Chapitre IV : patrons de conception et modèles à objets ouverts pour le RàPC

Résumé

En suivant les deux niveaux de spécificité et les trois axes de variabilité retenus, nous détaillons les modèles à objets permettant de lier les points d'ouverture de notre plate-forme au sein d'une architecture abstraite. Notre démarche originale dans le cadre du RàPC tient principalement dans l'explication systématique de notre modélisation en termes de *patrons de conception*. Un patron de conception définit une modélisation typique répondant à un problème récurrent dans la conception de logiciels à objets. Les patrons permettent d'anticiper des problèmes d'ouverture et facilitent la compréhension des décisions de conception. Pour la gestion du raisonnement, nous proposons l'assemblage de composants gérant chacune des phases (recherche, réutilisation, révision et apprentissage). Chaque phase est configurée par la spécialisation d'un schéma générique à sélectionner. Pour la représentation des cas, nous permettons une structuration des indices. Enfin, pour l'organisation de la mémoire, nous permettons d'une part la structuration de la base de cas, et d'autre part la structuration des index. De plus, nous avons défini un cadre général et extensible pour la représentation des mesures de similarité. La représentation de l'indexation par situations comportementales vient enrichir la modélisation de base avec des objets spécifiques. La modélisation obtenue, justifiée en termes de patrons, est ainsi réutilisable au niveau de la conception de systèmes de RàPC. Cette modélisation a également mené à la réalisation concrète de la plate-forme CBR*Tools programmée dans le langage à objets Java. Cet outil permet alors la spécialisation et la capitalisation d'objets-métier notamment sous les formes d'index, de fonctions de similarité et de schémas de phase de raisonnement. Nous donnons enfin une estimation *a priori* de la complexité d'utilisation de CBR*Tools (effort minimum et dépendances systématiques).

Dans le chapitre précèdent, nous avons introduit notre approche originale proposant la conception d'une plate-forme à objets pour le RàPC et plus particulièrement pour l'indexation par situations comportementales. Nous avons également introduit les points d'ouverture nécessaires à une telle approche. Nous présentons alors dans ce chapitre les modèles à objets qui explicitent l'architecture abstraite de la plate-forme et la conception de chaque point d'ouverture. Notre démarche de conception est centrée sur l'utilisation de patrons de conception (*design pattern*) qui permettent de réutiliser des décisions typiques de conception pour obtenir l'ouverture désirée.

Dans les première et deuxième parties, nous détaillons la modélisation proposée respectivement, pour le cycle du RàPC de manière générale, et pour l'indexation par situations comportementales. Nous expliquons et argumentons l'utilisation des patrons en suivant les trois axes de variabilité retenus : gestion du raisonnement, représentation des cas et organisation de la mémoire. Chaque axe se décline alors suivant différents modèles à objets pour réaliser les points d'ouverture explicités précédemment. Nous nous concentrons plus particulièrement sur la représentation des cas et l'organisation de la mémoire qui sont essentielles pour notre modèle d'indexation. Pour la gestion du raisonnement, nous nous limitons à définir un cadre cohérent permettant d'intégrer les composants de notre modèle d'indexation dans un système complet de RàPC. Les diagrammes présentant notre modélisation respectent la notation UML (UML, 1997 ; Muller, 1997) dont les éléments de base sont précisés dans l'annexe A. Les patrons utilisés sont plus amplement décrits dans l'annexe B. Enfin, dans la troisième partie, nous décrivons la réalisation de cette plate-forme qui est alors nommée CBR*Tools. Un exemple simple de l'utilisation de cet outil est donné dans l'annexe C. Le chapitre suivant présente notre système Broadway dont la réalisation repose sur CBR*Tools.

1 Modèles à objets ouverts pour le RàPC

Nous présentons tout d'abord l'utilisation des patrons de conception pour la modélisation du cycle du raisonnement et des concepts généraux au RàPC suivant les trois axes de variabilité.

1.1 Gestion du raisonnement

Le raisonnement à partir de cas est généralement décomposé en quatre phases : recherche, réutilisation, révision et apprentissage. Nous montrons, dans cette section, comment sont organisés les 11 points d'ouverture liés à ce cycle (cf. chapitre III, §2.3.1.1) : contrôle du raisonnement, phases de raisonnement, résultat des phases, état du raisonnement et création des objets nécessaires au raisonnement. Nous introduisons tout d'abord notre modélisation par délégation des phases de raisonnement avec le patron *Stratégie*. Un raisonnement est alors construit par assemblage. Puis, pour la réalisation de chaque phase, nous proposons la définition d'une bibliothèque de *schémas de phase.* Chaque schéma peut être spécialisé pour réaliser une phase particulière. L'ouverture de cette modélisation repose notamment sur les patrons de conception *Stratégie*, *Façade*, *Fabrique abstraite*, *Patron de méthode* et *Chaîne de responsabilité* (Gamma *et al.,* 1995).

1.1.1 Patrons de conception et représentation du cycle du raisonnement

Pour la gestion du cycle du raisonnement, nous détaillons tout d'abord le mécanisme de contrôle qui permet la coordination des phases. Nous donnons alors l'exemple du protocole standard de

raisonnement. Puis, nous proposons des mécanismes permettant d'expliciter et de vérifier les contraintes syntaxiques et sémantiques garantissant la cohérence du raisonnement.

1.1.1.1 *Contrôle du raisonnement*

Nous proposons de déléguer l'exécution et le contrôle de chaque phase du raisonnement à un objet différent en appliquant quatre fois le patron de conception *Stratégie*. Ce patron permet la définition d'une famille d'algorithmes pour chaque phase du raisonnement, parmi lesquels il est possible de sélectionner celui à utiliser. Cette modélisation permet la réalisation des quatre points d'ouverture importants pour le raisonnement dont les classes de base sont Retrieve, Reuse, Revise et Retain. Chacune de ces classes définit une interface abstraite pour les phases du raisonnement et différentes implantations peuvent être réalisées. Le raisonnement est alors construit par assemblage des différents algorithmes pour chaque phase. Le contrôleur du raisonnement (classe Reasoner) est alors la façade du système de raisonnement à partir de cas (patron de conception *Façade*) et définit donc les points d'entrée (cf. Figure IV-1). Il permet de lancer un raisonnement étant donné un cas cible et donne l'accès à la mémoire. De plus, il constitue le point d'ouverture par lequel la gestion globale du raisonnement peut être spécialisée.

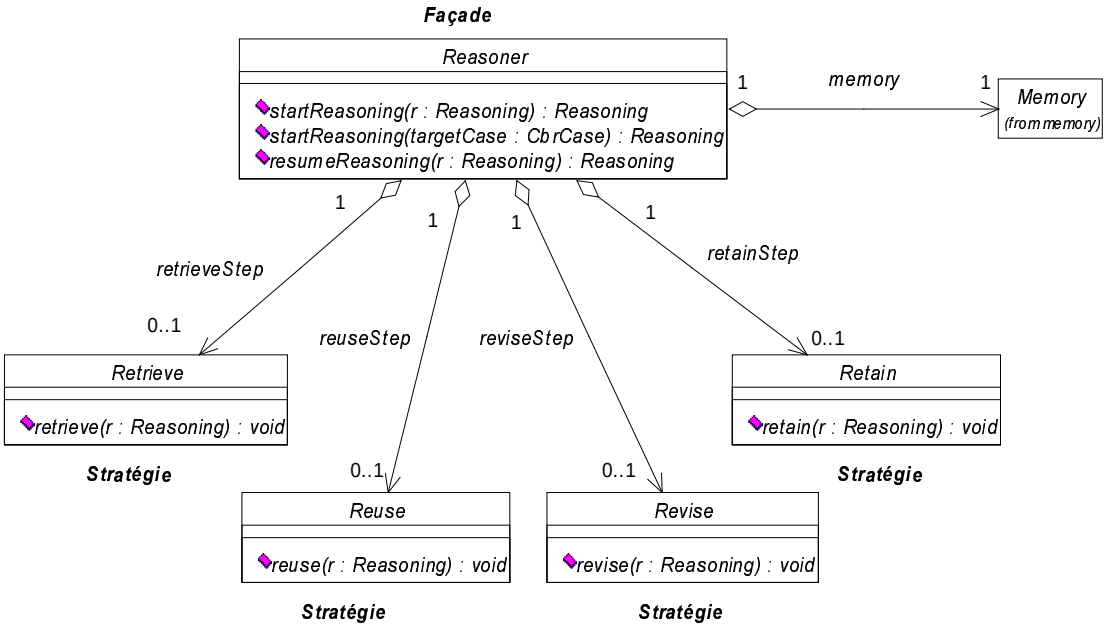

Figure IV-1 : représentation du contrôleur de raisonnement

Dans le raisonnement à partir de cas, chaque phase possède des paramètres spécifiques (nombre maximum de cas à rechercher par exemple) et opère sur les résultats issus des phases de raisonnement déjà exécutées. Par exemple, la phase de réutilisation utilise les résultats de la recherche, et la phase d'apprentissage peut utiliser les résultats des autres phases pour mettre à jour l'indexation de cas. Nous proposons alors de réunir dans un objet (point d'ouverture de classe Reasoning) l'ensemble des données liées au déroulement du raisonnement. En ce sens, nous suivons une des règles de conception donnée dans (Johnson & Foote, 1988) pour réduire le nombre de paramètres d'une méthode en créant un objet regroupant ces paramètres. Cet objet est alors modifié par chaque étape pour refléter l'avancement du raisonnement (traces du raisonnement), c'est pourquoi il est composé d'objets donnant les résultats de chaque phase (points d'ouverture de classes respectives RetrieveResult, ReuseResult, ReviseResult et RetainResult, cf. Figure IV-2). L'utilisation de cet objet présente

également l'avantage de pouvoir sauvegarder l'état d'un raisonnement qui peut alors être suspendu puis repris. De plus, cette modélisation permet d'analyser le raisonnement après son exécution.

Cette modélisation est apparentée à une architecture qui est souvent utilisée dans les systèmes multi-agents (Haton *et al.*, 1991) sous la forme d'un tableau noir (*blackboard*). Le modèle de tableau noir a été également présenté sous la forme d'un patron de conception (Bushmann *et al.*, 1996, pages 71–95). L'architecture typique à base de tableau noir est composée d'un ensemble d'agents appelés *sources de connaissances*, du tableau noir proprement dit et d'un module de contrôle. Les sources de connaissances possèdent une certaine compétence et ont une vue réduite sur l'état de la solution en cours d'élaboration dans le tableau noir. Le tableau noir est une sorte de structure de données qui permet de mémoriser l'état de la solution résultant de l'application des différentes sources de connaissances. Le module de contrôle supervise le déclenchement des différentes sources de connaissances. Son rôle est d'identifier les sources de connaissances qui peuvent s'appliquer, d'en choisir une et de l'exécuter. Toutefois, ce module peut être absent et dans ce cas, les sources de connaissance sont autonomes et réagissent aux modifications des données auxquelles elles ont accès dans le tableau noir. Dans notre modélisation, les sources de connaissances sont les phases du raisonnement, le tableau noir est représenté par l'objet sauvegardant l'état du raisonnement et le module de contrôle (obligatoirement présent) correspond au contrôleur du raisonnement. Ainsi, nous reprenons les avantages reconnus de ce type d'architecture (modularité et évolutivité), tout en faisant apparaître des concepts spécifiques au RàPC.

