
         …
      }
};
```

On a cependant préféré déléguer la capacité de créer de nouveaux fichiers de traces à une autre classe et utiliser de ce fait le *design pattern* « fabrication ».

Le design pattern « fabrication[1] »

Ce *design pattern* [Gamma 95] consiste à donner à une classe particulière, la *fabrique*, la responsabilité de fabriquer les objets d'une autre classe, à savoir les *produits*. La figure 9-7 montre la structure de la fabrication. Vous y remarquerez la représentation UML 2 d'un *design pattern* sous la forme d'une collaboration.

1. *Factory* dans la version anglaise originale.

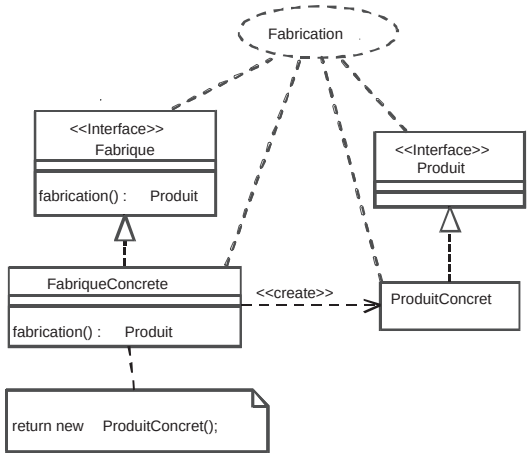

Figure 9-7 : Structure du design pattern fabrication

Au même titre que le singleton, la fabrication est une technique souvent utilisée en conception orientée objet. Nous avons pris à titre d'illustration la journalisation sur le poste client. La classe *Journal* a pour responsabilité de fabriquer les fichiers de traces, et prend ainsi le rôle de facteur. On obtient alors le modèle de la figure 9-8.

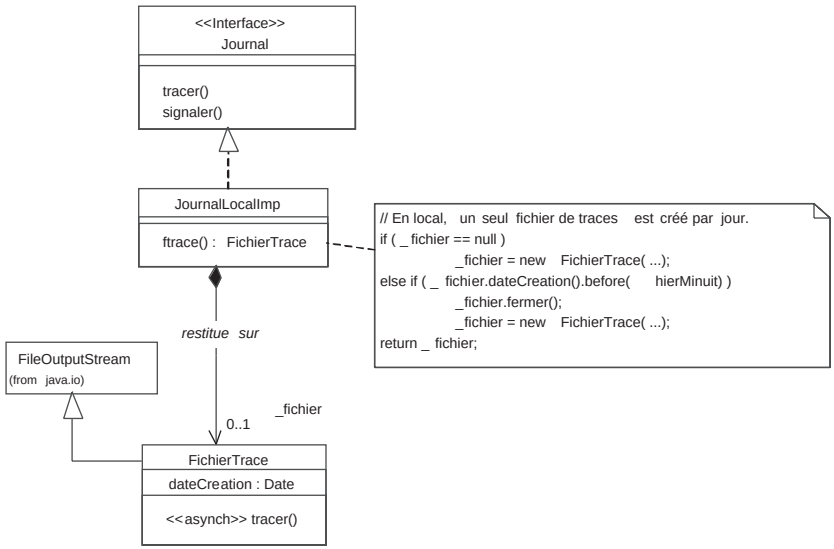

Figure 9-8 : Application d'une fabrication à la journalisation

Nous allons maintenant poursuivre la conception de la journalisation de SIVEx, en appliquant d'autres *design patterns*.

ÉTUDE DE CAS : CONCEPTION D'UNE INTERFACE UNIQUE DE JOURNALISATION

L'un des rôles de la conception est de simplifier les interfaces de manière à offrir le mode opératoire le plus simple possible aux couches exploitantes d'un composant. Dans ce cadre, une interface de journalisation doit être strictement identique, que le service soit local ou distribué. Comme les mécanismes de distribution, en l'occurrence EJB, induisent des contraintes de déclarations étrangères à la seule problématique de journalisation, nous avons recouru au design pattern « adaptateur » [Gamma 95]. L'adaptateur consiste à transformer par délégation les points d'entrée d'un composant que l'on désire intégrer à l'interface voulue par le concepteur.

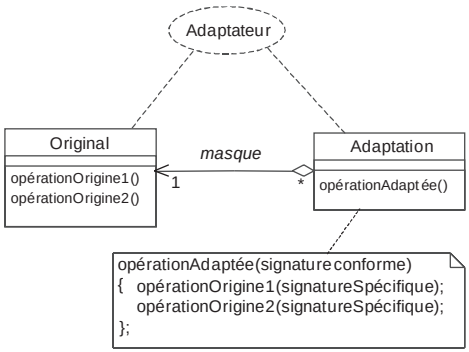

Figure 9-9 : Structure du design pattern Adaptateur

Dans cette optique, l'interface des journaux a été réduite à son strict nécessaire :

- *tracer* produit un message dans l'un des fichiers d'exploitation identifié par un nom de trace ;
- *signaler* produit à la fois une trace et un événement en vue d'alerter le système de supervision du SI.

Deux singletons implémentent l'interface de journalisation. Ils assurent et synchronisent la cohérence des traces entre serveur et client :

- un journal système est chargé d'adapter l'interface de distribution EJB au contexte local. La classe *JournalSysteme* joue donc le rôle d'adaptateur.
- un journal local est chargé de produire les traces sur un fichier local (classe JournalLocalImp).

