Une approche à base de “patrons” pour la spécification et le développement de systèmes d’information

Christine Choppy
LIPN—Université Paris 13
99 Av. J.-B. Clément
93430 Villetaneuse, France
Christine.Choppy@lipn.univ-paris13.fr

Maritta Heisel
Institut für Informatik
Westfälische Wilhelms-Universität Münster
D-48149 Münster, Germany
heisel@uni-muenster.de

Résumé : Les “patrons” (ou “patterns”) tels que les “problem frames” (schémas de problèmes) et les styles d’architecture sont utilisés ici comme support pour la spécification formelle et le développement de systèmes d’information. De nouveaux schémas de problèmes spécifiques pour les systèmes d’information sont proposés pour décrire les sous-problèmes identifiés et pour aider la spécification formelle. La recomposition est effectuée en utilisant une approche basée sur les composants et un style d’architecture qui permet de réunir les différents composants. Une méthode originale est proposée pour accompagner ce processus, avec la mise à profit de certains apports d’UML pour le premier niveau de décomposition, puis l’utilisation des “patterns”. Ces idées sont illustrées sur une étude de cas.
Mots clés : spécification et développement de systèmes d’information, problem frames, styles d’architecture, composants,spécification formelle

1 Motivation

Il est reconnu que ce sont les premières étapes du développement du logiciel qui sont cruciales pour parvenir à la meilleure adéquation entre les besoins exprimés et la réalisation proposée, et pour éliminer au maximum toute source d’erreur. L’accent est donc mis tout d’abord sur une expression précise et non ambigüe des besoins. Différentes approches sont proposées pour aller dans ce sens, avec leurs avantages et inconvénients respectifs. La notation UML [UML] a le bénéfice de la diffusion, de concepts ayant fait leur preuve dans le monde industriel, de notations graphiques, et l’inconvénient de l’absence de sémantique formelle (ce qui pourrait limiter la portée de tout guide méthodologique ...). Les spécifications formelles pallient à ce problème, et, par leur expression précise, conduisent à se poser des questions qui font progresser dans la compréhension du problème à traiter. Un écueil demeure, c’est celui de la taille et/ou de la complexité, qui rendent malaisée la compréhension synthétique et peuvent égarer dans la démarche. Les “patterns” (traduits par schémas ou patrons) proposent des familles de structures fréquemment rencontrées que l’utilisateur est invité à “essayer” (quitte à les adapter) sur le problème à traiter pour ainsi bénéficier de concepts structurants en “prêt à porter”. Les “patterns” peuvent donc être vus comme un moyen élaboré de réutiliser des connaissances acquises par l’expérience. Les problem frames [Jac01] (que nous traduirons par “schémas de problèmes”) sont proposés par M. Jackson pour structurer les problèmes de manière globale. Les styles d’architecture [GS93, BCK98] offrent des structures de granularité plus fine qui sont souvent utilisables au niveau de la conception. L’approche d’architecture par composants [CD01] permet de préciser comment les composants, que la structuration propose de développer indépendamment, seront finalement intégrés.

Il peut être souhaitable de combiner ces approches, pour bénéficier de leurs avantages conjugués, et notre propos ici est de présenter une méthode correspondante que nous avons élaborée pour le cas particulier des systèmes d'information. Notre attention a été attirée sur cette classe de problèmes qui couvre un nombre important d'applications parce qu'il nous semblait que les schémas de problèmes proposés par Jackson ne proposent pas de structuration adaptée. L'approche que nous présentons ici peut être valable de manière générale, mais nous l'avons mise au point plus particulièrement pour cette classe de problèmes. Après avoir proposé des schémas de problèmes spécifiques pour les systèmes d'information (schémas de problèmes pour la consultation et la mise à jour), nous décrivons notre démarche (Figure 1) qui propose un guide méthodologique et indique comment relier de manière systématique les apports (i) des *cas d'utilisation* et des *scénarios* utilisés couramment pour exprimer les besoins, (ii) des *schémas de problèmes* qui permettent d'identifier des structures de problème auxquels nous proposons d'associer les *spécifications formelles* des différents éléments des structures reconnues, (iii) et des *architectures de composants* pour recomposer les différents composants développés.

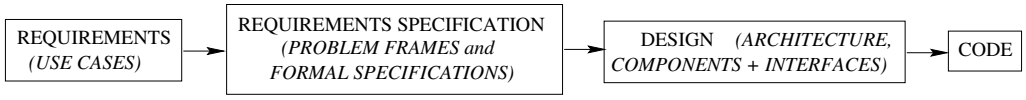

FIG. 1 – *Approche de développement*

Après avoir rappelé (Section 2) les caractéristiques des différents concepts utilisés dans notre démarche, nous proposons (Section 3) une approche méthodologique guidée pour leur utilisation successive et combinée en une suite d'étapes. Nous illustrons ce travail par une étude de cas (Section 4) avant de conclure.

2 Concepts de base

Nous donnons ici quelques éléments sur les concepts utilisés qui nous paraissent utiles pour la compréhension de notre démarche. Nous supposons connues de manière générale les spécifications formelles, et présenterons quelques éléments sur le langage Z quand ils seront utilisés.

2.1 Cas d'utilisation et scénarios

Les cas d'utilisation ont été introduits par Jacobson et al. [JCJO92] à partir de l'idée des scénarios qui donnent les différentes possibilités pour un cas d'utilisation. Les cas d'utilisation sont utilisés pour décrire les besoins relatifs à un système informatique tout en donnant une vue d'ensemble sur ce qui s'y passe. UML [UML] propose un diagramme pour les cas d'utilisation (un exemple est donné Figure 7) et indique qu'il doit être accompagné de descriptions, et que la suite des activités d'un cas d'utilisation est documentée par des spécifications de comportement, telles que les diagrammes d'interaction (diagrammes de séquence, par exemple Figures 8 et 9).

2.2 "Problem Frames"

Un problem frame [Jac01] (ou schéma de problème) est un schéma qui définit de manière intuitive une classe de problèmes identifiée en termes de son contexte et des caractéristiques de ses domaines, de ses interfaces et des besoins. Le système à développer est représenté par la "machine".