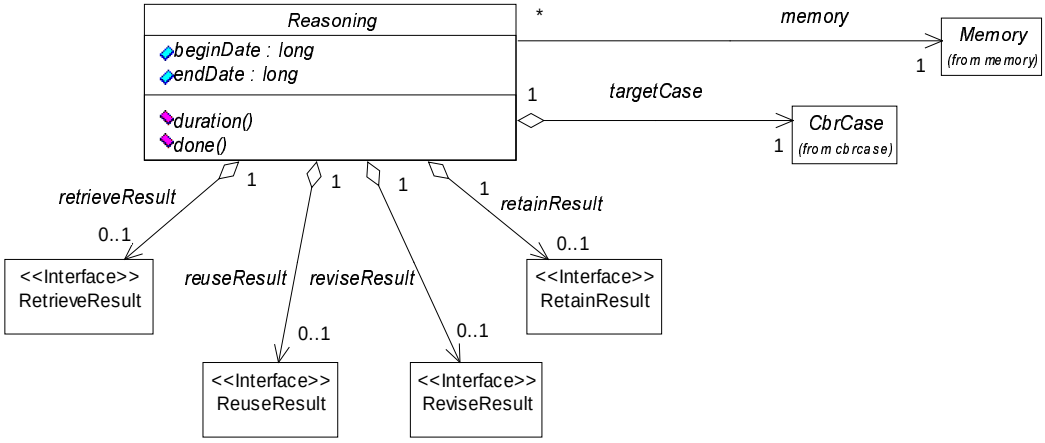

Figure IV-2 : représentation de l'état d'un raisonnement

A titre d'exemple, nous décrivons le protocole standard de raisonnement qui est sélectionné par défaut dans la plate-forme. Ce protocole est implanté dans la classe concrète SequentialReasoner qui spécialise la classe Reasoner (cf. Figure IV-3). Lors de l'appel d'un raisonnement, le contrôleur commence par initialiser le raisonnement (utilisation de la fabrique, cf. §1.1.1.2.2), puis appelle successivement les phases de recherche, de réutilisation, de révision et d'apprentissage. Le raisonnement peut être stoppé, soit sur l'initiative d'une des phases, soit en lançant le raisonnement avec un paramètre indiquant la phase d'arrêt. De plus, aucune des phases n'est obligatoire et les phases non définies ne sont pas exécutées.

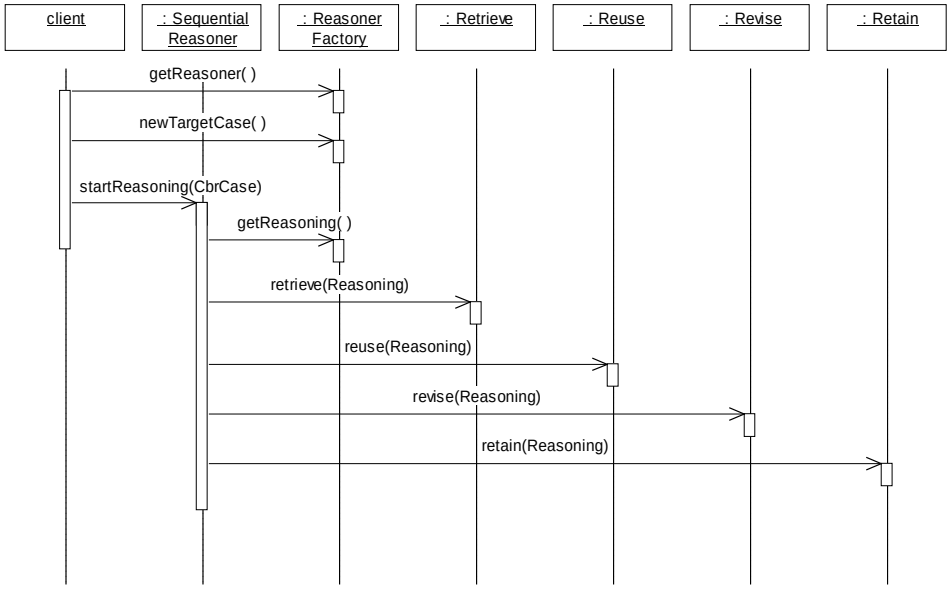

Figure IV-3 : protocole standard de raisonnement

1.1.1.2 *Cohérence de l'assemblage des phases*

Grâce à l'utilisation du patron de conception *Stratégie* pour les phases du raisonnement et la centralisation de leur interaction dans un objet sauvegardant l'état du raisonnement, nous permettons la réutilisation par assemblage de différentes stratégies de recherche, de réutilisation, de révision et d'apprentissage. Toutefois, cet assemblage ne peut être effectué aveuglément. Par exemple, si des hypothèses émises par une phase de réutilisation ne sont pas satisfaites par les résultats effectifs de la phase de recherche, le raisonnement est alors impossible. Il s'agit donc d'assurer la *compatibilité* entre les phases. Dans ce but, nous avons défini les notions de profils d'entrée et de sortie, et nous utilisons le patron *Fabrique abstraite* pour centraliser la création des objets nécessaires au raisonnement.

1.1.1.2.1 Profils d'entrée et de sortie

Nous proposons d'associer à chaque phase un *profil d'entrée* et un *profil de sortie*. Un profil d'entrée pose des contraintes sur le contenu des résultats des phases qui lui sont utiles et un profil de sortie définit les propriétés des résultats produits par une phase. Pratiquement, un profil est constitué par un ensemble d'interfaces de manipulation de données. Ces interfaces sont organisées de manière hiérarchique (cf. Figure IV-4). Nous avons particulièrement étudié les interfaces suivantes, nécessaires pour décrire les types des résultats de la phase de recherche :

- ensemble de cas (interface RetrieveResult),
- singleton (interface SingleCaseRetrieveResult),
- association d'un degré d'utilité à chaque cas retrouvé (interface UsefulnessRetrieveResult),
- association de la similarité sur chaque indice pour les cas retrouvés (interface DetailedSimRetrieveResult).

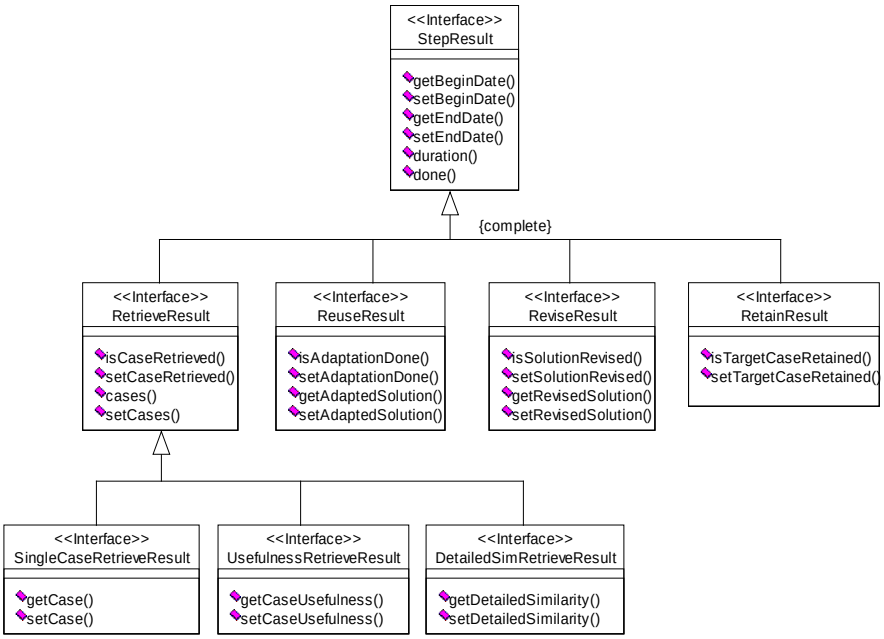

Figure IV-4 : interfaces de base pour les résultats des phases de raisonnement

Ainsi, les phases d'un raisonnement pourront être assemblées si et seulement si, pour chaque phase, l'union des profils de sortie des autres phases vérifie son profil d'entrée. L'union de deux profils est un profil, et un profil P vérifie un profil Q, si et seulement si, il existe dans P pour chaque interface de Q une interface identique ou plus spécifique. Cette relation, notée vérifie, s'écrit alors de la façon suivante :

Profil : {Interface}

vérifie : Profil, Profil $\rightarrow$ Booléen

{ P vérifie Q $\Leftrightarrow \forall$ i $\in$ Q, $\exists$ j $\in$ P tel que i=j ou j plus spécifique que i }

	Assemblage A	*Assemblage B*
Profils de recherche	entrée : {} sortie : {SingleCaseRetrieveResult}	entrée : {} sortie : {DetailedSimRetrieveResult}
Profils de réutilisation	entrée : {RetrieveResult} sortie : {ReuseResult}	entrée : {UsefulnessRetrieveResult} sortie : {ReuseResult}
Profils de révision	entrée : {ReuseResult} sortie : {ReviseResult}	entrée : {} sortie : {}
Profils d'apprentissage	entrée : {ReuseResult, ReviseResult} sortie : {RetainResult}	entrée : {} sortie : {RetainResult}
Contrôle de l'assemblage	*compatible*	*incompatible*

Tableau IV-1 : exemples de contrôles d'assemblages

Cette modélisation présente trois avantages importants. Premièrement, la hiérarchie d'interfaces est facilement extensible et permet la prise en compte de nouveaux types de résultats sans être liée à l'ordre d'exécution des phases. Deuxièmement, le contrôle de l'assemblage peut être effectué dès la conception, les profils jouant le rôle d'une documentation associée aux phases. Le contrôle peut

également être effectué lors de l'assemblage des objets[39], et nous évitons ainsi de retarder le contrôle jusqu'à l'exécution. Enfin, l'utilisation concrète de ces profils repose sur la définition d'une classe héritant des interfaces nécessaires pour chaque phase. Cette classe d'objets peut permettre d'assembler des phases *a priori* incompatibles en encapsulant des opérations de conversion (patron *Adaptateur*). Le couplage direct entre les phases est ainsi minimisé pour une meilleure réutilisabilité.

Le Tableau IV-1 donne l'exemple de deux assemblages : les phases de l'assemblage *A* sont compatibles contrairement à celles de l'assemblage *B*. En effet, dans l'assemblage *B*, la phase de recherche retourne la similarité détaillée de chaque cas retrouvé, mais ne donne pas de facteur d'utilité nécessaire à la phase de réutilisation. La modélisation proposée consiste à assembler ces deux phases en utilisant un objet de résultats réalisant les deux interfaces mises en cause (UsefulnessRetrieveResult et DetailedSimRetrieveResult). L'héritage multiple d'interfaces est alors une solution élégante pour réaliser le patron de conception *Adaptateur* (cf. Figure IV-5). Cet objet encapsule alors le calcul du facteur d'utilité de chaque cas à partir de la similarité détaillée.

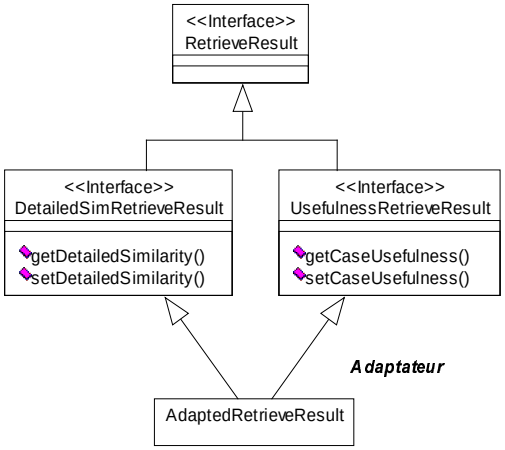

Figure IV-5 : adaptation d'interfaces pour l'assemblage

1.1.1.2.2 Gestion centralisée de la création des objets

L'utilisation des profils garantit la cohérence syntaxique de l'assemblage mais ne garantit pas la compatibilité sémantique de l'ensemble des objets intervenant durant un raisonnement. Cette compatibilité est difficile à vérifier automatiquement dans la mesure où elle repose principalement sur les connaissances d'un domaine : il serait coûteux de représenter ses propriétés pour pouvoir gérer leur cohérence au niveau de l'assemblage. Nous laissons donc la responsabilité du choix des objets à assembler au concepteur, mais nous offrons un modèle créateur permettant la centralisation de la gestion de l'assemblage. Dans ce but, nous utilisons le patron de conception *Fabrique abstraite* et nous réunissons dans une classe (ReasonerFactory) les méthodes de création des objets participant à cet assemblage (cf. Figure IV-6) : contrôleur de raisonnement, phases de recherche, objet de raisonnement, mémoire, cas cible. La centralisation des méthodes de création renforce la notion de famille d'objets dont la cohérence doit être assurée. Plusieurs fabriques concrètes peuvent alors être définies. Cette modélisation facilite également la substitution d'un type d'objet par un autre sachant qu'à chaque fois qu'un objet important doit être créé, il faudra s'adresser à la fabrique. Cette substitution est simplement réalisée par la dérivation d'une fabrique concrète existante et par la redéfinition d'une méthode de création.