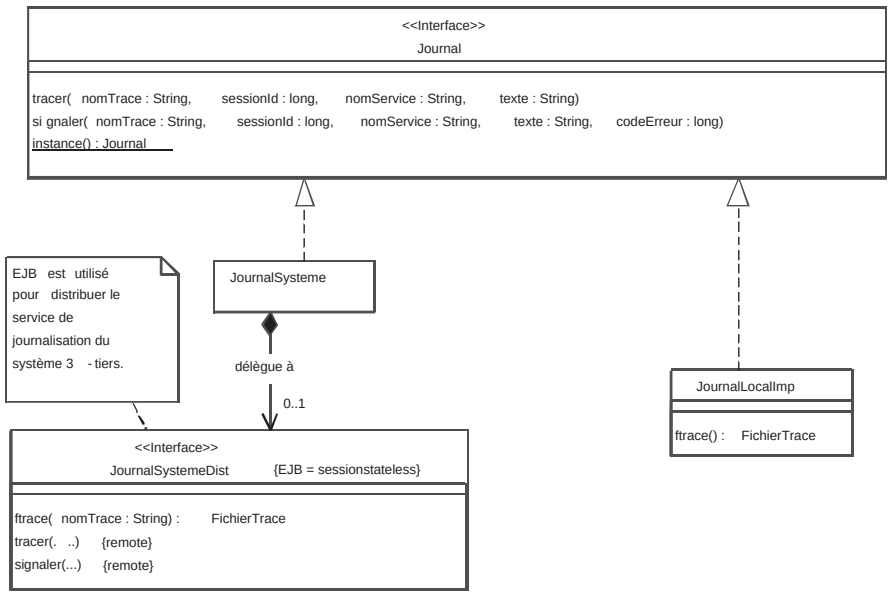

Figure 9-10 : Conception d'une interface unique et adaptation de l'interface distribuée

La distribution des services de journalisation offre une souplesse de répartition des traces. Les composants métier sont en effet clients du système de journalisation, au même titre que les applications. Cela permet d'obtenir une grande cohérence entre les traces produites par les différentes composantes du système et de développer un outil précis de diagnostic d'erreurs.

Construire de nouveaux design patterns

En fonction des besoins de conception ou des pratiques en usage parmi les développeurs, vous pouvez introduire vos propres *design patterns*.

Dans le cas de SIVEx, le serveur de journalisation doit maintenir un fichier par nom de trace. De la sorte, les traces sont réparties sur plusieurs fichiers différents ce qui facilite le travail de recherche d'erreurs mené par l'ingénieur d'exploitation. Le journal système est donc construit sur le modèle d'une fabrication pilotée à la fois par un label – en l'occurrence le nom de la trace – et par le maintien de références sur les instances déjà créées.

Faute de l'avoir rencontré dans les publications existantes, nous avons introduit une variante du *design pattern* fabrication, que nous avons dénommée « fabrication cataloguée ». Ce *design pattern* dont la structure est illustrée à la figure 9-11, définit un catalogue chargé de fabriquer un article lors d'une

première demande de référence inexistante. Le catalogue conserve ensuite la référence de tous les articles fabriqués, pour que lors d'une nouvelle demande de référence, le même article déjà construit soit renvoyé.

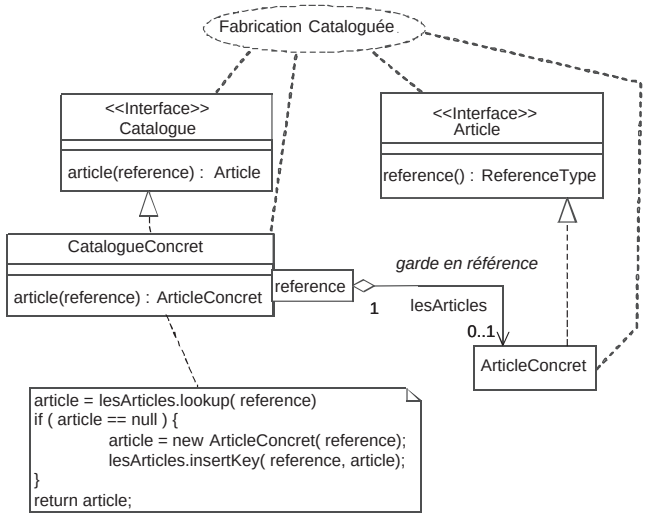

Figure 9-11 : Structure de la fabrication cataloguée

Conformément aux besoins de journalisation, le journal système sert de fabrication cataloguée aux fichiers référencés par un nom de trace. Les règles de construction d'une trace sont cependant un peu plus complexes, puisqu'un nouveau fichier de trace est créé lors d'une modification de date.

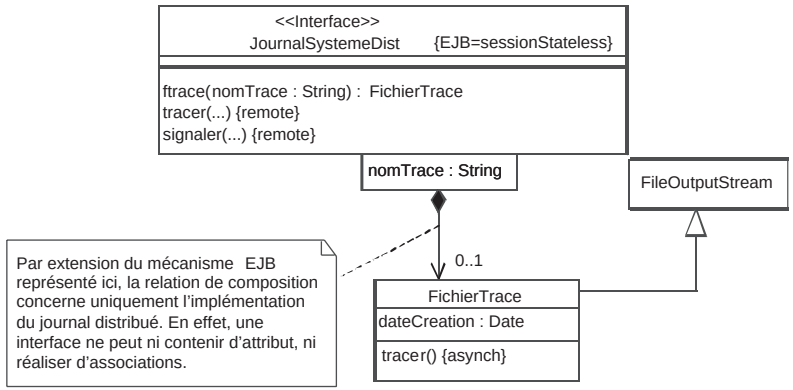

Figure 9-12 : Application d'une fabrication cataloguée au journal système


Conseil

INTÉGREZ LES *DESIGN PATTERNS* DANS LE MODÈLE DE CONCEPTION

Il est important de connaître et de documenter tous les *design patterns* que vous utilisez dans votre conception. À cet effet, nous avons introduit un stéréotype de package : *design pattern*.

Les *design patterns* du domaine public ont une référence bibliographique et seule leur structure statique a besoin d'être rappelée. Les *design patterns* que vous avez construits pour la conception doivent présenter tous les diagrammes statiques et dynamiques nécessaires à leur compréhension.

Conception dynamique d'un *framework*

La conception ne peut se contenter d'une simple étude du modèle structurel. Tout comme en analyse, il est nécessaire d'examiner la dynamique du modèle, en l'occurrence la façon dont les composantes du *framework* se synchronisent.

UML offre différentes approches pour exprimer la dynamique. Nous avons choisi d'utiliser :

- un diagramme d'interaction (diagramme de séquence ou de communication) lorsque le mécanisme étudié implique la synchronisation d'appels provenant de différentes classes ;
- un diagramme d'états lorsque le mécanisme étudié est sous la responsabilité d'une classe qui va occuper plusieurs états, avec des alternatives de transitions entre les états ;
- un diagramme d'activité ou un diagramme global d'interaction, lorsque le mécanisme concerné est entièrement englobé par une seule méthode chargée d'en dérouler les étapes fonctionnelles.