Pour chaque schéma de problème un diagramme est établi (Figure 2). Les rectangles simples dénotent les domaines de l'application (qui existent déjà), les rectangles avec une double barre

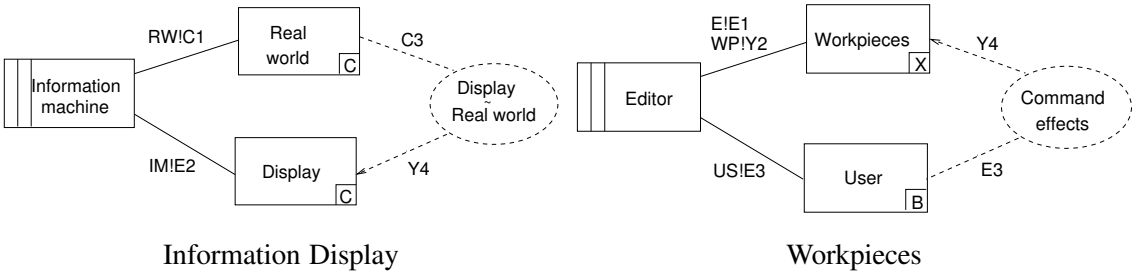

FIG. 2 – *Diagrammes de problem frames*

dénotent les domaines “machine” qui sont à réaliser, et les besoins sont notés par un ovale pointillé. Les lignes qui les relient représentent des interfaces, aussi appelées “phénomènes partagés”. Jackson distingue les domaines “causaux” qui obéissent à certaines lois, les domaines lexicaux qui sont des représentations physiques des données, et les domaines “biddable” (qui émettent des commandes) qui sont des personnes. Jackson définit cinq schémas de base, et nous en présentons deux ici (Figure 2). Le schéma “Information Display” (ou affichage d’information) propose une structure pour les problèmes d’affichage de données physiques du “monde réel”. Le “`C`” indique que le domaine du monde réel (“Real World”) est causal, et `RW!C1` indique que le phénomène `C1` est contrôlé par le monde réel. Le trait pointillé représente une référence aux besoins, et la flèche indique que c’est une référence avec contrainte. Le schéma “Workpieces” est utilisé pour des outils qui permettent à l’utilisateur de créer et éditer une classe d’objets graphiques ou textuels qui puissent être copiés, imprimés. Le “`X`” indique que le domaine “Workpieces” est lexical (ou inerte). L’utilisation d’un schéma de problème consiste à instancier les domaines, les interfaces et les besoins.

2.3 Styles d’architecture

Les styles d’architecture [GS93, BCK98] sont des schémas d’architectures logicielles qui sont caractérisés par

- un ensemble de types de composants (par exemple répertoire de données, processus, ...) qui effectuent certaines fonctions à l’exécution,
- une répartition topologique de ces composants indiquant leurs relations à l’exécution,
- un ensemble de contraintes sémantiques,
- un ensemble de connecteurs pour la communication, la coordination ou la coopération entre composants.

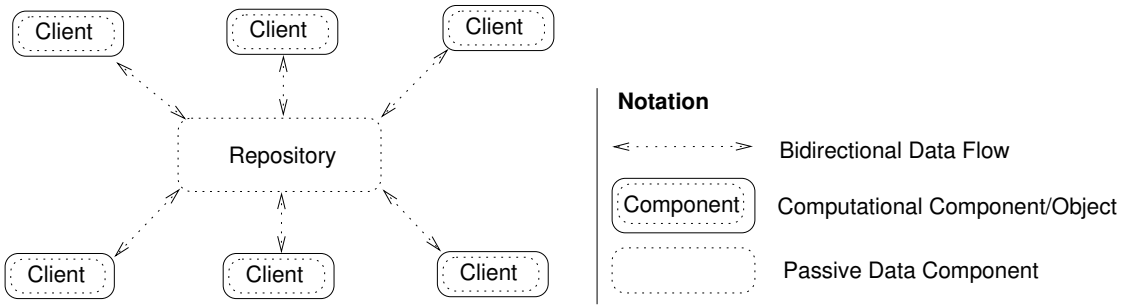

FIG. 3 – *Style d’architecture centré sur les données*

Parmi les principaux styles d’architecture, le style “repository”(Figure 3) , centré sur les données, où différents clients accèdent à des données partagées, convient bien pour les systèmes d’information.

2.4 L'ingénierie du logiciel basée sur les composants

Dans le domaine des styles d'architecture, le mot "composant" désigne juste une partie de logiciel qui effectue quelques calculs. Il n'est pas demandé que cette partie de logiciel satisfasse des contraintes particulières.

Depuis quelques années cependant, est apparu un nouveau domaine de l'ingénierie du logiciel basée sur les composants [Szy99]. L'idée de base est de construire des logiciels à partir de parties de logiciel pré-fabriquées, qui sont bien encaspulées et relativement indépendantes. Ces parties de logiciel sont aussi nommées "composants", mais dans ce contexte, les composants doivent satisfaire les conditions suivantes :

- Tous les services rendus et tous les services requis par un composant sont accessibles uniquement via des *interfaces* bien définies.
- Un composant adhère à un *modèle de composants*. Ce modèle spécifie entre autres des conventions syntaxiques pour la définition des interfaces et la manière dont les composants communiquent. Il sert à garantir l'interopérabilité de plusieurs composants qui adhèrent au même modèle de composant.
  Des exemples de modèles de composants existants sont les *JavaBeans* [Sun97], les *Enterprise Java Beans* [Sun01], *Microsoft COM*$^+$ [Mic02], et le *CORBA Component Model* [Obj02].
- Les composants sont intégrés sous forme binaire. Il est possible que le code source ne soit pas accessible à un "consommateur" de composants (c'est un agent qui compose des systèmes à partir de composants). C'est pourquoi la spécification d'un composant doit contenir tout information nécessaire à son utilisation.

Dans l'approche de l'ingénierie du logiciel basée sur les composants les principes architectureaux jouent aussi un rôle important, parce que la composition d'un système à partir des composants est effectuée selon une architecture choisie. Les deux domaines sont donc assez fortement liés.

Notre approche consiste en la décomposition d'un problème d'information complexe en plusieurs sous-problèmes, qui sont tous décrits à l'aide de schémas de problèmes. Pour chaque sous-problème, un logiciel doit être développé, et le système entier consiste en une combinaison appropriée de ces logiciels qui résolvent les sous-problèmes.

Nous proposons de réaliser les logiciels résolvants les sous-problèmes sous forme de composants, avec des interfaces dérivées des interfaces définies en instanciant les schémas de problèmes. La combinaison des différents composants se fait à l'aide d'une *architecture de composants*, comme par exemple proposé par Cheesman et Daniels [CD01].