[39] Dans CBR*Tools, les profils sont pour l'instant uniquement utilisés comme sources de documentation.

En reprenant l'exemple de la section précédente (cf. Figure IV-5), la réalisation de l'assemblage entraîne la spécialisation de la fabrique pour faire appel à la création des objets gérant la phase de recherche (méthode newRetrieveStep), la phase de réutilisation (méthode newReuseStep) et le résultat de la recherche (méthode newRetrieveResult).

De plus, lors de la création d'un contrôleur de raisonnement (méthode getReasoner, cf. Figure IV-6), l'objet de fabrique est automatiquement affecté comme composant du contrôleur. Le contrôleur joue donc pleinement son rôle de façade en donnant à ses clients l'accès à la fabrique. Ces derniers pourront ainsi créer les cas cibles compatibles avec le système de raisonnement construit. Le contrôleur lui-même utilise la fabrique pour créer, à chaque raisonnement, l'objet d'état du raisonnement dont la classe de base est Reasoning. Le système peut alors assurer que les objets utilisés dans le raisonnement sont issus de la famille définie par le concepteur dans la fabrique.

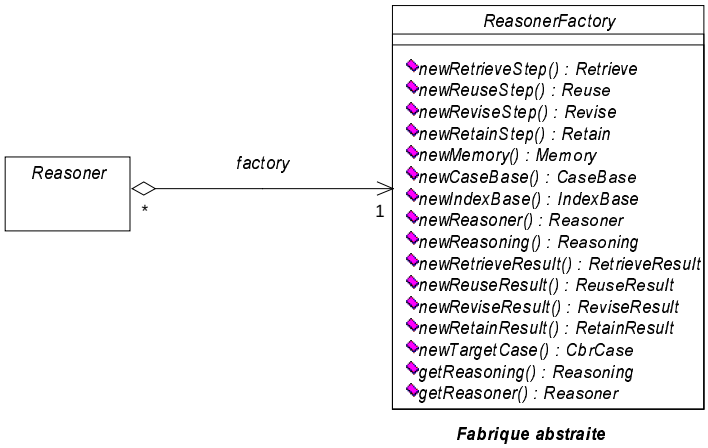

Figure IV-6 : la fabrique comme composant du contrôleur du raisonnement

1.1.2 Schémas de base pour les phases du raisonnement

Suivant notre modèle, le contrôleur du raisonnement délègue l'exécution de chacune des phases à un de ses composants. Ainsi, des objets indépendants contrôlent à leur tour les phases de recherche, de réutilisation, de révision et d'apprentissage. Afin de prendre en compte la diversité des techniques employées pour réaliser ces différentes phases, nous proposons une approche pragmatique basée sur la description de *schémas*. Un schéma est une solution typique pour la réalisation d'une phase du raisonnement adaptée à un cadre bien défini. Chaque schéma peut être spécialisé pour une application concrète, puis assemblé à d'autres pour former un raisonnement complet. Un schéma comprend un ensemble de classes abstraites et de composants concrets ; ainsi un schéma est semblable à une plate-forme à objets de très petite taille.

Nous donnons, à titre d'exemple, les schémas modélisés et implantés dans CBR*Tools pour les phases de recherche, de réutilisation et d'apprentissage. Ces schémas reposent tous sur une classe d'objets définissant un protocole de contrôle du schéma suivant la patron de conception *Patron de méthode*. Ce protocole se décompose en quatre étapes au minimum : initialisation, opérations spécifiques, mise à jour des résultats de la phase en accord avec le profil de sortie et finalisation. L'utilisation de ces schémas se fait principalement par spécialisation du protocole. Les étapes d'initialisation et de finalisation sont utilisées comme des démons pouvant être spécialisés pour déclencher des actions de propagation (pour l'interface utilisateur par exemple). L'étape de mise à jour des résultats peut également être spécialisée pour formuler les résultats suivant un profil de sortie requis. Cette modélisation permet ainsi d'adapter finement les étapes d'un schéma à une utilisation spécifique.

1.1.2.1 *Schéma de configuration pour la recherche*

La phase de recherche s'appuie sur les index qui organisent les cas suivant leur indices pour retrouver les cas les plus utiles étant donné un cas cible. Plusieurs index de haut niveau, formant des stratégies de recherche différentes, peuvent coexister dans CBR*Tools (cf. §1.3.3). Lors de la phase de recherche, il s'agit principalement de sélectionner et de configurer la stratégie à adopter. Le schéma que nous proposons pour la recherche permet de contrôler et de lancer la stratégie retenue. Le protocole de ce schéma est défini dans la classe BasicRetrievePattern et les paramètres de configuration d'une stratégie sont donnés dans un objet de classe GlobalIndexParams (cf. Figure IV-7). Ces paramètres peuvent être clonés suivant le patron *Prototype*. Cette modélisation présente donc l'avantage de permettre la modification dynamique des paramètres par défaut et la constitution de plusieurs configurations types.

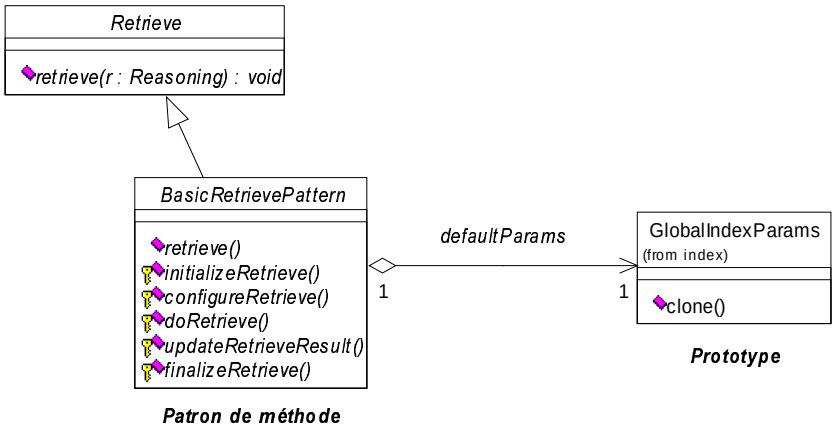

Figure IV-7 : classes d'objets du schéma de base pour la phase de recherche

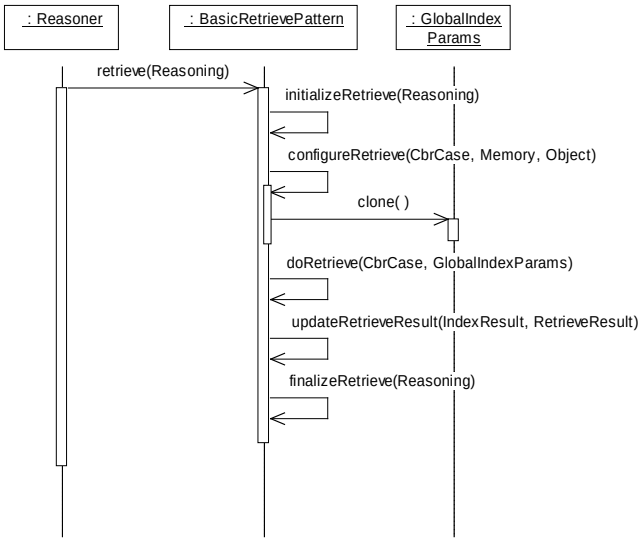

Figure IV-8 : protocole de configuration et d'exécution de la recherche

Le protocole de recherche repose alors sur trois étapes principales de contrôle qui sont exécutées en séquence (cf. Figure IV-8) :

1. *configuration* (méthode configureRetrieve) : clone les paramètres par défaut et les modifie au besoin,
2. *recherche* (méthode doRetrieve) : lance la recherche avec la stratégie retenue,
3. *mise à jour des résultats* (méthode updateRetrieveResult) : transforme les résultats de la stratégie de recherche (de type IndexResult) en résultats de la phase de recherche (de type RetrieveResult).

Le profil d'entrée de ce schéma est vide et celui de sortie est à définir par l'utilisateur qui doit obligatoirement spécialiser l'étape de mise à jour des résultats suivant le profil désiré.

1.1.2.2 *Deux schémas pour la réutilisation*

Pour la phase de réutilisation, nous proposons dans cette section deux schémas : le premier permet une adaptation basée sur l'analyse des indices du meilleur cas retrouvé et le deuxième permet de réaliser une adaptation des solutions proposées par un ensemble de cas. Nous donnerons également un troisième schéma de réutilisation issu de notre application Broadway (cf. chapitre V, §2.2.1).

1.1.2.2.1 Schéma d'adaptation d'un cas par analyse des indices

Le premier schéma permet de réaliser une adaptation similaire à celle du système REMIND (cf. chapitre III, §1.1.1) où la solution du cas cible est calculée en prenant en compte les différences entre les indices du cas cible et ceux du cas source. Par exemple, nous pouvons grâce à ce schéma représenter la formule d'adaptation utilisée dans REMIND qui permet d'évaluer la valeur d'une maison en fonction de la valeur s_{valeur} d'une autre maison (cas source) et de l'indice donnant le nombre de chambres :

$$c_{valeur} = s_{valeur} + 10000\,(c_{chambres} - s_{chambres})$$

La Figure IV-9 présente le diagramme de classes participant à ce schéma : la classe IndiceAnalysisReusePattern définit le protocole qui délègue l'analyse des indices et le calcul de la solution à un objet de classe IndiceAnalysisAdpatationFormula (patron *Stratégie*). La modélisation proposée est flexible puisqu'un ensemble de formules peut être défini, tout en permettant le changement dynamique de la formule utilisée.

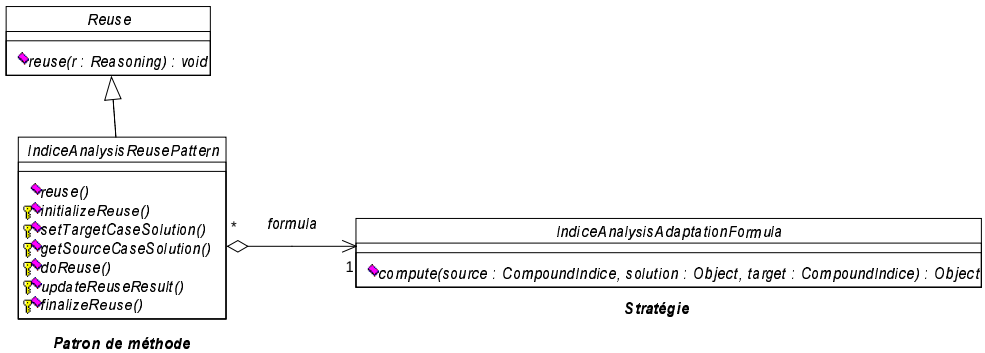

Figure IV-9 : schéma d'adaptation d'un cas par analyse des indices

Le protocole comprend deux étapes importantes (cf. Figure IV-10) :

1. *Calcul de la solution* (méthode doResuse) : la solution du cas source est récupérée (méthodes getSourceCaseSolution) et transmise à la formule d'analyse avec ses indices et les indices du cas cible. Cette formule retourne alors la solution qui est affectée au cas

cible (méthode setTargetCaseSolution). Les méthodes d'accès à la solution du cas doivent obligatoirement être définies par l'utilisateur suivant le type de cas considéré pour une meilleure encapsulation de sa structure.

2. *Mise à jour des résultats* (méthode updateReuseResult).

Les profils par défaut d'entrée et de sortie de ce schéma sont respectivement {SingleCaseRetrieveResult} et {ReuseResult}.