À titre d'illustration, un diagramme de communication permet ici d'étudier comment le journal local et les journaux distribués vont se synchroniser.

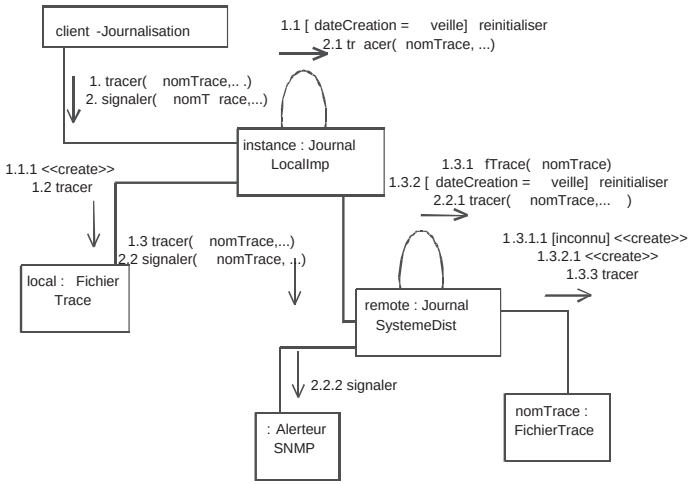

Figure 9-13 : Dynamique de synchronisation entre journaux

Nous avons utilisé la notation décimale (numérotation des messages) pour décrire précisément l'enchaînement des appels entre méthodes. Ainsi, la succession des appels du diagramme ci-dessus correspond au pseudo-code suivant.

```
JournalLocalImp::tracer(…) {
      if ( dateCreation.before( hierMinuit) ) then
         reinitialiser()     //1.1
      _fichier.tracer(…)     //1.2
      JournalSystemeDist::instance().tracer(…)   // 1.3
}
JournalLocalImp::reinitialiser() {
      _fichier = new FichierTrace(…)     // 1.1.1
}
```

```
JournalSystemeDist::tracer(…) {
      FichierTrace _trace = fTrace(…);   // 1.3.1
      if ( _trace.dateCreation.before( hierMinuit) ) then
         reinitialiser()     //1.3.2
   _trace.tracer(…)      // 1.3.3
}
```

ÉTUDE DE CAS : CONCEPTION D'UN MÉCANISME ASYNCHRONE

L'appel d'une opération du journal système correspond, comme on l'a vu, à un accès EJB sur le composant distribué chargé de maintenir les différents fichiers de traces. Malheureusement, EJB 2.0 ne prend en charge les appels asynchrones qu'au travers d'un EJB, dit « message driven », particulier à la norme 2.0. Ainsi, tout appel au composant distribué provoque une file d'attente qui peut ralentir l'exécution des procédures du poste client. Or, l'envoi d'une trace n'implique

aucune donnée en retour au client ; l'exécution du code client a donc de fortes chances de se voir inutilement ralentie.

Une façon de remédier au problème consiste à utiliser la capacité multitâche de Java que l'on va concevoir au travers d'un nouveau mécanisme d'asynchronisme, {asynch} qui s'appliquera aux opérations des classes mises en œuvre sur les serveurs. En l'occurrence, l'appel de l'opération *tracer* va déclencher la création d'une tâche *AsynchTracer*, déclarée comme classe imbriquée de la classe *FichierTrace*. La figure 9-14 schématise la structure associée au mécanisme d'asynchronisme. Vous remarquerez la notation que nous avons adoptée pour exprimer les classes imbriquées de Java et les méthodes *synchronized*.

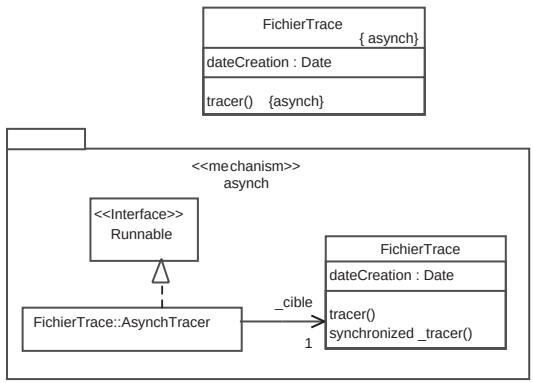

Figure 9-14 : Structure du mécanisme {asynch}

La classe imbriquée *FichierTrace::AsynchTracer* est une tâche Java déclenchée à l'appel de l'opération *tracer*. Cette dernière opération se termine dès que la tâche est lancée, de sorte qu'elle ne sera pas bloquée par l'attente d'une écriture sur le fichier et ne bloquera donc pas le client. En revanche, la tâche lancée attend le temps nécessaire pour écrire sur le fichier via l'opération *_tracer*. Le diagramme de communication de la figure 9-15 décrit cette dynamique et utilise une lettre en tête de notation des messages pour différencier les flots appartenant aux deux tâches.

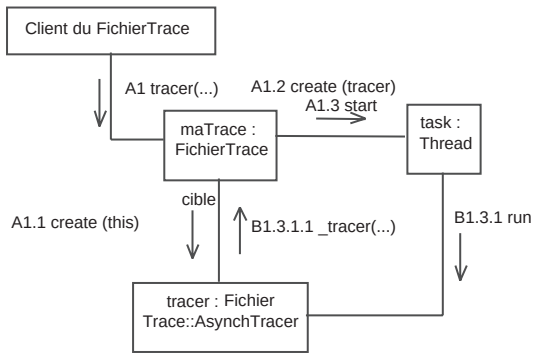

Figure 9-15 : Communication du mécanisme de synchronisation

En conséquence, voici les extraits du code correspondant au système décrit.

```
package SIVEx.frameworks.journalisation.serveur;
…
public class FichierTrace {
      // classe imbriquée Java
      public class AsynchTracer implements Runnable {
         private FichierTrace _cible;
         private String _message;
         AsynchTracer( FichierTrace cible, String message) {
            _cible = cible;
            _message = message;
         }
         public void run() {
            _cible._tracer( _message);
         }
      }
      private Date _dateCreation;
      public void tracer( String message) {
         AsynchTracer tracer = new AsynchTracer( this, message);
         Thread task = new Thread( tracer);
         Task.run(); // libère l'appelant, car retour sans attente.