3 Description de l'approche proposée

Nous proposons des schémas de problèmes spécifiques pour les systèmes d'information (schémas de problèmes pour la consultation et la mise à jour), puis nous décrivons notre démarche de développement qui permet de concilier les bénéfices apportés par les concepts structurants des schémas de problèmes et des styles d'architecture, ainsi que les concepts d'intégration des composants, avec les bases solides apportées par les spécifications formelles.

3.1 Schémas de problèmes pour les systèmes d'information

Les principales opérations des systèmes d'information sont les mises à jour des informations dans une base de données, et les requêtes pour obtenir certaines informations. Il est habituel de

pouvoir effectuer simultanément plusieurs opérations de requête, alors que pour une opération de mise à jour, un “verrou” empêchera toute autre opération simultanée. Pour les schémas de problèmes (ou “problem frames”) des systèmes d’information, nous utilisons un domaine pour la base de données (DBM ou DataBase Model) qui, dans le cas d’une mise à jour, sera “contraint” (en ce sens que les modifications des valeurs doivent être conformes aux besoins) en fonction des règles de mise à jour, et qui ne sera pas contraint dans le cas de requêtes. Nous distinguerons donc ces deux cas.

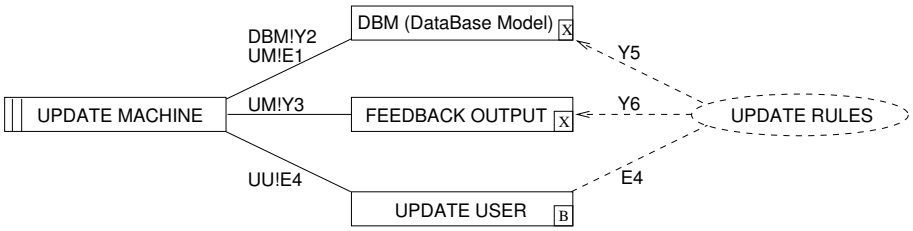

FIG. 4 – *Schéma de mise à jour*

Update Le schéma que nous proposons dans la figure 4 pour la mise à jour peut être vu comme une adaptation et une extension du schéma de problème “workpieces” (Figure 2). [1] Les composants du domaine que nous identifions sont un modèle de base de données (“X” indique qu’il s’agit d’un domaine lexical, c’est-à-dire passif), l’utilisateur qui émet des commandes de mise à jour (“B” indique qu’il s’agit d’un domaine “biddable”, qui émet des commandes), et une sortie de rétroaction vers l’utilisateur. Les besoins sont exprimés par les règles de mise à jour, et la machine qui effectue la mise à jour est à concevoir. Les interfaces que nous identifions prennent en compte le fait qu’une opération de mise à jour peut être précédée d’une requête afin de vérifier que ses préconditions sont satisfaites :

• E4 (aussi dans UU!E4) commande de l’utilisateur impliquant une mise à jour

• E1 (dans UM!E1) phénomène de requête ou de mise à jour contrôlé par la machine

• Y2 (dans DBM!Y2) messages d’information sur l’état de la base de données (et sur son historique)

• Y3 (dans UM!Y3) message de sortie contrôlé par la machine informant du succès ou des raisons d’échec de la commande de l’utilisateur

• Y5 effets de la commande de l’utilisateur sur la base de données, en accord avec les règles de mise à jour, et exprimés par des valeurs de données

• Y6 effets de la commande de l’utilisateur sur les messages de sortie, en accord avec les règles de mise à jour, et exprimés par des valeurs de données.

On notera que les interfaces étiquetées par Y5 et Y6 comportent des flèches vers le modèle de base de données et la sortie utilisateur qui expriment des contraintes.

Query Le schéma que nous proposons pour la requête dans la figure 5 peut être vu comme une adaptation et une extension du schémas de problèmes “information display” (Figure 2). Les composants du domaine que nous identifions sont un modèle de base de données, l’utilisateur qui émet des commandes de requête, et une sortie vers l’utilisateur. Les besoins sont exprimés par les règles de requête, et la machine qui effectue la requête est à concevoir. Les interfaces que nous identifions sont les suivantes :

1. Notons que le schéma que nous proposons est aussi proche du schéma “Commanded Information” que Jackson propose comme variante de “Information Display” avec un utilisateur dont les domaines sont causaux (et non lexicaux), et avec une interface simple entre la machine et “Real World”.

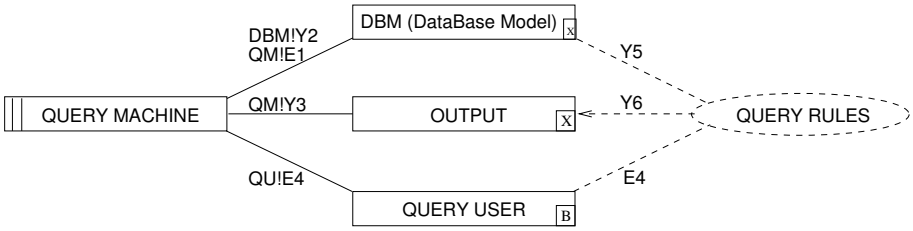

FIG. 5 – *Schéma de requête*

• `E4` (aussi dans `QU!E4`) commande de l'utilisateur impliquant une requête
• `E1` (dans `QM!E1`) phénomène de requête contrôlé par la machine
• `Y2` (dans `DBM!Y2`) messages d'information sur l'état de la base de données (et sur son historique)
• `Y3` (dans `QM!Y3`) message de sortie contrôlé par la machine, réponse à la requête de l'utilisateur ou message d'erreur
• `Y5` informations sur l'état de la base de données en rapport avec la commande de l'utilisateur (en accord avec les règles des requêtes), exprimés par des valeurs de données
• `Y6` effets de la commande de l'utilisateur sur les données contenues dans les messages de sortie (en accord avec les règles des requêtes).
On notera que, dans ce schéma, seule l'interface étiquetée par `Y6` comporte une flèche de contrainte vers la sortie produite. Les deux schémas sont donc très proches, mais il nous paraît important de les distinguer car seule l'opération de mise à jour requiert un verrou sur la base de données, et c'est utile de disposer de cette information notamment lorsque les schémas sont composés.