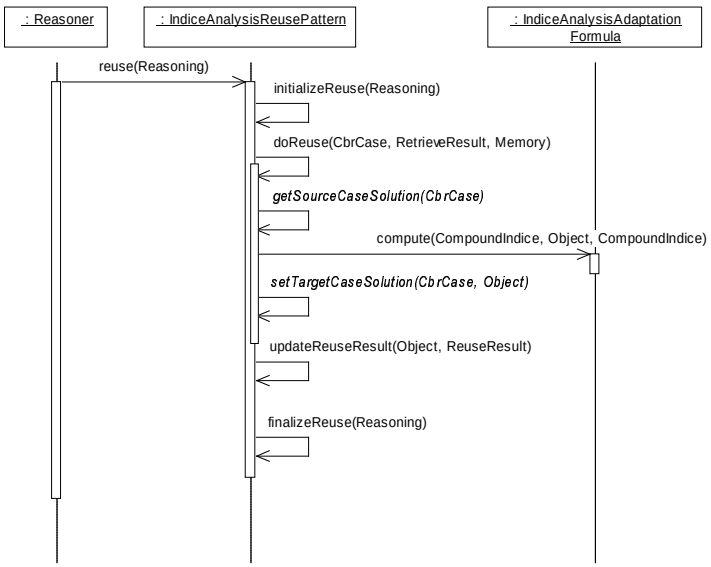

Figure IV-10 : protocole de réutilisation par analyse d'indices

1.1.2.2.2 Schéma d'adaptation de plusieurs cas par formule

L'objectif du deuxième schéma est de calculer la solution du cas cible en fonction des solutions proposées par un ensemble de cas sources retrouvés et de leur degré d'utilité. Ce type d'adaptation peut par exemple être utilisé pour la conduite de la nutrition des plantes. Dans notre approche (cf. chapitre II, §2.3), chaque cas retrouvé donne en effet comme solution la concentration de l'irrigation qui a été utilisée. A la suite de la phase de recherche qui a identifié un ensemble de cas sources, la solution peut être calculée en faisant une moyenne (ou tout autre formule) des solutions passées pondérées par les degrés d'utilité des cas sources associés (similarités des cas sources avec le cas cible courant).

La Figure IV-11 donne le diagramme de classes participant à ce schéma : la classe CaseSetAdaptationFormula définit le protocole qui délègue, en suivant les mêmes étapes que le schéma précédant, le calcul de la formule d'adaptation à un objet de classe CaseSetAdpatationFormula (patron *Stratégie*). Ce schéma permet donc de capitaliser différentes formules concrètes dont deux sont données en exemple : moyenne des solutions passées pondérées par leur degré d'utilité (classe MeanAdaptationFormula) et sélection de la solution présentant la meilleure utilité (classe BestAdaptationFormula).

Le profil par défaut d'entrée de ce schéma est donné par l'ensemble d'interfaces {UsefulnessRetrieveResult} alors que le profil de sortie est le singleton {ReuseResult}.

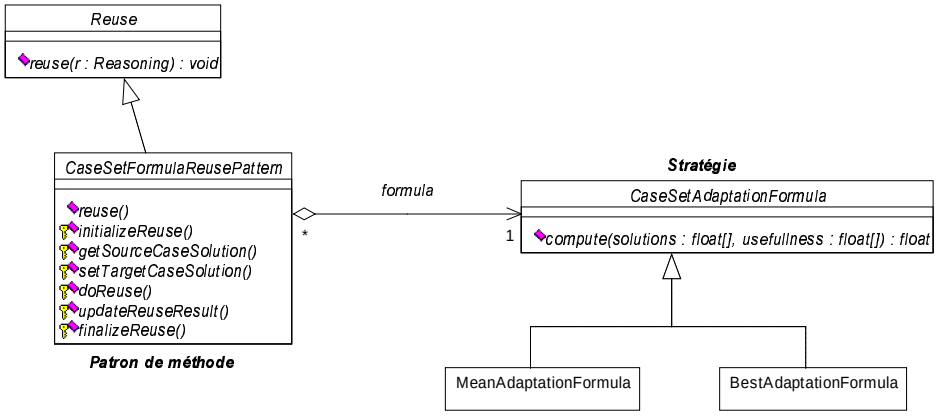

Figure IV-11 : schéma de réutilisation de plusieurs cas par formule

1.1.2.3 *Schéma de décision de l'ajout d'un cas pour l'apprentissage*

Dans le cadre du raisonnement à partir de cas, plusieurs types d'actions peuvent être effectués lors de l'apprentissage dont notamment la réorganisation générale de la mémoire avec modification de l'indexation, ou l'ajout et l'indexation du cas cible courant dans la mémoire. Nous proposons un schéma d'apprentissage permettant de simplifier la gestion de ce dernier type d'actions en appliquant le patron de conception *chaîne de responsabilité*. De manière similaire à la conception des schémas pour les autres phases (cf. Figure IV-12), ce schéma comprend un contrôleur de protocole (classe ResponsibilityChainRetainPattern) qui délègue la prise de décision de l'ajout d'un cas et de son indexation à un ensemble d'analyseurs (classe RetainAnalyser) organisés suivant une chaîne de responsabilité. L'intérêt de cette modélisation réside dans sa capacité à pouvoir prendre en compte de nouveaux types d'analyseurs tout en permettant leur combinaison.

Chaque analyseur étudie une caractéristique du cas cible ou de l'état du système de raisonnement pour décider s'il faut ou non ajouter le cas cible courant. Nous proposons trois analyseurs de base :

- un analyseur qui prend toujours la même décision, quel que soit le cas cible courant (classe ConstantRetainAnalyser),
- un analyseur vérifiant que la taille de la base de cas ne dépasse pas un certain seuil (classe CaseBaseSizeRetainAnalyser),
- un analyseur qui effectue une recherche dans la mémoire pour vérifier si un cas source existant n'est pas déjà trop similaire au cas cible courant (classe SimilarityRetainAnalyser).

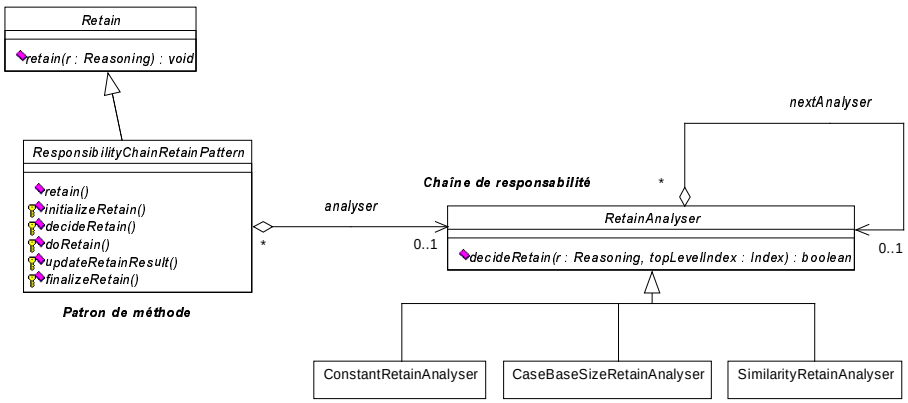

Figure IV-12 : schéma d'apprentissage par chaîne de responsabilité

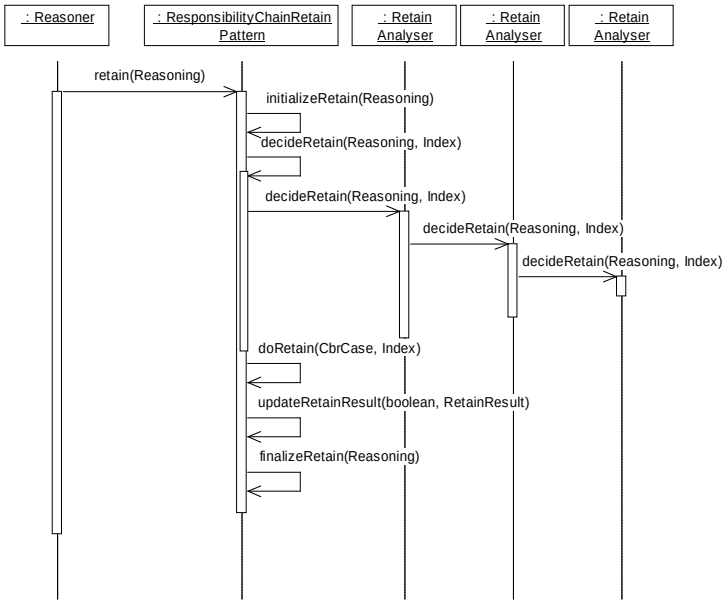

Figure IV-13 : protocole d'apprentissage par chaîne de responsabilité

Le protocole liant ces objets est donné dans la Figure IV-13, et repose sur trois étapes principales dont les deux premières sont exécutées pour chaque stratégie d'indexation définie dans le système de RàPC :

1. *Décision de l'ajout du cas* (méthode decideRetain) : la chaîne de responsabilité est parcourue. Au sein de cette chaîne, chaque analyseur a la possibilité, soit de passer la main à son successeur s'il est d'accord avec l'ajout du cas ou s'il ne sait pas décider, soit d'arrêter l'analyse en prenant une décision définitive.
2. *Ajout effectif du cas* (méthode doRetain) : si l'ajout a été décidé, il est alors réalisé.
3. *Mise à jour des résultats* (méthode updateRetainResult).

Le profil d'entrée par défaut de ce schéma est vide, et le profil de sortie est le singleton {RetainResult}.

1.2 Représentation des cas

Pour la représentation des cas, nous montrons la réalisation des quatre points d'ouverture correspondants (cf. chapitre III, §2.3.1.2) : tout d'abord la représentation des cas eux-mêmes, puis la représentation des indices, de leurs types et des vecteurs de poids. La modélisation proposée s'appuie en particulier sur le patron *Composite* pour structurer les indices.

1.2.1 Cas et situation

Dans notre modélisation, un cas est constitué de deux composants par défaut (cf. Figure IV-14) :

- un identificateur (classe Key) qui permet d'identifier un même cas dans différents systèmes, dans des fichiers d'échange et dans les dialogues avec l'utilisateur. La clef d'un cas peut prendre n'importe quelle forme (chaîne de caractères, entier ou autre) et peut être spécifiée en spécialisant la classe Key.
- une situation (classe CompoundIndice) qui donne un ensemble d'indices effectifs d'un cas parmi ces indices potentiels.

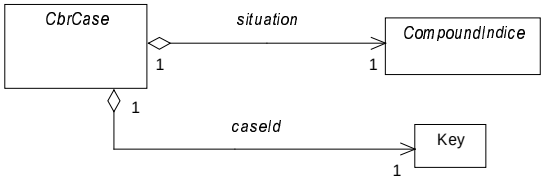

Figure IV-14 : cas et situation

Enfin, la classe CbrCase est un point d'ouverture représentant les cas, qui doit être spécialisé pour contenir les informations spécifiques à une application : informations de gestion (date de création, auteur), représentation du problème (en dehors des indices) et de la solution.

1.2.2 Indices structurés et poids d'importance

Pour la représentation des indices, nous appliquons le patron de conception *Composite* permettant de représenter de manière uniforme des objets simples ou composés (cf. Figure IV-15) : la classe Indice est l'abstraction des indices simples (classe SimpleIndice) et des indices composés (classe CompoundIndice). Un indice composé fait donc référence à un ensemble ordonné de sous-indices pouvant, à leur tour, être indifféremment des indices composés ou simples. Cette structuration permet de regrouper les indices suivant leur sémantique et sera utilisée dans la recherche, d'une part pour restreindre la mise en correspondance à un groupe d'indices, et d'autre part pour spécifier des mesures de similarité adaptées à chaque groupe.

L'expression d'un type d'indices composés correspond à la notion de *classe* d'un langage à objets. Il s'agit en effet de décrire les propriétés nécessaires au raisonnement pour chaque sous-indice : domaine de valeurs avec ou sans relation d'ordre, valeurs maximale et minimale si elles sont définies, convention de représentation des valeurs inconnues ou manquantes. Nous ne supposons pas la présence d'un métaniveau au sein du langage hôte utilisé pour l'implantation, qui nous permettrait de définir une métaclasse afin de d'ajouter les informations nécessaires au niveau de chaque type d'indices composés. Nous complétons alors la définition d'un type d'indices composés (classe CompoundIndice) par des informations supplémentaires (classes IndiceClassDescriptor et IndiceDescriptor). Ces informations sont partagées par les instances d'un même type d'indices composés et permettent principalement de connaître le type de chaque sous-indice (point d'ouverture IndiceType).