      }
      public synchronized void _tracer( String message) {
         // traitement normal de l'accès au fichier via le catalogue…
         …
      }
};
```

Organisation du modèle logique de conception technique

Nous allons maintenant revenir au processus 2TUP et examiner les points de vue sur lesquels s'appuie le modèle. Il existe en fait une analogie d'approche entre les branches fonctionnelle et technique, seuls les intervenants et la nature des objectifs changent.

Comparons en effet les deux approches :

- sur la branche gauche, l'analyste détermine les besoins en recourant au modèle de spécification fonctionnelle. Sur la branche droite, l'architecte technique détermine et structure également les besoins techniques suivant les couches du modèle de spécification logicielle ;

- sur la branche gauche, l'analyste détaille ensuite son problème en classes au niveau de chacune des catégories de son modèle structurel. Sur la branche droite, l'architecte technique construit pareillement des classes, des mécanismes et des design patterns au sein de *frameworks* techniques qu'il va organiser dans son modèle logique de conception technique.

Le modèle de conception technique est donc également organisé par packages de classes qui représentent les *frameworks* développés pour résoudre les problèmes purement techniques - cet aspect purement technique fait que la conception, à ce niveau, est qualifiée de générique. Le modèle logique est donc organisé suivant les dépendances qui s'établissent entre *frameworks* techniques.

La figure 9-16 indique l'organisation retenue pour l'étude de cas SIVEx. Notez la façon dont cette organisation de *frameworks* techniques est influencée par les couches logicielles dans lesquelles nous avons exprimé les besoins. Cette influence est bien évidemment une conséquence du périmètre des responsabilités techniques affectées à chaque package : par cohérence et homogénéité en effet, une responsabilité technique doit concerner une seule couche logicielle. Le modèle de conception technique ajoute aussi des services qui sont transverses aux couches. Le *framework* de journalisation est typiquement l'un de ces services transverses utilisables depuis toutes les couches logicielles.

ÉTUDE DE CAS : ORGANISATION DES FRAMEWORKS TECHNIQUES

L'organisation du modèle logique reprend les couches logicielles. À chaque couche correspond un *framework* technique, en partie abstrait, qui définit des interfaces de réalisation des responsabilités logicielles :

- le noyau présentation définit les classes, les interfaces et les mécanismes de base pour réaliser la gestion des objets en mémoire en vue de leur utilisation par les utilisateurs ;
- le noyau applicatif définit ces mêmes éléments pour rafraîchir les vues, charger les modèles de fonctionnement et contrôler les commandes d'une application ;
- le *framework* EAI établit les transformations de données nécessaires à la synchronisation avec les progiciels de SIVEx et le reste du système d'information ;
- le noyau métier définit les éléments permettant d'identifier les objets métier et de les gérer en services distribués ;
- le noyau d'accès aux données définit les mécanismes de chargement, de sauvegarde, de mise à jour et de recherche des objets persistants.

Les autres *frameworks* réalisent des services transverses et complètent la conception des couches logicielles. Les services correspondent aux problématiques de distribution, de sécurité, de journalisation et d'authentification.

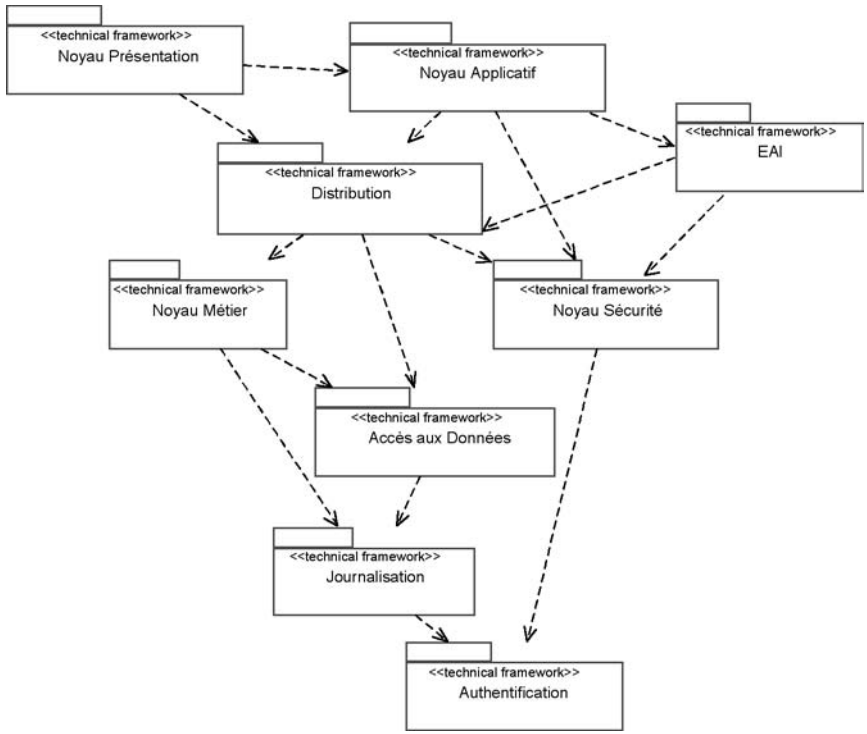

Figure 9-16 : Organisation du modèle logique de SIVEx (diagramme de packages)

La définition des *frameworks* et des dépendances doit obligatoirement compléter ce diagramme de packages d'organisation du modèle logique. En voici deux exemples :

- Noyau Sécurité : *framework* de gestion des habilitations. Il correspond aux besoins transverses des couches Application et Métier.

Ce *framework* dépend de l'authentification pour accéder aux informations relatives aux utilisateurs.

- Accès aux données : *framework* en partie abstrait gérant l'accès aux données et le stockage en base de données. Il couvre les besoins de la couche d'accès aux données.

Ce *framework* dépend de la journalisation pour gérer un journal des requêtes d'accès à la base de données.

Les contraintes de réutilisation dans la conception générique

Au même titre que les contraintes d'optimisation, les contraintes de réutilisation interviennent lors de la conception. La réutilisation répond à plusieurs critères qui sont :

- l'avantage économique, correspondant au temps de développement qu'il aurait fallu consacrer à l'élaboration du composant ou à son équivalent dans le projet ;
- le gain en fiabilité : les composants du commerce sont réputés robustes. Lorsqu'il s'agit de composants maison, il faut parfois évaluer leur degré de confiance, ce qui requiert des efforts à prendre en compte ;
- les contraintes d'intégration : les documentations, les exemples et le découplage technique qui facilitent l'utilisation du composant doivent également être évalués.