3.2 Méthode de développement

Nous proposons, pour une certaine classe de systèmes, un guide méthodologique pour une utilisation combinée des concepts présentés en Section 2 avec les spécifications formelles.

Critères pour les systèmes considérés Le but des systèmes auxquels nous nous adressons est la gestion de données. L'environnement de ces systèmes est un "business domain" ou domaine d'affaires, c'est-à-dire une organisation créée par l'homme, qui obéit à des règles qui ne sont pas du domaine de la physique. Il n'y a ni capteurs ni actionneurs, ni aucun moyen informatique de garantir la correspondance entre le monde réel et son modèle. C'est la responsabilité des personnes d'informer le système des changements d'états du monde réel.

Etape 0 Si le système examiné répond à ces critères, alors il faut d'abord en donner un diagramme de cas d'utilisation et fournir une modélisation du monde réel (par exemple avec un diagramme de classes). Ce modèle sera commun aux différentes machines. A cet effet, on applique la méthode d'analyse orientée objet habituelle, d'abord on identifie les acteurs, qui communiquent avec le système, mais qui ne font pas partie du système. Ensuite, on identifie les différents cas d'utilisation. Chaque cas d'utilisation doit être précisé par une description des suites d'actions qui appartiennent au cas d'utilisation. Dans cet article, des diagrammes de séquence sont utilisés pour décrire des scénarios.

Les scénarios contiennent des flèches d'un acteur vers le système. Ces flèches correspondent à des opérations disponibles pour l'acteur. Dans la réalisation du système, chacune de ces opérations fera partie d'une interface correspondant au cas d'utilisation en question (voir description de l'étape 3). L'identification des différents cas d'utilisation fournit un cadre naturel de décomposition sur lequel s'appuie le travail de l'étape suivante.

Etape 1 Pour chaque cas d'utilisation, une instance de schéma de problème est fournie, si le cas d'utilisation induit un changement d'état dans le modèle, le schéma de mise à jour ("update machine") sera utilisé, sinon ce sera le schéma de requête ("query machine"). Il est aussi possible de constater qu'une décomposition supplémentaire est nécessaire, et/ou qu'un autre type de schéma de problème doit être utilisé (par exemple, le "translation frame" [Jac01] qui peut intervenir pour des traitements de données, des calculs).

L'instantiation des domaines `UPDATE USER` et `QUERY USER` (voir figures 4 et 5) utilise l'information élicitée dans l'étape 0 : chaque opération disponible pour un acteur devient une commande et par conséquence un élément du domaine `UPDATE USER` ou bien `QUERY USER`.

A l'issue de cette étape, pour chaque cas d'utilisation, une instance de schéma de problème est établie.

Etape 2 Il s'agit ici d'associer à chaque instance de schéma de problème la spécification formelle correspondante. Pour ce faire, la méthode (proposée dans [CR00] et développée également dans [CH03b]) qui consiste à associer à chaque élément de diagramme de schéma de problème un élément de spécification est appliquée. Les types des données manipulées doivent également être spécifiés. Le langage de spécification formelle Z [Spi92] est utilisé ici pour mettre à profit la notion d'état qu'il propose.

A l'issue de cette étape une spécification en Z décrivant formellement les instances de schémas de problèmes (correspondant à chaque cas d'utilisation) est établie. Cette spécification décrit précisément les différents domaines et la machine à réaliser. Le travail effectué à cette étape permet de bénéficier de la rigueur apportée par la spécification formelle, et des questions posées pour spécifier les éléments requis, ce qui apporte des précisions à un niveau plus fin sur la réalisation demandée.

Etape 3 La spécification d'une machine pour chaque sous-problème fait partie du résultat de l'étape 2. Chaque machine correspond à un cas d'utilisation, utilise la même base de données DBM, et est réalisée par un composant. Nous proposons à présent une méthode d'intégration des composants réalisés d'après leur spécification obtenue à l'étape précédente. La figure 6 montre l'architecture de composants que nous proposons pour mettre ensemble les différentes machines. Cette architecture est une instance du style d'architecture "repository" (Figure 3) où les données partagées sont dans *Database*, et les clients sont les différents composants *UseCase_JMgr*.

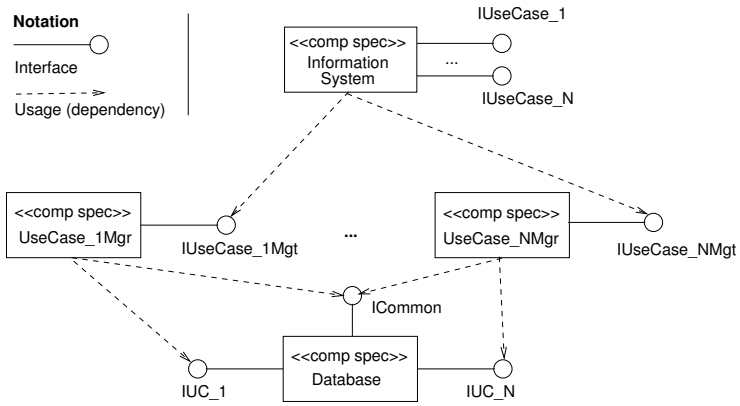

FIG. 6 – *Architecture de composants du système global*

Il y a un composant pour chaque machine et pour le système global, ainsi que pour la base de données. Le système global *Information System* met à la disposition de son environnement une in-

terface pour chaque cas d'utilisation *J*, nommées ici *IUseCase_J*. Les composants correspondant aux machines des sous-problèmes sont nommées *UseCase_JMgr* (selon une convention utilisée par Cheesman et Daniels [CD01]). Pour réaliser une interface *IUseCase_J*, il faut utiliser le composant *UseCase_JMgr* via son interface *UseCase_JMgt*, ce qui est indiqué par les flèches dénotant des dépendances.

Tous les composants *UseCase_JMgr* communiquent avec la base de données. Celle-ci met une interface *IUC_J* à la disposition de chaque composant *UseCase_JMgr*. Pour ne pas avoir trop de répétitions dans les interfaces *IUC_J*, les opérations qui sont utilisées par plus d'un composant *UseCase_JMgr*, sont mises dans une interface *ICommon*.