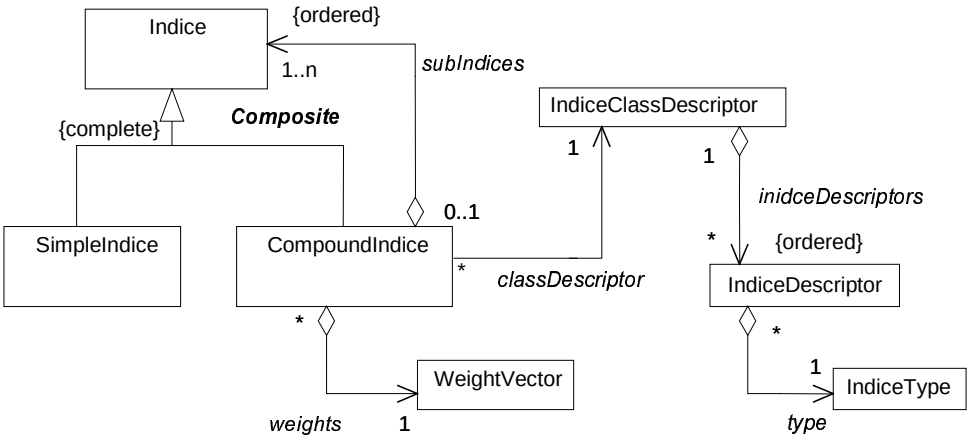

Figure IV-15 : indices structurés et types d'indices

A chaque indice composé est associé un vecteur de poids. Le poids d'importance de chaque indice n'a de sens que comparativement aux autres poids. C'est pourquoi un poids n'est pas un attribut d'un indice simple. De plus, la représentation d'un vecteur de poids comme un objet (point d'ouverture

WeightVector) permet la modification aisée de l'ensemble des poids d'un indice composé, l'encapsulation de la gestion de la cohérence des poids (normalisation des poids par exemple) et l'optimisation mémoire de la représentation. Nous avons défini une hiérarchie de vecteurs de poids (cf. Figure IV-16) : vecteurs donnant le même poids pour chaque sous-indice (classe ConstantWeightVector), vecteurs permettant d'associer à chaque sous-indice un poids différent (classe ArrayWeightVector), et des vecteurs calculant les poids de manière décroissante en fonction de la position du sous-indice dans la relation de composition (classe DecreasingFctWeightVector). Cette hiérarchie est extensible et d'autres types de vecteurs peuvent être définis. Nous avons également appliqué le patron de conception *Prototype*, ainsi les vecteurs de poids peuvent être clonés pour les affecter dans différents cas.

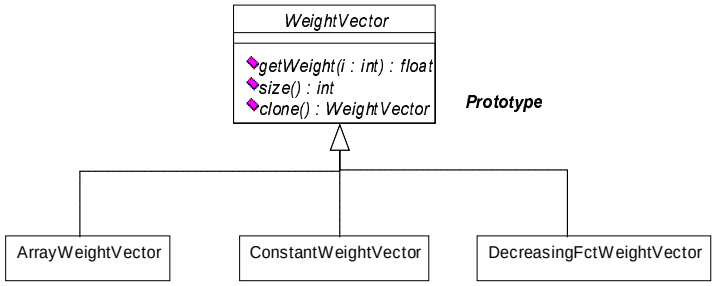

Figure IV-16 : hiérarchie de poids d'importance pour les indices

La Figure IV-17 donne un exemple de la représentation d'un ensemble structuré d'indices. Il s'agit d'une application chargée de calculer un facteur de risque pour un assureur automobile. Les indices d'un cas donnent les caractéristiques d'une voiture ayant un impact sur ce facteur. Dans cet exemple, nous avons séparé les attributs liés à la performance, au design et au prix. Nous montrons plus précisément les indices simples associés à la taille de la voiture (hauteur, largeur et longueur). Un vecteur de poids est associé à chaque indice composé.

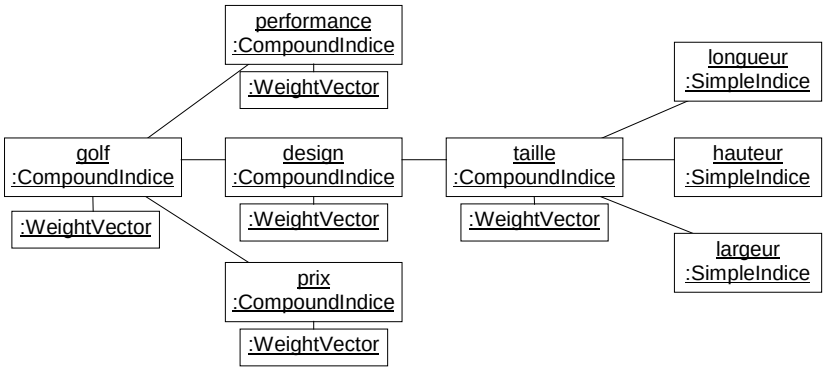

Figure IV-17 : extrait du diagramme d'objets représentant les indices d'un cas

1.3 Organisation de la mémoire

Dans cette section, nous précisons la conception des 11 points d'ouverture pour l'organisation de la mémoire que nous avons précédemment introduits (cf. chapitre III, §2.3.1.3). Plus précisément, pour l'organisation des cas, nous proposons l'utilisation de bases de cas structurées améliorant ainsi la manipulation des cas durant leur cycle de vie, quelle que soit leur localisation physique. Pour l'organisation de l'indexation, nous définissons le concept *d'index* encapsulant à la fois l'algorithme de recherche et la structure de données associée. Ces index sont également structurés et permettent ainsi d'utiliser finement les connaissances du domaine et les caractéristiques des indices pour

construire des stratégies de recherche adaptées. Enfin, nous proposons une modélisation à objets des mesures de similarité aisément modifiables et extensibles. Cette modélisation met en avant les patrons de conception *Façade*, *Composite*, *Stratégie* et *Prototype*.

1.3.1 Indépendance des index et du stockage des cas

Nous proposons de séparer la responsabilité du stockage (logique et physique) des cas et celle de la gestion de leur indexation (cf. Figure IV-18). La mémoire (classe Memory) est alors composée d'une base de cas (classe CaseBase) et d'une base d'index (classe IndexBase). Nous faisons ainsi apparaître trois points d'ouverture dans lesquels la mémoire est une façade (patron de conception *Façade*) gérant l'accès global aux cas. Une base de cas encapsule l'accès aux cas étant donné leur identificateur : elle permet de parcourir l'ensemble des cas, de connaître leur nombre et d'ajouter ou de retirer un cas de cet ensemble. La base d'index permet de réunir un ensemble d'index dont chacun permet l'accès aux identificateurs des cas indexés à partir de leurs indices.

Cette modélisation simplifie la maintenance de la base de cas et des index. En effet, elle permet d'effectuer des modifications et des évolutions tout en gardant le contrôle de la cohérence sous la responsabilité de la mémoire. Par exemple, l'ajout d'un cas peut être effectué directement dans une base de cas ou au niveau de la mémoire. Dans ce dernier cas, il sera automatiquement à la fois ajouté dans la base de cas et indexé dans l'index par défaut. De même, le retrait d'un cas au niveau de la mémoire implique son retrait de la base de cas et des index. D'autres fonctions de gestion sont également définies (cf. Figure IV-18) : reconstruction des index à partir des cas de la base (méthode buildFromCaseBase) et effacement des cas non indexés (méthode deleteNotIndexedCases). Cette gestion peut alors être modifiée ou enrichie en spécialisant le comportement de la mémoire.

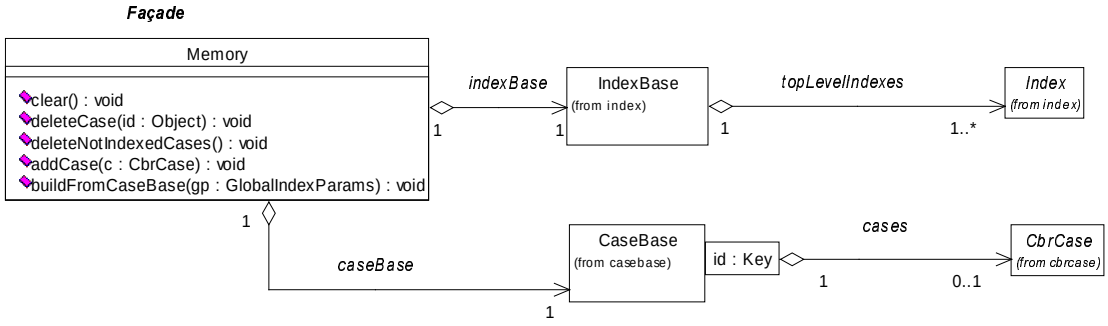

Figure IV-18 : délégation du stockage et de l'indexation

1.3.2 Bases de cas structurées : organisation logique et physique des cas

Dans notre modélisation, nous offrons un point d'ouverture pour l'organisation hiérarchique des bases de cas suivant, d'une part leurs caractéristiques logiques liées au cycle de vie des cas et, d'autre part leur localisation physique. Dans ce but nous appliquons le patron de conception *Composite* (cf. Figure IV-19) : la classe CaseBase est alors la classe de base d'une hiérarchie de classes séparant les bases de cas composées (classe CompoundCaseBase) et les bases de cas simples (classe SimpleCaseBase). Chaque base de cas peut être ouverte en deux modes (en lecture seule ou en modification) et possède des informations sur les données (classe CaseBaseMetadata). L'ajout d'un nouveau cas dans une base n'est correct que s'il est compatible (méthode isCaseCompatible). Ce critère est utilisé pour ajouter un nouveau cas par classification dans la hiérarchie formée par des bases de cas composées.

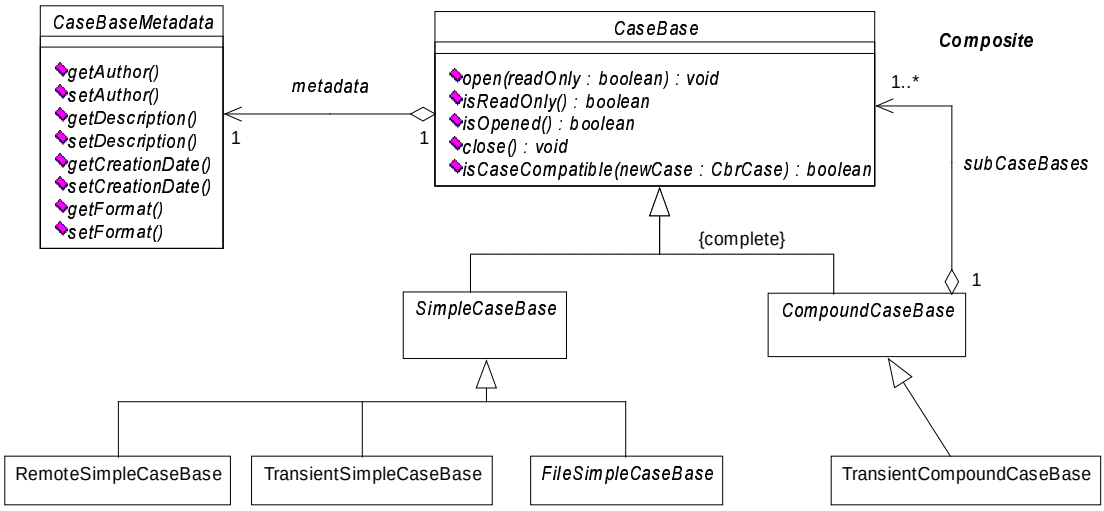

Figure IV-19 : hiérarchie de bases de cas

Plusieurs types de bases de cas simples peuvent alors être définis en spécialisant la classe SimpleCaseBase : bases de cas volatiles (classe TransientSimpleCaseBase), bases de cas avec sauvegarde sur un fichier (classe FileSimpleCaseBase), ou bases de cas distantes (classe RemoteSimpleCaseBase). Les bases de cas composées servent à regrouper les bases de cas suivant une sémantique liée à l'application. Les bases de cas doivent assurer l'unicité de l'affectation des identificateurs aux nouveaux cas. Dans ce but, nous fournissons des objets chargés du calcul des identificateurs (classe IdentifierGenerator). Soit les bases de cas utilisent des générateurs de type différent, l'unicité est alors garantie de par la conception des bases, soit plusieurs bases utilisent le même générateur. Un générateur est en fait une version objet de la notion de *séquence* du système de gestion de bases de données Oracle (Oracle, 1992, pp. 4-188–192).