Plagiant Molière, nous adopterons la maxime suivante :

IL FAUT RÉUTILISER POUR CONCEVOIR ET CONCEVOIR POUR RÉUTILISER !

Réutiliser pour concevoir s'entend pour ses avantages économiques. La seconde assertion se comprend mieux par l'expérience et mérite quelques explications.

En ajoutant effectivement l'exigence de réutilisation à notre conception, on impose :

- un minimum de documentation,
- une assurance de robustesse,
- le découplage nécessaire à l'extraction de composants et à leur capacité de test hors de tout contexte applicatif.

Ainsi, même si la réutilisation des parties de l'application en cours de développement n'est pas requise, l'envisager apporte implicitement des qualités de découplage, d'évolutivité, de compréhension et de robustesse du code. Autant de propriétés qui vont permettre la diminution notable des phases ultérieures de recette et de mise au point.

Concevoir pour réutiliser est d'autant plus pertinent que le développement de la branche droite est par nature réutilisable. Par son caractère générique, la solution est en effet indépendante de toute fonctionnalité. De ce fait, le cycle en Y a été conçu pour favoriser la réutilisation de l'architecture technique dans son ensemble.

Il est donc bien dans l'intention des concepteurs de SIVEx de réutiliser globalement les mêmes concepts techniques pour toutes les applications. De la gestion du quai jusqu'à la comptabilité, la journalisation, la sécurité, les mécanismes d'accès aux données et de distribution, les interfaces du noyau métier, du noyau applicatif et de la présentation seront identiques.

Remarquez enfin comment la cartographie des dépendances entre les *frameworks* techniques est importante pour la réutilisation. C'est cette carto-

graphie qui permet de maîtriser le couplage entre composants et d'ordonner la réalisation des tests unitaires et d'intégration. C'est encore cette cartographie qui rend possible la subdivision du modèle de conception technique en sous-parties potentiellement réutilisables dans d'autres contextes que SIVEx. Un futur système de l'entreprise pourra par exemple réutiliser le *framework* d'authentification. En raison des principes de réutilisation de la conception, toute sous-partie doit donc être potentiellement réutilisable. Si l'accès aux données nous intéresse, l'organisation du modèle de conception technique nous indique qu'il faudra réutiliser également la journalisation et l'authentification, ou bien procéder à des adaptations.

Ne pas faire

RÉUTILISER SANS COORDINATION

Nous avons souvent rencontré des développeurs consciencieux qui, apportant le plus grand soin à l'organisation de leur code, sont désolés de ne pas voir leur code réutilisé. Inversement, des projets décidant de récupérer un ensemble logiciel réputé réutilisable, ont connu à leurs dépens un supplément de maintenance non prévu [Ezran 99].

Dans les deux cas, aucun effort n'a été apporté à la coordination entre les créateurs et les utilisateurs de la réutilisation. L'expérience acquise en développement objet montre en effet que la réutilisation du logiciel impose d'organiser et de faciliter la concertation [Jacobson 97]. Les facteurs essentiels de réussite sont donc à la fois :

- le respect des règles d'architecture par l'organisation soignée du modèle,
- une implication de la direction pour consentir aux dépenses d'empaquetage en composants,
- l'existence d'un comité d'assistance à la réutilisation, auprès des projets en cours.

Quand bien même ces critères ne sont pas respectés, la maxime « réutiliser pour concevoir et concevoir pour réutiliser » peut être appliquée en tant que règle d'amélioration de la conception. Vous aurez en effet donné à votre conception les qualités attendues par un architecte logiciel : facilité d'intégration, robustesse et capacité d'évolution.

Élaboration du modèle d'exploitation de la conception technique

Définition

COMPOSANT (DÉFINITION UML)

Dans UML, un composant est une partie physique et remplaçable du système qui réalise un ensemble d'interfaces [UML-UG 05].

Pour établir un modèle d'exploitation, nous avons besoin de distinguer deux niveaux de composant : les composants d'exploitation déjà représentés au chapitre 5, et les composants qui servent à la configuration logicielle. Nous ne retrouvons pas cette dichotomie parmi les stéréotypes prédéfinis d'UML :

- « executable » représente un exécutable capable de fonctionner sur une des machines du système physique. Cette caractéristique en fait un composant d'exploitation ;
- « library » correspond à une librairie statique ou dynamique. Seules les librairies dynamiques peuvent être assimilées à un composant d'exploitation dans la mesure où elles sont explicitement installées par l'ingénieur d'exploitation.

Les deux autres stéréotypes que nous avons retenus s'apparentent à la configuration logicielle :

- « file » représente un fichier de code. Dans le cadre d'un développement Java, la correspondance systématique entre une classe et un fichier fait que la représentation des fichiers de code n'apporte aucune information supplémentaire aux diagrammes de classes ;
- « table » correspond à une table définie pour une base de données. Nous savons également qu'il est rarement nécessaire d'en venir à ce niveau de détail pour l'exploitation.

En conception générique, le modèle d'exploitation montre l'organisation des composants correspondant aux différents *frameworks* techniques que l'on peut mettre en œuvre. Pour établir la vue des composants d'exploitation, nous avons défini les stéréotypes ci-après :

- « application » représente un exécutable directement accessible à un utilisateur. Un composant de ce type se déploie généralement sur un poste client ou sur un serveur d'applications.
- « EJB server » est un serveur EJB. Un tel composant distribué, sous la forme de fichier EAR, se déploie généralement sur une machine de type serveur d'applications.
- « DB engine » constitue un moteur de base de données.
- « DB instance » correspond à une instance de la base de données, en principe un ensemble de tables et d'utilisateurs conçus dans un cadre fonctionnel précis.
- « EAI broker » correspond au serveur des messages de transmission des échanges entre les applications du SI.
- « EAI adapter » correspond aux terminaux récepteurs/émetteurs de ces messages qui réalisent le travail de mise à jour au sein des applications.