Les opérations qui constituent les interfaces *UseCase_JMgt* correspondent aux interfaces des différentes *Update Machine*s et *Query Machine*s développées dans l'étape 1. Les opérations des interfaces *IUseCase_J* correspondent aux commandes disponibles aux utilisateurs du sous-problème correspondant. Les opérations des interfaces *UseCase_JMgt* doivent fournir les éléments nécessaires à la réalisation des opérations des interfaces *IUseCase_J*. Cette condition vérifie que les solutions de tous les sous-problèmes suffisent pour résoudre le problème de départ.

Pour spécifier la communication des différents composants plus en détail, on peut utiliser des diagrammes de collaboration (comme par exemple dans l'approche de Cheesman et Daniels [CD01]).

4 Etude de cas: une boutique en ligne

Un système pour un commerce sur internet offre les services suivants :
- Les clients peuvent rechercher si les produits qui les intéressent sont en vente, obtenir des informations sur ces produits, et passer des commandes.
- Les employés s'occupent du traitement des commandes et du réapprovisionnement du stock.
- Le gestionnaire peut demander des rapports sur les statistiques des ventes, décider d'interrompre la commercialisation de certains produits ou de lancer la commercialisation de nouveaux produits.

4.1 Etape 0 : cas d'utilisation

La description ci-dessus conduit à identifier les différents cas d'utilisation suivants (Figure 7) :

(i) pour un acteur client de cette boutique ("Customer"), la recherche d'informations ("Browse"), et l'envoi de commandes ("Send Order"),

(ii) pour un acteur employé, le traitement des commandes ("Process order"), et le réapprovisionnement de stock ("Refill stock"),

(iii) enfin pour un acteur gérant de la boutique, la prise de décisions concernant la commercialisation des produits ("Take Product Decisions"), et la demande de rapports de gestion ("Require Management Reports").

Nous traiterons plus en détail les deux premiers cas d'utilisation, et nous donnons leurs scénarios possibles d'utilisation exprimés par des diagrammes de séquence. Un client peut (Figure 8) rechercher des produits selon certains critères `cr` ou demander les propriétés d'un produit `p`.

Un client `c` envoie une commande (Figure 9) d'une certaine quantité `q` d'un produit `p`.

4.2 Etape 1 : instances de schémas de problèmes

Les différents cas d'utilisation identifiés donnent lieu à

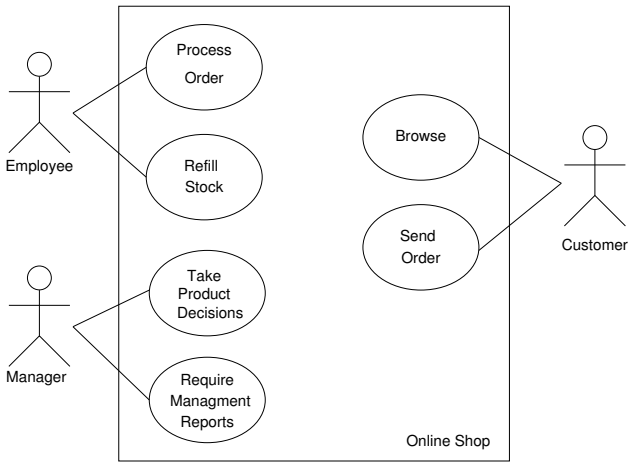

FIG. 7 – *Cas d'utilisation de la boutique en ligne*

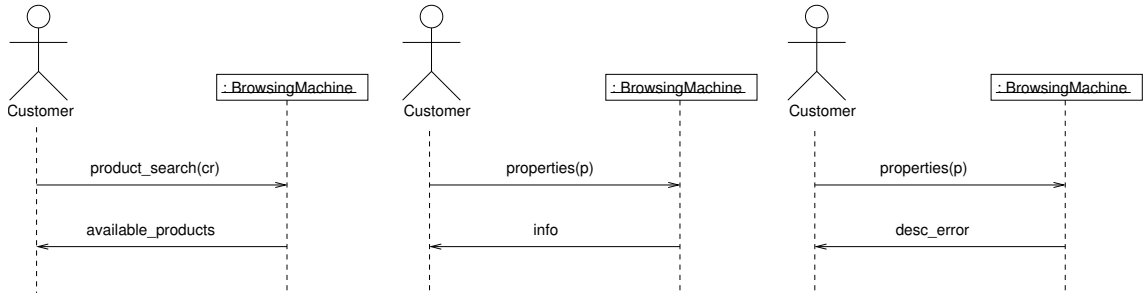

FIG. 8 – *Scénarios pour le cas d'utilisation Browse (recherche d'informations)*

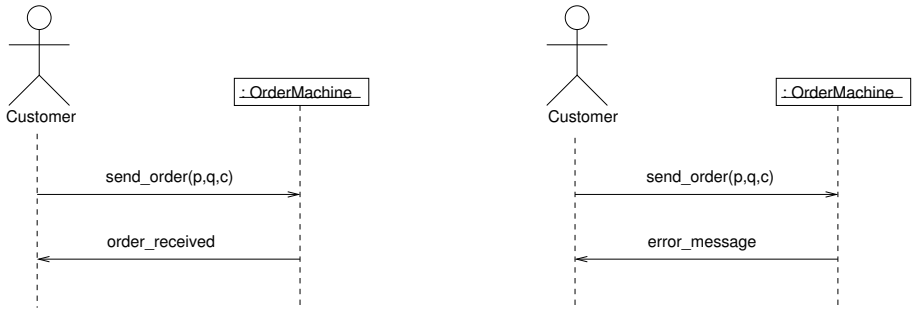

FIG. 9 – *Scénarios pour le cas d'utilisation SendOrder (envoi de commande)*

- des instances des schémas de mise à jour pour l'envoi de commandes ("Send Order"), le traitement des commandes ("Process order"), le réapprovisionnement de stock ("Refill stock"), et la prise de décisions concernant la commercialisation des produits ("Take Product Decisions")
- des instances des schémas de requête pour la recherche d'informations ("Browse"), et la demande de rapports de gestion ("Require Management Reports").

Nous donnons ci-dessous deux exemples, qui correspondent aux cas d'utilisation sur la recherche d'informations ("Browse") dans la Figure 10, et sur l'envoi de commandes ("Process order") dans la Figure 11.