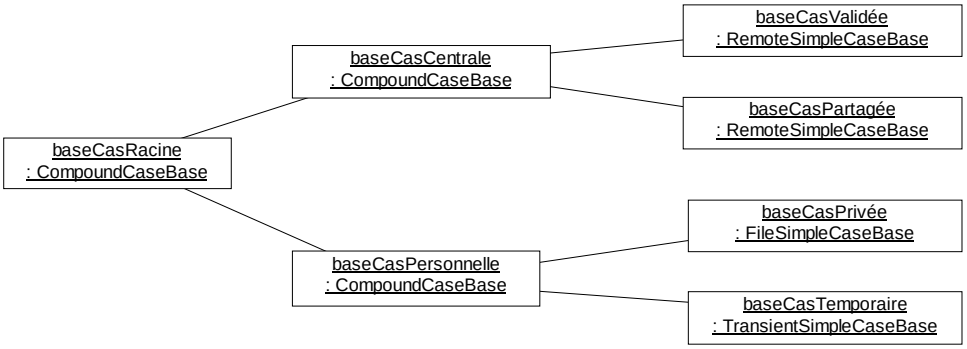

Figure IV-20 : exemple d'une structuration de bases de cas

La Figure IV-20 donne un exemple dans lequel deux principales bases de cas sont réunies : la base de cas personnelle de l'utilisateur et la base de cas centrale. La base de cas personnelle est à son tour décomposée en une base de cas temporaire et une base de cas privée. La base de cas temporaire contient les cas créés lors d'une session de raisonnement. La base de cas privée contient les cas dont l'utilisateur ne souhaite pas une utilisation générale. D'un autre côté, la base de cas centrale contient d'une part les cas publiés par les utilisateurs et n'ayant pas encore été validés (base de cas partagée), et d'autre part les cas acceptés et vérifiés par l'administrateur ou les experts de l'application (base de cas validée). Cet exemple met en évidence l'apport de cette structuration pour mieux suivre les cas dans leur cycle de vie et de validation, tout en montrant l'utilisation des différents types de bases de cas simples. Grâce au patron *Composite* cette structuration est transparente pour le raisonnement.

1.3.3 Index et stratégie de recherche

Un index est un objet qui encapsule une structure de données et une méthode de recherche d'un ensemble de cas sources suivant la situation du cas cible. Nous proposons tout d'abord l'association d'un filtre d'indices à un index permettant ainsi à chaque index de prendre en compte un sous-ensemble d'indices suivant une représentation adaptée. Nous définissons ensuite une hiérarchie extensible d'index mettant l'accent sur la possibilité de concevoir des index composites pour la réalisation de stratégies de recherche complexes. Enfin, nous étudions les conditions de composition permettant de garantir la cohérence de la construction d'une telle stratégie.

1.3.3.1 *Index et filtrage des indices*

Plus précisément, un index offre un ensemble d'opérations (indexation ou retrait d'un cas, reconstruction de sa structure interne) et permet notamment la recherche d'un ensemble de cas suivant une situation (méthode searchCases). Un index offre également un accès direct à la liste des cas qu'il indexe, il est ainsi possible de consulter les propriétés de cette relation (nombre de cas indexés, test d'appartenance).

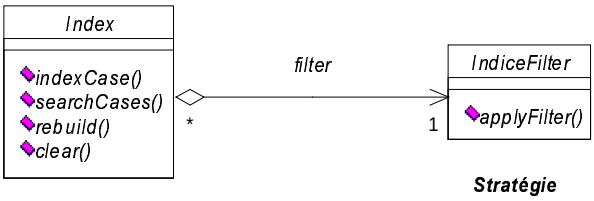

Figure IV-21 : index et filtre d'indices

Nous introduisons la notion de *filtre d'indices* (classe IndiceFilter) permettant de modifier l'ensemble des indices pris en compte à partir de la structure générale des situations (cf. Figure IV-21). Un filtre permet d'effectuer deux types d'opérations (qui peuvent être combinées) sur les indices :

- *sélection* d'une sous-partie,
- *transformation* de la représentation.

Par exemple, il est possible de concevoir un index n'utilisant que la sous-partie décrivant le design d'une voiture (indice composé design dans l'exemple de la Figure IV-17), au lieu de prendre l'ensemble des indices (design, prix et performance). Un index peut également nécessiter de regrouper les indices de manière différente (vecteur à plat par exemple) ou bien d'inclure des indices calculés à partir des indices initiaux. Un filtre permet la construction de cette représentation spécifique. Si cette transformation est coûteuse, elle devra être faite au niveau de la représentation initiale, ainsi la transformation sera réduite à une sélection. Un filtre est utilisé par l'index lors de la recherche ou de l'ajout de cas. Cette conception suit le patron *Stratégie*, ce qui permet une évolution indépendante des filtres et leur partage entre différents index.

1.3.3.2 *Hiérarchie extensible d'index*

Pour la construction de stratégies complexes de recherche, nous proposons une hiérarchie extensible d'index pouvant être composés (cf. Figure IV-22). Dans ce but, nous utilisons encore une fois le patron de conception *Composite* : la classe Index est la racine de la hiérarchie d'héritage comportant notamment les index simples (classe SimpleIndex) et les index composés (classe CompoundIndex). Grâce à cette modélisation, il est possible d'utiliser l'index adapté à un sous-espace des indices ou aux caractéristiques d'indices spécifiques (notamment les propriétés de leur domaine de valeurs), tout en les combinant pour former une organisation complète. Au sein de cette structure

hiérarchique, l'indexation d'un nouveau cas ou la recherche des cas sources sont effectuées suivant un parcours récursif dans lequel chaque index composite contrôle l'exécution.

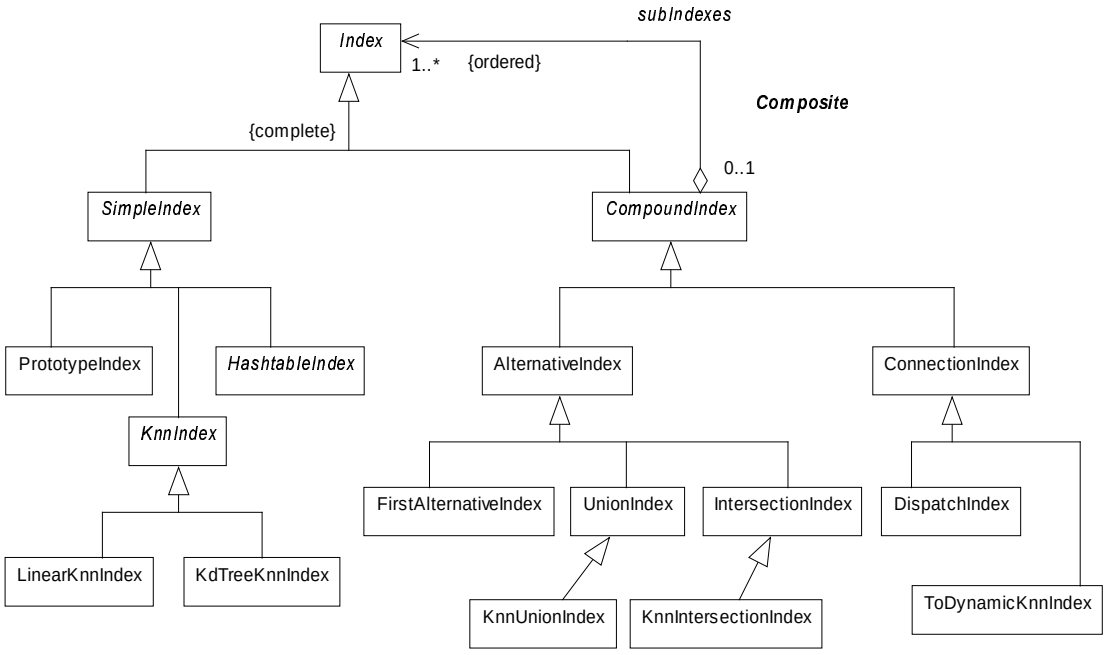

Figure IV-22 : hiérarchie extensible d'index

Les index simples permettent la manipulation d'une seule structure de données, utilisée par une méthode de recherche de cas. Par exemple, dans CBR*Tools, nous offrons quatre index simples :

- un index de prototypes (classe PrototypeIndex) : cet index repose sur la définition d'une hiérarchie de prototypes où chaque prototype[40] définit une fonction de compatibilité. Un prototype est alors relié à un ensemble de cas compatibles. La recherche s'effectue par une opération de classification dans cette hiérarchie, et les identificateurs des prototypes compatibles ainsi que les cas reliés à ces prototypes sont retournés. La représentation des prototypes et le protocole de classification peuvent être spécialisés. Cet index permet de réaliser des arbres discriminants construits *a priori* ou avec un algorithme d'induction. La compatibilité d'un cas avec un prototype peut être booléenne ou basée sur le calcul d'une similarité.
- un index à base d'une table de hachage (classe HashtableIndex) : cet index permet de retrouver un ensemble de cas à partir d'une clef calculée en prenant en compte un sous-ensemble d'indices.
- un index par plus proches voisins linéaire (*K-Nearest Neighbours*, classe LinearKnnIndex) : cet index parcourt un ensemble de cas sources, calcule les similarités entre les situations des cas sources et celle du cas cible. Finalement, les k meilleures situations sont retenues suivant une relation d'ordre définie sur le domaine de valeurs des similarités. Cet index est le plus souple dans son utilisation : il permet de gérer tout domaine de valeurs (y compris avec des valeurs manquantes) du moment qu'une similarité a été définie. Par contre, la complexité de cette recherche est linéaire en fonction du nombre de cas à parcourir, dans la mesure où le calcul des similarités et le nombre d'indices sont négligeables devant le nombre de cas.

[40] Au sens de l'outil REMIND (cf. chapitre III, §1.1.1).

- un index par plus proches voisins optimisé, utilisant des *k*-d trees (classe KdTreeKnnIndex) : cet index utilise les relations d'ordre définies sur les domaines de valeurs des indices pour construire un arbre partageant l'espace de recherche en sous-espaces. Le résultat est identique à une recherche linéaire par plus proches voisins, mais certains sous-espaces sont d'office éliminés d'après les propriétés de construction (Wess *et al.,* 1994). La recherche est alors plus efficace[41] (complexité moyenne en $O(\log_2 n)$, où n est le nombre de cas) mais nécessite la construction d'un arbre au préalable (complexité moyenne en $O(k\, n \log_2 n)$).

Nous pouvons identifier deux types d'index composés dont le comportement diffère lors de leur utilisation :

- les index *d'embranchement* (sous-classes de AlternativeIndex) qui permettent d'effectuer des recherches suivant un ensemble de sous-index et de traiter les résultats pour les réunir. Par exemple, nous proposons d'effectuer : l'union des résultats (classe UnionIndex), l'intersection des résultats (classe IntersectionIndex), ou la sélection du premier sous-index donnant des résultats de recherche satisfaisants (classe FirstAlternativeIndex).
- les index de *connexion* (sous-classes de ConnectionIndex) qui permettent d'utiliser deux index en séquence : les résultats d'un premier index sont utilisés comme paramètres pour le second. Par exemple, l'index de répartition (DispatchIndex) permet d'activer des sous-index suivant les résultats du premier index. L'index de connexion vers un index dynamique (ToDynamicKnnIndex) permet d'opérer une recherche uniquement en prenant en compte les cas identifiés par le premier index.