La vue des composants d'exploitation complète la conception générique. Ce point de vue permet d'identifier les premiers éléments du système logiciel et définit les règles de déploiement et d'intégration des différentes composantes

de SIVEx. La vue d'exploitation précise par exemple l'ordre dans lequel l'exploitant doit initialiser les applications en fonction de leurs dépendances réciproques.

ÉTUDE DE CAS : MODÈLE D'EXPLOITATION DE LA CONCEPTION TECHNIQUE

Voici comment nous avons choisi d'organiser les composants génériques qui devront être intégrés au prototype de validation de la conception générique :

- pour des raisons de performances, la base de données du référentiel métier et le référentiel des informations d'intégrité sont séparés ;
- le composant d'accès aux données correspond à la partie générique du *framework* qui pilote la connexion à la base de données ;
- le composant de journalisation correspond à la partie distribuée du *framework* de synchronisation des traces ;
- le superviseur de la distribution est le chef d'orchestre des composants, il est notamment chargé de démarrer tous les autres composants distribués. La relation de démarrage constitue une dépendance particulière. C'est pourquoi nous avons introduit le stéréotype « start ».

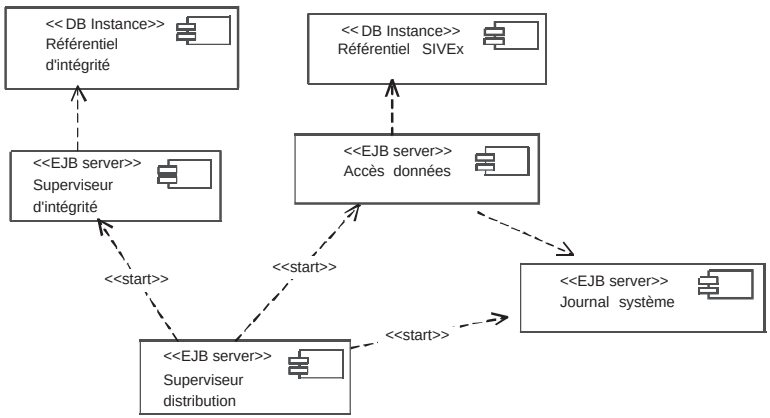

Figure 9-17. : Structure de la vue d'exploitation du modèle de conception technique (diagramme de composants)

À cette vue viendront s'ajouter ultérieurement les composants RMI métier et EAI qui ne sont pas définis dans le cadre de la conception générique.

Élaboration du modèle de configuration logicielle de la conception technique

SOUS-SYSTÈME

En UML, un sous-système correspond à un regroupement d'éléments de modélisation dont le but est de fournir une même unité de comportement au sein du système [UML-UG 05].

Un sous-système est représenté par un package portant le mot-clé *subsystem*, qui ne regroupe pas forcément des composants comme ont pu le suggérer les premières versions d'UML, ou certains outils CASE. Une première ébauche de découpage en sous-systèmes nous est d'ailleurs fournie par l'organisation en *frameworks* techniques du modèle logique ou de manière plus macroscopique en associant un sous-système à chacun des deux progiciels choisis pour SIVEx.

Il peut être intéressant de développer le point de vue de la configuration logicielle, afin d'identifier les sous-ensembles pouvant se fabriquer et s'utiliser indépendamment les uns des autres. Un sous-système se caractérise donc par un procédé de fabrication indépendant. Il s'agit par exemple du fichier *makefile* ou du fichier projet de votre plate-forme de développement. Le sous-système se caractérise en outre par un numéro de version qui fixe son état d'évolution au sein de la construction globale du système. Par l'usage que nous en faisons, le sous-système devient donc l'élément de structuration du modèle de configuration logicielle. Le découpage en sous-systèmes nous sert à identifier les dépendances de compilation des EJB, les dépendances d'implantation des sources Java et les regroupements en fichiers JAR, EAR ou WAR.

La structure en sous-systèmes établit généralement une correspondance directe avec les composants d'exploitation qui représentent autant de cibles de fabrication, et avec tous les sous-ensembles de classes qui sont réutilisés par les différents composants.

Par expérience, le modèle de configuration logicielle n'a d'intérêt que pour des systèmes conséquents tels que SIVEx. Lorsqu'il s'agit de réaliser des applications qui, à terme, ne produisent qu'un ou deux composants de déploiement, l'expression d'un modèle de configuration logicielle est plus que facultative.

ÉTUDE DE CAS : MODÈLE DE CONFIGURATION LOGICIELLE

Le modèle de configuration logicielle est développé au niveau de la conception technique. Les sous-systèmes identifiés sont autant de cibles de fabrication indépendantes, de manière à pouvoir être réutilisés par différents projets, par exemple dans le cas de SIVEx :

- le sous-système « Schéma d'intégrité », qui correspond à la fabrication des tables et des EJB permettant de fabriquer et de lancer en exploitation la base de données servant de « référentiel d'intégrité » ;
- le sous-système « Journalisation », qui concerne l'ensemble des packages Java et des composants EJB fournissant les services de journalisation ;

Prise en compte de la génération de code

La conception générique consiste à développer le squelette technique d'un projet, garant de son intégrité conceptuelle future. Comme nous l'avons déjà mentionné, la conception générique se compose de *frameworks* techniques plus ou moins abstraits.

Le *framework* d'accès aux données est par exemple abstrait parce qu'il est essentiellement composé d'interfaces. Parmi ces interfaces, *IDataClasseBD* représente toute structure échangée avec la base de données. En conception détaillée, cette interface sera implémentée par rapport aux besoins fonctionnels des applications de SIVEx. Elle donnera lieu à des implémentations en partie calquées sur la structure des classes d'analyse. Le diagramme ci-après illustre notre exemple :

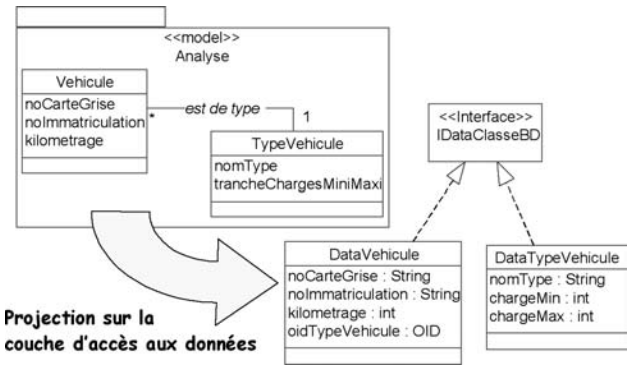

Figure 9-18 : Correspondance entre le modèle d'analyse et un framework technique

On peut donc constater que l'implémentation des *frameworks* abstraits se fonde sur des informations déjà disponibles dans le modèle d'analyse. Par ailleurs, la constitution manuelle du code Java s'avérerait particulièrement pénible, tant pour la création - car il s'agit d'une interprétation quasi-automatique d'informa-

tions déjà disponibles dans d'autres modèles - que pour la maintenance. En effet, toute modification du modèle doit entraîner la mise à niveau correspondante des différentes lignes de code.