La figure 10 est une instance du schéma de problème proposé pour les requêtes (Figure 5), où les interfaces `C` sont les requêtes du client `C!` pour rechercher des produits selon des critères

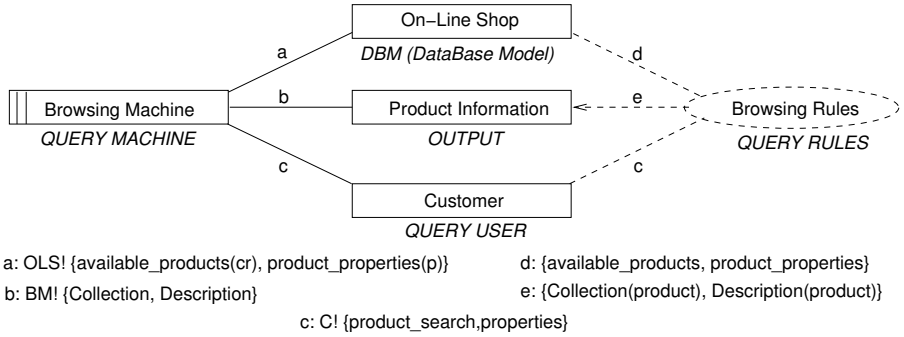

FIG. 10 – *Schéma de requête pour la recherche d'informations*

`cr` ou des caractéristiques d'un produit `p`, et l'interface `a` désigne les informations fournies par la boutique en ligne `OLS!`. La figure 11 est une instance du schéma de problème proposé pour

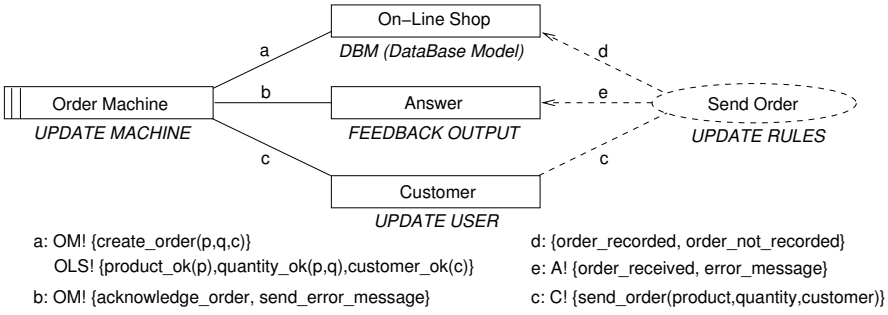

FIG. 11 – *Schéma de mise à jour pour l'envoi de commandes*

les mises à jour (Figure 4), où les interfaces `c` sont les envois de commandes du client `C!`, et l'interface `a` désigne la création d'une commande par la machine `OM!` et les vérifications fournies par la boutique en ligne `OLS!` sur le produit `product_ok(p)`, etc.

4.3 Etape 2 : spécification en Z

Pour cette étape, nous spécifions formellement la boutique en ligne en Z, c'est-à-dire que (selon l'étape 2 de notre méthode de développement Section 3.2) nous spécifions les différents domaines (et leurs opérations) et la machine à réaliser (ici pour les cas d'utilisation *Browse* et *SendOrder*). Nous avons choisi ce langage car l'état de la boutique en ligne est modifié par les commandes des clients et d'autres actions.

4.3.1 Définitions Auxiliaires

Les types de base suivants sont nécessaires :

$[ProductCode, CustomerCode, OrderCode, ProductDescription]$

Pour le type *ProductDescription* un élément erreur est nécessaire (cf. Figure 8).

$| \; desc\_error : ProductDescription$

La fonction *max_quantity* enregistre, pour un produit donné, la quantité disponible à la vente à un moment donné. Par exemple, notre boutique en ligne ne pourrait pas fournir 2 millions d'exemplaires du livre "Problem Frames" de Michael Jackson. Cette fonction est partielle car de nouveaux produits peuvent apparaître sur le marché dont la quantité n'est pas encore connue.

| $max\_quantity : ProductCode \pfun \mathbb{N}$

Notre boutique en ligne maintient à jour l'information relative aux produits, aux clients et aux commandes. A cet effet, un code unique est attribué à chacun. Cette contrainte d'unicité est exprimée par l'invariant d'état du schéma *OnLineShop* (Section 4.3.2).

Product

$pcode : ProductCode$
$quantity : \mathbb{N}$
$descr : ProductDescription$

$descr \neq desc\_error$

Customer

$ccode : CustomerCode$

Order

$ocode : OrderCode$
$product : ProductCode$
$quantity : \mathbb{N}$
$cus : CustomerCode$

4.3.2 Définition de l'état pour le domaine *OnlineShop*

L'état de la boutique en ligne est défini par le schéma suivant (dont le nom est légèrement différent). L'invariant exprime l'unicité des codes ainsi que les contraintes que, pour chaque produit en vente, la fonction *max_quantity* est définie et que pour chaque commande, il existe un client et un produit enregistré dans la base de données.

OnLineShop

$products : \mathbb{P}\, Product$
$customers : \mathbb{P}\, Customer$
$orders : \mathbb{P}\, Order$

$\forall p_1, p_2 : products \mid p_1 \neq p_2 \bullet p_1.pcode \neq p_2.pcode$
$\forall c_1, c_2 : customers \mid c_1 \neq c_2 \bullet c_1.ccode \neq c_2.ccode$
$\forall o_1, o_2 : orders \mid o_1 \neq o_2 \bullet o_1.ocode \neq o_2.ocode$
$\forall p : products \bullet p.pcode \in \mathsf{dom}\, max\_quantity$
$\forall o : orders \bullet (\exists c : customers \bullet o.cus = c.ccode) \land (\exists p : products \bullet o.product = p.pcode)$

4.3.3 Opérations de base du domaine *OnlineShop*

Les opérations de base correspondent aux interfaces du domaine *OnlineShop* dans les diagrammes de schémas de problèmes instanciés (Section 4.2).

Les prédicats *ProductOK*, *CustomerOK*, et *QuantityOK* (cf. interface `a` Figure 11) ci-dessous sont utilisés dans d'autres opérations, c'est pourquoi ils prennnent en compte un changement d'état possible (Δ*OnLineShop* au lieu de Ξ*OnLineShop*) mais ne spécifient pas comment est effectué ce changement d'état.