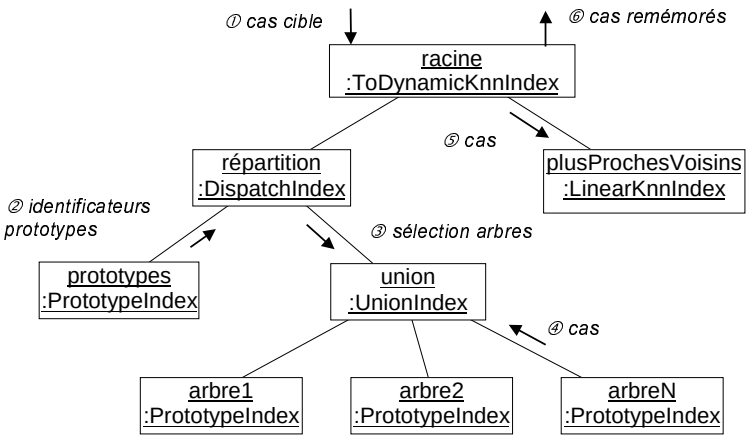

Figure IV-23 : exemple de composition d'index

La Figure IV-23 donne l'exemple d'un diagramme d'objets représentant une stratégie de recherche basée sur la combinaison des techniques d'indexation de REMIND (cf. chapitre III, §1.1.1) : prototypes puis arbres d'induction (ou plus généralement arbres discriminants) et plus proches voisins. Nous montrons ainsi que notre modélisation permet d'effectuer ce type de combinaison tout en apportant une flexibilité accrue : des étapes peuvent en effet être ajoutées ou supprimées suivant les besoins. Dans cette structure, la recherche des cas sources les plus utiles au raisonnement, pour un cas cible donné, est effectuée par un parcours récursif en profondeur d'abord. Pour commencer, un ensemble de prototypes compatibles avec le cas cible est identifié (index prototypes). Chaque prototype représente un ensemble de cas en compréhension dont l'identificateur est relié sémantiquement à un arbre discriminant. L'index prototypes retourne les identificateurs des prototypes compatibles et l'index union continue le parcours récursif en ne considérant que les arbres

[41] L'implantation de cet index dans CBR*Tools a été réalisée par M. Golinelli durant un stage sous notre encadrement (Golinelli, 1997). De nombreux tests comparatifs avec l'index par plus proches voisins linéaire ont été également réalisés.

discriminants ainsi sélectionnés. Chaque arbre permet d'identifier un sous-ensemble de cas (index arbre1 à arbreN). L'union de ces ensembles (index union) est alors passée en paramètre pour effectuer une recherche par plus proches voisins (index plusProchesVoisins). Finalement, les meilleurs cas suivant cette recherche sont retournés (index racine).

Un parcours similaire est effectué lors de l'indexation d'un nouveau cas. Le nouveau cas est tout d'abord classé dans les prototypes (index prototypes) puis ajouté dans les arbres discriminants associés aux prototypes (index arbre1 à arbreN). Si la structure interne des arbres discriminants n'est pas incrémentale, les index doivent être reconstruits pour que le cas soit pris en compte.

1.3.3.3 *Communication entre les index et cohérence de l'assemblage*

A partir des index introduits précédemment, nous avons identifié six types de résultats rendus par un index lors d'une recherche ou d'une indexation. Ces types de résultats sont organisés de manière hiérarchique (cf. Figure IV-24) :

- *Ensemble de cas* (interface CaseSetIndexResult).
- *Ensemble ordonné de cas suivant leur similarité* (interface CaseSimIndexResult) : une similarité est associée à chaque cas, l'ensemble des cas est alors une liste ordonnée suivant cette similarité.
- *Ensemble ordonné de cas suivant leur similarité, complété de leur similarité détaillée* (interface CaseDetailedSimIndexResult) : chaque cas possède une similarité détaillée au niveau de chacun de ses indices.
- *Sous-ensembles de cas* (interface CaseSubsetIndexResult) : un ensemble d'identificateurs de sous-ensembles de cas est défini (sous-ensembles de cas en compréhension).
- *Sous-ensembles en extension* (interface CaseSubsetExtensionIndexResult) : les sous-ensembles sont en extension, c'est-à-dire que chaque identificateur de sous-ensemble est lié à un ensemble en extension de cas.
- *Sous-ensembles évalués* (interface CaseSubsetSimIndexResult) : chaque sous-ensemble est lié à une similarité.

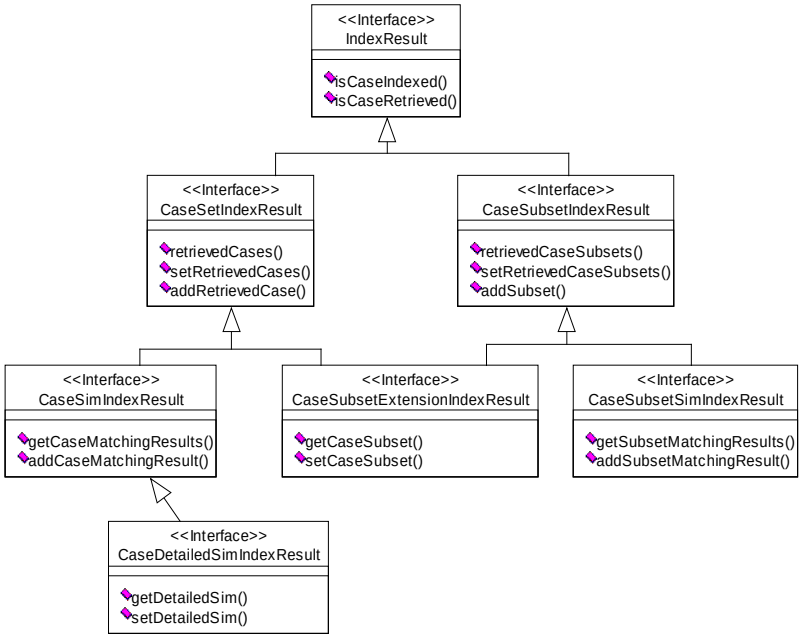

Figure IV-24 : hiérarchie d'interfaces des résultats d'un index (recherche et indexation)

Comme nous venons de le montrer, les index peuvent être composés pour former des structures complexes. Lors d'une recherche, chaque index est utilisé pour affiner progressivement la sélection des cas. Lors d'une indexation, chaque index met à jour sa structure en fonction des mises à jour faites par les autres index. Ainsi, au sein d'une stratégie complexe, un index utilise les résultats des index déjà parcourus. Nous proposons alors de passer en paramètres, lors de l'utilisation d'un index, deux types de résultats :

- le résultat courant (cf. Figure IV-25),
- les résultats intermédiaires qui sont sauvegardés dans une table.

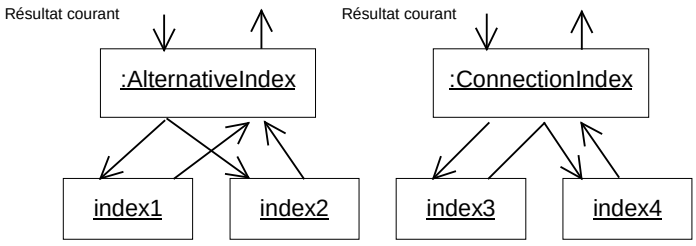

Figure IV-25 : transmission du résultat courant par les index composés

Pour assurer la cohérence d'un assemblage d'index, il s'agit de déterminer si chaque index dispose de l'accès aux résultats attendus. Nous utilisons la notion de *profil* que nous avons introduite pour l'assemblage des phases du raisonnement (cf. §1.1.1.2.1). Un profil est un ensemble d'interfaces issues de la hiérarchie dont la classe de base est IndexResult. Chaque index définit un *profil de paramètre* pour le résultat courant, ainsi que pour chaque entrée de la table des résultats intermédiaires dont il a besoin. De même, chaque index définit un *profil de résultat* indiquant le type de données retournées. Nous représentons alors un objet index avec ses profils de paramètre et de résultat en les plaçant respectivement en-dessous et au-dessus, comme l'exemple de la Figure IV-26. Le terme « pour » indique qu'il s'agit d'un profil de paramètre pour un résultat intermédiaire sauvegardé dans la table sous un nom particulier (« Index0 » dans l'exemple). Si ce terme est absent, le profil porte sur le résultat courant. Le terme optionnel « dans » indique que le résultat de l'index, respectant le profil de résultat mentionné, est sauvegardé dans la table en lui assignant un nom (« index1 » dans l'exemple).

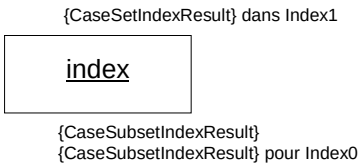

Figure IV-26 : indication des profils associés à un index

Les index simples définissent explicitement leurs profils qui peuvent toutefois être sélectionnés parmi plusieurs possibles suivant leur configuration. Les index composés transmettent le résultat courant et peuvent poser des contraintes sur celui-ci. Par exemple, pour les index d'embranchement, le profil de paramètre est, soit {CaseSubsetIndexResult}, soit le profil vide. Dans le premier cas, chaque identificateur de sous-ensemble de cas est associé à un des sous-index. Lors de la recherche ou de l'indexation, la présence d'un identificateur rendra actif le sous-index associé. Le profil de résultat est par défaut égal au profil du dernier sous-index. De plus, un index d'embranchement comme UnionIndex peut imposer un type de résultat pour ses sous-index de manière à pouvoir effectuer son opération (une union). De manière similaire, l'index de connexion ToDynamicKnnIndex impose que le profil de paramètre du second sous-index soit au minimum {CaseSetIndexResult}.

La vérification de la cohérence d'une stratégie doit se faire lors de la conception, les profils servant de moyens pour expliciter les contraintes. Ces profils sont alors utilisés pour configurer concrètement les index et indiquer s'il doivent sauvegarder leur résultat dans la table des résultats intermédiaires.

Dans la Figure IV-27, nous reprenons l'assemblage des index de la Figure IV-23 en montrant la configuration des profils des index. Cet assemblage respecte les contraintes énoncées. L'index prototypes ne suppose rien sur le résultat courant et retourne un résultat de type {caseSubSetIndexResult} qui vérifie le profil de paramètre de l'index union. Le résultat courant est alors transmis à chaque sous-index sélectionné parmi arbre1 à arbreN. Ces index ont un profil de paramètre vide et retourne un résultat de type {CaseSetIndexResult} qui vérifie le profil de résultat imposé par l'index union. A son tour, ce résultat vérifie le profil de paramètre de l'index plusProchesVoisins qui finalement retourne un résultat de type {caseSimIndexResult}.

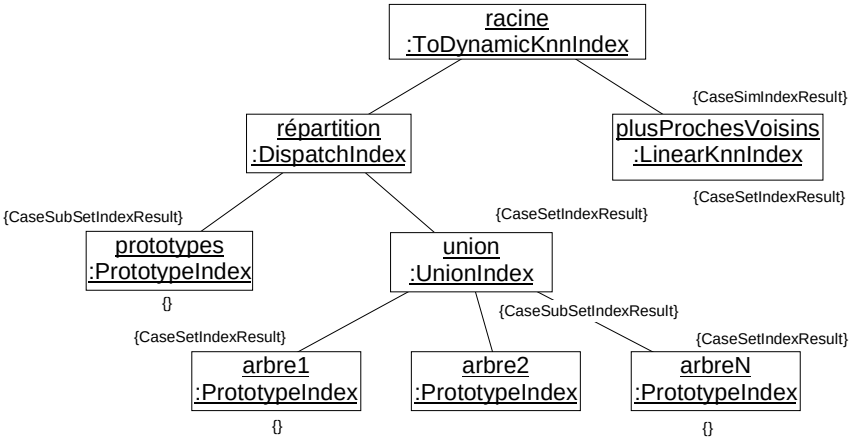

Figure IV-27 : exemple d'assemblage valide des index

1.3.4 Similarités structurées, modifiables et extensibles

La modélisation que nous proposons repose sur une représentation structurée des similarités et permet une modification aisée de leur définition.