L'implémentation des *frameworks* abstraits à partir des définitions fonctionnelles fait donc l'objet d'une écriture de code répétitive et fastidieuse à maintenir. Si la variabilité réside dans les informations que comportent les modèles UML, le générateur de code s'avère particulièrement efficace. Cette approche est aujourd'hui promue par l'OMG sous la dénomination MDA (Model Driven Architecture) comme nous vous l'avons déjà expliqué au chapitre 2 de cet ouvrage.

Il existe des contextes pour lesquels la génération de code complète très utilement la conception générique.

CE QU'ON PEUT ATTENDRE D'UN GÉNÉRATEUR DE CODE

Un générateur de code accompagne la plupart du temps l'outil CASE que vous utilisez pour éditer vos modèles UML. De ce point de vue, on peut déjà séparer les outils CASE en trois familles : ceux qui vous permettent de développer vos propres générateurs, ceux qui vous livrent des générateurs munis de capacités de paramétrage et les outils de dernière génération qui sont conformes à l'architecture MDA.

Si vous estimez que votre projet doit inclure de la génération de code et qu'il doit se conformer à une conception générique particulière, écartez immédiatement les outils qui ne vous donnent pas la possibilité d'obtenir votre propre générateur. Conformément à ce qui vous est expliqué au paragraphe précédent, le générateur de code complète votre conception, et non l'inverse. En conséquence, les générateurs fournis par défaut ne sont utiles que s'ils peuvent s'adapter à vos propres contraintes de conception.

Un générateur de code paramétrable doit vous permettre d'accéder à toutes les informations du modèle UML et de produire des fichiers qui restituent cette information sous le format de votre choix. Avec un générateur, vous pouvez donc aussi bien créer un rapport texte qu'un fichier de code Java. Notez que certains générateurs peuvent se coupler avec les suites bureautiques et produire des documents de travail directement issus du modèle. Le recours aux générateurs est donc un excellent moyen pour synchroniser l'évolution d'un modèle avec le code et la documentation qu'il produit.

Nous avons retenu trois niveaux de complexité de génération :

1. Générer les squelettes : c'est le niveau le plus facile et le plus répandu pour lequel le générateur récupère les classes, leurs attributs, les opérations et les signatures. Le générateur produit alors un fichier Java pour chaque classe incluant des méthodes vides à compléter manuellement.
2. Générer les mécanismes systématiques : c'est le niveau le plus efficace pour compléter une conception générique. Les classes sont générées avec des méthodes spécifiques aux mécanismes du *framework* technique visé. La difficulté de ce niveau est le paramétrage spécifique que vous devez ajouter pour avoir un contrôle plus précis sur les différents cas de figure rencontrés.
3. Générer le corps des méthodes fonctionnelles : pour les opérations liées à un diagramme d'interactions ou les classes liées à un diagramme d'états, le générateur récupère en outre les informations du modèle dynamique afin de générer en Java les méthodes équivalentes à la description UML.

Dans tous les cas, le générateur doit être incrémental pour être opérationnel, ce qui signifie que toute nouvelle génération de code doit préserver le code que vous avez ajouté manuellement.

Un générateur de code doit donc répondre aux trois qualités suivantes :

- il doit permettre de coller à votre conception générique ;
- il doit être capable d'implémenter tout un *framework* abstrait à partir des informations déjà disponibles dans le modèle d'analyse ;
- il doit être incrémental.

Avec de telles spécifications, le générateur devient un projet dans le projet. Il demande de procéder à une analyse, à une conception, puis à un codage. Il implique donc un investissement de ressources. Un bon critère de rentabilité consiste à mesurer l'intérêt d'un générateur de code. À cet effet, vous multipliez le nombre de classes d'analyse par le nombre de *frameworks* concernés. Un intérêt de 150 à 200 *classe*framework* vous fournit alors une limite de décision. Au-delà de cette limite, nous vous conseillons fortement d'utiliser la génération de code.

ÉTUDE DE CAS : CALCUL DE L'OPPORTUNITÉ D'UN GÉNÉRATEUR DE CODE

Recensement des frameworks techniques de SIVEx impliqués par la génération de code :

- framework d'accès aux données,
- framework de distribution,
- framework du noyau client,
- outre les frameworks, on inclut les scripts des tables relationnelles.

Calcul de l'intérêt à la génération de code : 4 * 50 classes issues de l'analyse de SIVEx :

Intérêt à la génération = 200 classe*framework

Nous avons donc intérêt à développer un générateur pour compléter la conception générique.

Ne pas faire

NE PAS SOUS-ESTIMER LE COÛT D'UN GÉNÉRATEUR DE CODE

Le constat que nous avons pu faire sur les différents projets où nous sommes intervenus est le suivant : la génération de code est encore trop rarement exploitée, voire parfois abandonnée suite à l'évaluation des moyens offerts par les outils CASE. Nous avons cependant déjà pu expérimenter des générations de code rentables dans au moins trois contextes de développement différents.

Premier conseil : ne jamais sous-estimer le coût ou les risques accompagnant un outil de génération de code. Le développement d'un générateur est un projet dans le projet qui nécessite les mêmes critères de suivi : pilotage par les risques, développement par incréments et construction autour d'un modèle.

Le second conseil consiste à ne débuter l'analyse du générateur de code qu'une fois le modèle de conception technique stabilisé. Il est vain en effet d'anticiper ou d'évaluer des générateurs de code étrangers à votre conception.

Développement d'un prototype

Vous avez conçu tous les *frameworks* techniques qui vous permettront de répondre aux spécificités techniques de votre système. Vous avez éventuellement développé en sus les générateurs qui en facilitent l'implémentation.