ProductOK

$\Delta OnLineShop$
$pc? : ProductCode$

$\exists p : products \bullet$
$\quad p.pcode = pc?$

CustomerOK

$\Delta OnLineShop$
$cc? : CustomerCode$

$\exists c : customers \bullet$
$\quad c.ccode = cc?$

QuantityOK

$\Delta OnLineShop$
$pc? : ProductCode$
$quant? : \mathbb{N}$

ProductOK
$quant? \leq max\_quantity\, pc?$

L'opération suivante fait partie du cas d'utilisation *SendOrder*. C'est une opération partielle qui est définie seulement quand les prédicats définis plus haut sont vrais.

CreateOrder

$\Delta OnLineShop$
$pc? : ProductCode$
$quant? : \mathbb{N}$
$cc? : CustomerCode$

$ProductOK$
$CustomerOK$
$QuantityOK$
$\exists o : Order \mid (\forall oo : orders \bullet oo.ocode \neq o.ocode) \land o.product = pc? \land$
$o.quantity = quant? \land o.cus = cc?$
$\bullet orders' = orders \cup \{o\}$
$products' = products$
$customers' = customers$

4.3.4 Spécifications des domaines *Answer* and *Customer*

Le domaine *Answer* contient deux éléments.
$Answer ::= order\_received \mid error\_message$

Les domaines "biddable" tels que les clients sont spécifiés via les ordres qu'ils peuvent donner.

$OrderCustomer ::= order\langle\langle ProductCode \times \mathbb{N} \times CustomerCode\rangle\rangle$

4.3.5 Spécification du domaine *OrderMachine* et prise en compte du cas d'utilisation *SendOrder*

Les opérations de la Section 4.3.3 sont utilisées par *OrderMachine* (cf. Figure 11) soit pour engendrer une nouvelle commande, soit pour transmettre un message d'erreur, l'état de la boutique en ligne demeurant inchangé.

SendSuccess

$answ! : Answer$

$answ! = order\_received$

SendError

$answ! : Answer$

$answ! = error\_message$

$SendOrder \mathrel{\widehat=} (ProductOK \land CustomerOK \land QuantityOK \land CreateOrder \land SendSuccess)$
$\lor (\Xi OnLineShop \land SendError)$

4.3.6 Prise en compte du cas d'utilisation *Browse*

La prise en compte d'un nouveau cas d'utilisation portant sur le même système d'information conduit à spécifier des nouvelles opérations sur le domaine *OnlineShop*.

Un *BrowseCustomer* peut soit rechercher des produits selon certains critères [2], soit examiner la description d'un produit donné.

$BrowseCustomer ::= product\_search\langle\langle \mathbb{P}\, Product\rangle\rangle \mid properties\langle\langle ProductCode\rangle\rangle$

En conséquence, deux nouvelles opérations de base pour interroger sur l'état du système doivent être ajoutées à la définition du domaine *OnlineShop*.

2. En Z, les prédicats sont identifiés par leur extension, c'est-à-dire l'ensemble des éléments pour lesquels le prédicat est vrai. Donc un critère sur des produits est identifié par un ensemble de produits de type *Product*. $p \in cr?$ exprime que p satisfait le critère $cr?$.

ProductSearch

$\Xi OnLineShop$
$cr? : \mathbb{P}\, Product$
$available\_products! : \mathbb{P}\, ProductCode$

$available\_products! =$
$\quad \{p : products \mid p \in cr? \bullet p.pcode\}$

ProductProperties

$\Xi OnLineShop$
$pc? : ProductCode$
$info! : ProductDescription$

$\exists p : products \bullet p.pcode = pc? \wedge info! = p.descr$

Pour spécifier le domaine *BrowsingMachine*, un schéma d'erreur est nécessaire.

ProductNotPresent

$\Xi OnLineShop$
$pc? : ProductCode$
$info! : ProductDescription$

$\forall p : products \bullet p.pcode \neq pc?$
$info! = desc\_error$

L'interface de la machine de consultation comprend les opérations *ProductSearch* (cf. ci-dessus), et *Properties* qui est définie comme suit:

$Properties \mathrel{\widehat{=}} ProductProperties \vee ProductNotPresent$

4.4 Etape 3 : recomposition

La figure 12 montre l'architecture de composants qu'on obtient pour la boutique en ligne, en instanciant l'architecture donnée figure 6.

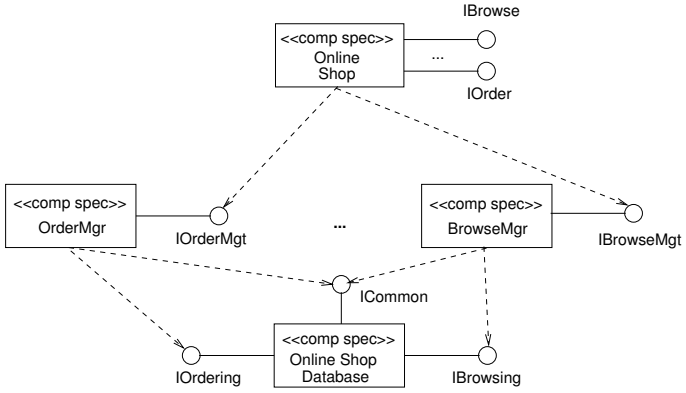

FIG. 12 – *Architecture des composants de la boutique en ligne*

Le composant *OnlineShopDatabase* est spécifié par le schéma *OnLineShop* donné dans le paragraphe 4.3.2 et les opérations définies sur ce schéma, à savoir *ProductOK*, *CustomerOK*, *QuantityOK*, *CreateOrder*, *AvailableProducts*, *ProductProperties*.

L'interface *ICommon* contient l'opération *ProductOK*, parce que cette opération est utilisée par les deux composants *OrderMgr* et *BrowseMgr*.

L'interface *IOrdering* contient les opérations *CustomerOK*, *QuantityOK* et *CreateOrder*. L'interface *IBrowsing* contient les opérations *AvailableProducts* et *ProductProperties*.

Le composant *OrderMgr* (qui correspond à *OrderMachine* de la Figure 11) met à la disposition de son environnement l'opération *SendOrder*, qui est définie dans le paragraphe 4.3.5. Cette opération utilise les opérations des interfaces *ICommon* et *IOrdering*.

Le composant *BrowseMgr* (qui correspond à *BrowsingMachine* de la Figure 10) met à la disposition de son environnement les opérations *ProductSearch* et *Properties*, qui sont définies dans le paragraphe 4.3.6. Ces opérations utilisent les opérations des interfaces *ICommon* et *IBrowsing*.

Les opérations des interfaces *IBrowse* et *IOrder* correspondent à celles des interfaces *IBrowsing* et *IOrdering*. La réalisation d'une opération des interfaces *IBrowse* et *IOrder* consiste juste en un appel de l'opération du même nom de l'interface *IBrowsing* ou *IOrdering*.

5 Conclusions et perspectives

Nous avons proposé une méthode pour la spécification et le développement de systèmes d'information. Cette méthode est basée sur différentes sortes de patrons. Les contributions de notre approche sont les suivantes :

- Nous avons donné des critères pour identifier les systèmes pour lesquels notre méthode est applicable (section 3.2). Pour achever cela, nous avons identifié une nouvelle sorte de domaines, à savoir les domaines d'affaires ("business domains"). Nos critères sont clairs et faciles à déterminer.
- Nous avons conçu deux nouveaux schémas de problèmes ("problem frames") qui sont bien adaptés aux systèmes d'information. Ces systèmes ne sont pas traités par les frames donnés par Jackson [Jac01]. Les systèmes d'information sont décomposés en plusieurs sous-systèmes, qui font soit des mises à jour, soit des requêtes. Cette décomposition se fait à partir des cas d'utilisation, ce qui est nouveau par rapport des travaux de Jackson.
- Ainsi, notre méthode établit un lien entre l'analyse orientée objet et les problem frames, qui n'existait pas jusqu'à maintenant.
- La méthode décrite ici intègre la méthode décrite dans [CR00, CH03a] où d'autres langages de spécification formelle étaient utilisés (CASL [BM04], CASL-LTL [RAC03], et LOTOS). Ici, une nouvelle expérience montre l'applicabilité de cette méthode à un autre langage, Z. [3] Chaque partie d'un schéma de problème instancié est spécifiée formellement. La méthode décrite dans [CH03a] fournit les conditions de validation de la spécification produite.
- Chaque sous-problème identifié est résolu relativement indépendamment des autres (en prenant en compte que la base de données est la même pour tous les sous-problèmes). Ce qui nous conduit au problème de la recomposition des solutions. Pour résoudre ce problème, nous proposons une architecture de composants qui est une instance du style d'architecture "repository".
- Dans son ensemble, notre méthode établit des liens entre l'analyse orientée objet, les problem frames, les spécifications formelles, les styles d'architecture et l'approche basée sur les composants. Jusqu'à présent, ces différents techniques ont été utilisées de manière plus ou moins isolée.
- Notre approche donne un guide méthodologique substantiel, qui conduit à développer une suite de documents reliés.

Dans l'avenir, nous avons l'intention de spécifier la communication entre les différents composants de l'architecture plus en détail, par example à l'aide de patrons de communication.

De plus, une telle méthode, qui vise à une utilisation guidée et intégrée de plusieurs techniques et plusieurs sortes de patrons, si elle est présentée ici pour le développement des systèmes d'information, doit trouver un champ d'application assez large en étant définie aussi pour les autres sortes de problèmes qui sont caractérisés par d'autres schémas de problèmes.

3. Notre propos ici n'est donc pas de privilégier l'usage d'un langage donné pour certains types d'application.

Références

[BCK98] Len Bass, Paul Clements, et Rick Kazman. *Software Architecture in Practice*. Addison-Wesley, 1998.

[BM04] Michel Bidoit et Peter D. Mosses. *CASL, The Common Algebraic Specification Language - User Manual*. Springer-Verlag, 2004. To appear. Available at `http://www.cofi.info/CASL_UserManual_DRAFT.pdf`.

[CD01] John Cheesman et John Daniels. *UML Components – A Simple Process for Specifying Component-Based Software*. Addison-Wesley, 2001.

[CH03a] Christine Choppy et Maritta Heisel. Systematic transition from problems to architectural designs. Rapport technique LIPN-2003-05, Université Paris XIII, France, 2003. 34 pages.

[CH03b] Christine Choppy et Maritta Heisel. Use of patterns in formal development: Systematic transition from problems to architectural designs. Dans M. Wirsing, R. Hennicker, et D. Pattinson, editeurs, *Recent Trends in Algebraic Development Techniques, 16th WADT, Selected Papers*, LNCS 2755, pages 205–220. Springer Verlag, 2003.

[CR00] Christine Choppy et Gianna Reggio. Using CASL to Specify the Requirements and the Design: A Problem Specific Approach. Dans D. Bert, C. Choppy, et P. D. Mosses, editeurs, *Recent Trends in Algebraic Development Techniques, 14th WADT, Selected Papers*, LNCS 1827, pages 104–123. Springer Verlag, 2000. A complete version is available at `ftp://ftp.disi.unige.it/person/ReggioG/ChoppyReggio99a.ps`.

[GS93] David Garlan et Mary Shaw. An introduction to software architecture. Dans V. Ambriola et G. Tortora, editeurs, *Advances in Software Engineering and Knowledge Engineering*, volume 1. World Scientific Publishing Company, 1993.

[Jac01] Michael Jackson. *Problem Frames. Analyzing and structuring software development problems*. Addison-Wesley, 2001.

[JCJO92] I. Jacobson, M. Christerson, P. Jonnson, et G. Overgaard. *Object-Oriented Software Engineering: A Use-Case Driven Approach*. Addison-Wesley, 1992.

[Mic02] Microsoft Corporation. *COM*$^+$, 2002. http://www.microsoft.com/com/tech/COMPlus.asp.

[Obj02] The Object Management Group (OMG). *Corba Component Model, v3.0*, 2002. http://omg.org/technology/documents/formal/components.htm.

[RAC03] Gianna Reggio, Egidio Astesiano, et Christine Choppy. CASL-LTL: A CASL extension for dynamic reactive systems – version 1.0 – summary. Rapport technique DISI-TR-03-36, Università di Genova, Italy, 2003.

[Spi92] J. M. Spivey. *The Z Notation – A Reference Manual*. Prentice Hall, 2nd édition, 1992.

[Sun97] Sun Microsystems. *JavaBeans Specification, Version 1.01*, 1997. http://java.sun.com/products/javabeans/docs/spec.html.

[Sun01] Sun Microsystems. *Enterprise JavaBeans Specification, Version 2.0*, 2001. http://java.sun.com/products/ejb/docs.html.

[Szy99] Clemens Szyperski. *Component Software - Beyond object oriented programming*. Addison Wesley, 1999.

[UML] UML Revision Task Force. *OMG UML Specification*. `http://www.uml.org`.