Rappelons encore que l'architecture technique ainsi obtenue garantit l'intégrité conceptuelle de tous vos futurs développements. Vous contribuez ainsi non seulement à une bonne partie du travail de conception, mais apportez également les meilleurs savoir-faire à reproduire pour le reste du projet.

Conseil

DÉVELOPPEZ UN OU PLUSIEURS PROTOTYPES COMME PREUVE DE CONCEPT

Sachant qu'une conception évolue encore lors de son implémentation et que tout changement concernant la conception générique devient coûteux en phase d'implémentation, vous avez intérêt à développer dès maintenant un prototype d'architecture. Vous procéderez ainsi le plus rapidement possible aux mises au point qui s'imposeront à votre conception générique.

La question qui se pose ensuite est la suivante : « Que mettre dans un prototype ? ». Sachez que l'ensemble de vos prototypes doit répondre aux fonctionnalités techniques demandées par le système. Dans le cadre de SIVEx, le prototype doit au moins valider :

- le mécanisme CRUD des objets (Create, Retrieve, Update, Delete) depuis la présentation jusqu'à la base de données,
- les transformations d'objets entre les couches : passer d'un objet de présentation à un objet de distribution puis à un objet métier,
- les mécanismes d'authentification et d'habilitations,
- l'intégrité des données sur le serveur et sur le poste client,
- les synchronisations EAI entre applications,
- les mécanismes de présentation.

Le prototype peut donc viser fonctionnellement un simple éditeur des commandes et des clients de SIVEx en synchronisation avec SAP R3 et SIEBEL. Le prototype doit de plus disposer d'un mode de présentation multivues pour couvrir tous les aspects techniques. Enfin, le prototype valide et prépare les trois niveaux de tests requis à la recette technique du système :

- il inclut le test unitaire des composants techniques pris un à un ;
- il prépare les tests d'intégration dans le cadre de la préparation au développement du système complet ;
- il répond déjà aux spécifications techniques, et notamment au respect des temps de réponse et des exigences de montée en charge.

Le prototype valide et termine éventuellement l'étape de conception générique. Nous sommes maintenant arrivés au terme de ce chapitre et allons en récapituler les phases de réalisation.

Phases de réalisation en conception générique

La conception générique avec UML s'appuie sur le développement de *frameworks* répondant aux spécifications techniques. Aujourd'hui, la conception orientée objet consiste à recourir aux *design patterns* ainsi qu'à schématiser les mécanismes les plus utilisés. L'organisation en architecture technique puis en composants doit ensuite répondre à des objectifs de réutilisation, de fabrication et de déploiement.

Le détail du processus suggéré en conception générique est le suivant :

1. Élaboration du modèle logique de conception technique : concevez les *frameworks* techniques :

- schématisez à l'aide de diagrammes de classes et d'interactions les *design patterns* que vous utiliserez ;
- représentez de la même façon les mécanismes de votre conception et identifiez-les avec des valeurs étiquetées ;

- identifiez et schématisez avec UML les *frameworks* techniques : réutilisez pour concevoir et concevez pour réutiliser ;
- organisez le modèle logique de conception technique – le découpage en frameworks.

2 Élaboration du modèle d'exploitation de conception technique : concevez les composants d'exploitation :

- identifiez les composants d'exploitation correspondant aux *frameworks* techniques ;
- organisez le modèle d'exploitation.

3 Seulement, si l'ampleur du projet le justifie, élaboration du modèle de configuration logicielle de conception technique : concevez les composants de configuration logicielle :

- identifiez les sous-systèmes de fabrication des composants d'exploitation, en fonction des classes et des *frameworks* techniques disponibles ;
- organisez la configuration logicielle, en précisant les dépendances entre sous-systèmes ;
- développez les composants de chaque sous-système si nécessaire.

4 Développez optionnellement un générateur de code (c'est un projet dans le projet) :

- analysez à partir des résultats de la conception générique ;
- concevez ;
- implémentez ;
- testez.

5 Développez un prototype :

- identifiez les objectifs du prototype ;
- implémentez la conception générique ;
- intégrez optionnellement le générateur de code ;
- testez ;
- mettez au point la conception générique et éventuellement le générateur de code.

La réutilisation doit rester un leitmotiv pour toute cette activité, même si elle n'est pas effectivement requise. Certains contextes justifient également de développer un générateur de code pour faciliter l'implémentation des *frameworks* abstraits.

Il est fortement recommandé de mettre en œuvre un ou plusieurs prototypes pour valider les décisions prises lors de cette phase essentielle pour le reste du projet.

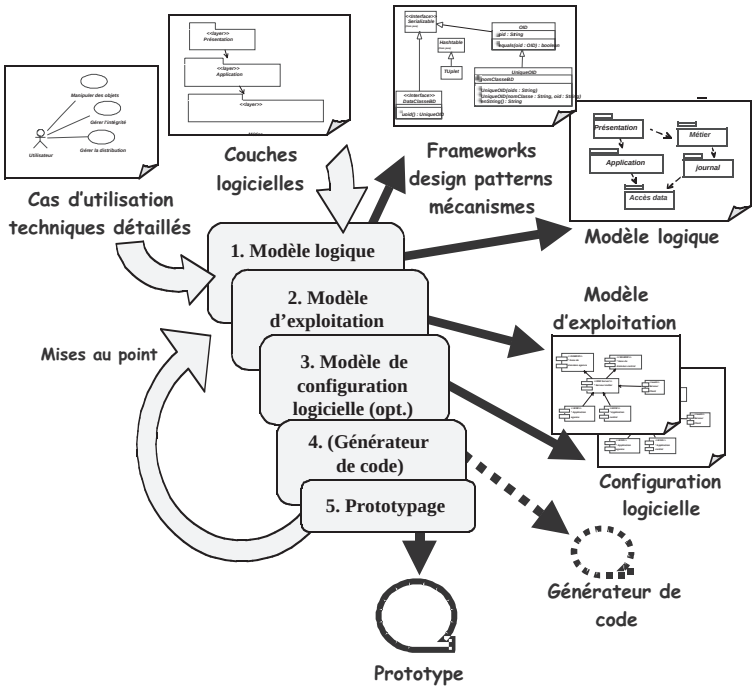

Figure 9-19 : Construction de l'étape de conception générique