1.3.4.1 *Domaines de valeurs des similarités*

Le résultat du calcul d'une mesure est un objet de classe CmpValue. L'encapsulation des valeurs de comparaison dans des objets permet de gérer différents types de résultats (point d'ouverture CmpValue). Par exemple, CBR*Tools offre trois types de valeurs (cf. Figure IV-28) :

- des booléens (classe CmpBoolValue),
- des *facteurs* de similarité donnant un réel élément de [0,1] (classe CmpFactorValue) comme dans la plupart des systèmes de RàPC,
- et des couples (possibilité, nécessité) (classe CmpPossibilityValue). Ce dernier domaine de valeurs est utilisé lorsqu'il est nécessaire de prendre en compte des indices dont les valeurs sont imprécises ou incertaines. Par exemple, ce domaine est utile dans le cadre de systèmes de RàPC en conception (Visser & Trousse, 1993 ; Trousse & Visser, 1993). Dans la suite du document, nous ne développerons pas l'utilisation de ce domaine, et le lecteur intéressé pourra se reporter à (Jaczynski & Trousse, 1994 ; Jaczynski, 1993).

D'autres types de valeurs peuvent être ajoutés par spécialisation de la classe CmpValue. Tout domaine de valeurs doit être muni d'une relation d'ordre permettant d'utiliser la similarité et de constituer des classements notamment dans un index par plus proches voisins. Nous avons défini une

hiérarchie extensible de relations d'ordre (point d'ouverture CmpValueOrder), permettant de définir d'une part des relations adaptées aux nouveaux domaines de valeurs éventuels, et d'autre part des relations différentes pour le même domaine de valeurs.

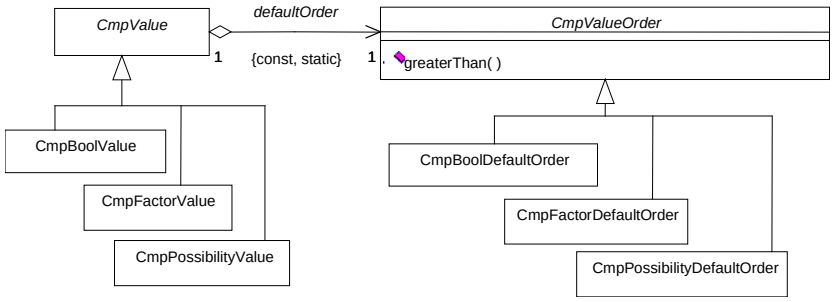

Figure IV-28 : valeurs des similarités et relations d'ordre

1.3.4.2 *Similarités structurées et modifiables*

La représentation des mesures de similarité doit être adaptée à la représentation des indices, c'est pourquoi nous utilisons de manière identique le patron de conception *Composite* nous permettant de définir des mesures structurées (cf. Figure IV-29). Une mesure (de classe Similarity) peut donc être une mesure de similarité adaptée à un indice simple (classe SimpleIndiceSimilarity) ou à un indice composé (classe CompoundIndiceSimilarity).

Nous appliquons également le patron de conception *Stratégie* pour la représentation des fonctions élémentaires de similarité et des fonctions d'agrégation. Ce patron permet la définition d'une famille de fonctions parmi lesquelles il est possible de sélectionner celle à utiliser. Ainsi, l'objet de similarité d'un indice simple délègue le calcul de la similarité à une fonction de similarité (classe ValueSimilarityFct). De même, l'objet de similarité d'un indice composé délègue le calcul de l'agrégation des valeurs de similarité issues de la comparaison des sous-indices, à une fonction d'agrégation (classe AggregationFct). Un vecteur de poids d'importance est également défini au niveau de chaque objet de similarité pour les indices composés. Enfin, les objets de similarité utilisent le patron de conception *Prototype* qui permet de créer une nouvelle similarité par clonage d'une similarité existante (méthode clone).

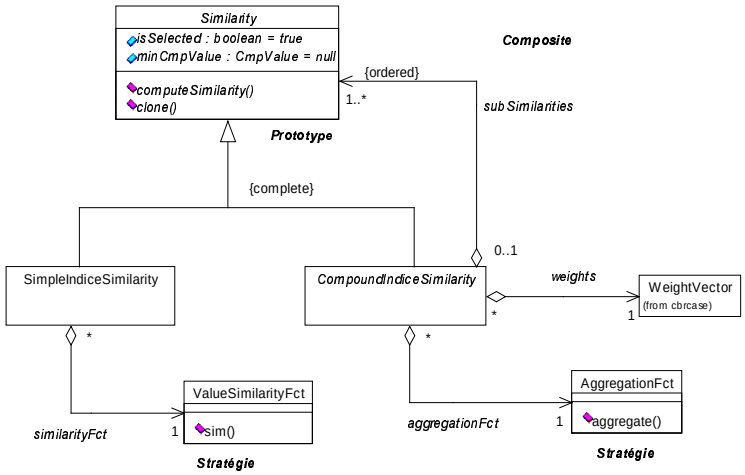

Figure IV-29 : mesures de similarité structurées et modifiables

Cette modélisation permet une modification aisée des mesures de similarité. Il est tout d'abord possible d'affecter dynamiquement de nouvelles fonctions élémentaires de similarité ou des fonctions

d'agrégation sans modifier la structure de la mesure de similarité. D'autre part, le clonage des similarités permet de gérer des mesures qui ne diffèrent que par les modifications apportées. Nous pouvons ainsi constituer de manière simple une bibliothèque de mesures de similarité. Cette modélisation présente également une bonne répartition des responsabilités : les objets formant la structure de la mesure de similarité contrôlent le calcul, alors que les fonctions sont utilisables dans différentes mesures et sont issues de hiérarchies de classes pouvant être étendues de manière indépendante. Le calcul est effectué par un parcours récursif de la structure : les mesures de similarité d'un indice simple contrôlent l'appel de la fonction élémentaire de similarité, et les mesures de similarité d'un indice composé contrôlent l'appel récursif aux sous-similarités ainsi que l'agrégation des résultats intermédiaires. Le calcul de la similarité suppose que les deux ensembles d'indices à comparer et la représentation de la similarité aient une structure identique. Si deux cas de structures différentes doivent être comparés, la similarité peut être construite dynamiquement pour refléter la structure commune aux deux cas.

Nous proposons la prise en compte de deux types de paramètres lors du calcul de la similarité : les *paramètres de construction* et les *paramètres d'évaluation* :

- Les paramètres de construction permettent de modifier finement le comportement de la similarité au niveau de chaque objet de la structure (attributs de la classe Similarity). Deux paramètres de construction sont proposés dans CBR*Tools permettant de fixer un seuil de similarité minimum au niveau de chaque indice (simple ou composé), et de sélectionner les indices qui participeront au calcul. Le premier paramètre est utile pour assurer un certain degré de similarité et permet d'anticiper l'échec d'une comparaison : si le seuil n'est pas respecté, le calcul de la similarité est de suite abandonné et les comparaisons restantes ne sont pas effectuées. Le deuxième paramètre permet d'utiliser une structuration de la similarité existante tout en sélectionnant les indices effectivement utilisés. Les paramètres de construction peuvent être modifiés et leur changement est persistant.
- Les paramètres d'évaluation sont des objets (de classe SimilarityEvalParams) transmis lors du calcul. Ces paramètres sont donc facilement modifiables à chaque appel et permettent notamment : l'activation des seuils de similarité, la sélection d'une méthode de choix des poids d'importance, et la définition de la valeur à utiliser lorsqu'un indice simple a une valeur inconnue. La méthode de sélection des poids est intéressante, puisque nous pouvons associer des poids d'importance au niveau des indices des cas sources, du cas cible et de la similarité. Les poids associés à chaque cas source indiquent quelles sont les importances relatives des indices permettant de juger de l'utilité du cas, et sont mis à jour lors de l'apprentissage. Les poids associés aux indices du cas cible proviennent d'une évaluation faite *a priori* lors de l'élaboration au début d'un raisonnement. Les poids issus de la similarité sont considérés comme des poids par défaut qui ont été estimés lors de la construction de la similarité. La sélection des poids est déléguée à un objet passé en paramètre, permettant ainsi de sélectionner ou de combiner les poids de manière ouverte. Par exemple, si l'utilisateur courant est un novice, le système utilisera les poids de la similarité ou celui des cas sources, par contre, si l'utilisateur est un expert, les poids du cas cible qu'il aura spécifiés seront utilisés. Enfin, il est intéressant de pouvoir modifier la valeur de comparaison dans le cas d'un indice de valeur inconnue. Par exemple, un facteur de 0.5 est habituellement utilisé ; cependant, des évaluations optimistes (valeur 1) ou pessimistes (valeur 0) peuvent être choisies.

1.3.4.3 *Hiérarchies extensibles de fonctions élémentaires de similarité et d'agrégation*

Le calcul des similarités s'appuie sur l'utilisation de deux types de fonctions : les fonctions élémentaires de similarité et les fonctions d'agrégation. Pour la représentation de ces fonctions, nous proposons deux hiérarchies extensibles de classes. La hiérarchie des fonctions élémentaires de similarité est donnée dans la Figure IV-30 et l'expression des fonctions est précisée dans Tableau IV-

2. Le premier critère de classification est constitué du domaine de valeurs de la fonction. Nous développons uniquement les sous-classes permettant de calculer des facteurs de similarité, et les sous-classes applicables pour n'importe quel type de domaine. Lors du calcul de la comparaison, le descripteur de l'indice considéré est passé en paramètre, ce qui permet à la méthode de calcul, d'une part, de vérifier que les propriétés nécessaires de l'indice sont bien respectées, et d'autre part, de récupérer les données descriptives de l'indice dont notamment son domaine de valeurs (par exemple domaine numérique borné, liste de valeurs).

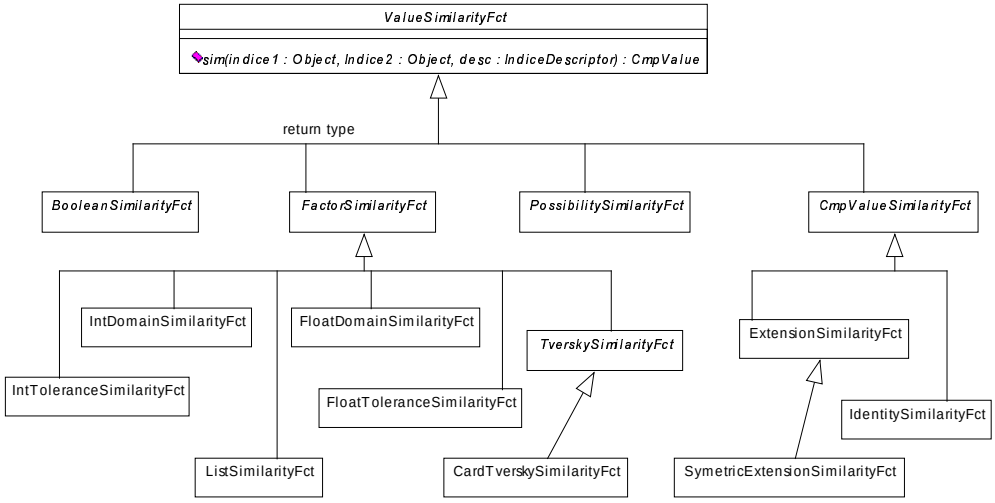

Figure IV-30 : extrait de la hiérarchie des fonctions élémentaires de similarité

Classe	*Fonction*	*Type de valeurs de l'attribut*
ExtensionSimilarityFct	$s(x,y) = M(x,y)$ M matrice de similarité	tout type de valeurs
SymetricExtensionSimilarityFct	$s(x,y) = M(x,y) = M(y,x)$ M matrice de similarité	tout type de valeurs
IdentitySimilarityFct	$s(x,y) = \begin{cases} v_1 \text{ si } x = y \\ v_2 \text{ sinon} \end{cases}$	tout type de valeurs
TverskySimilarityFct	$s(x,y) = \frac{f(x \cap y)}{f(x \cap y) + \alpha\ f(x-y) + \beta\ f(y-x)}$ $\alpha \geq 0, \beta \geq 0$	ensemble
CardTverskySimilarityFct	$s(x,y) = \frac{\text{Card}(x \cap y)}{\text{Card}(x \cap y) + \alpha\ \text{Card}(x-y) + \beta\ \text{Card}(y-x)}$ $\alpha \geq 0, \beta \geq 0$	ensemble
IntDomainSimilarityFct FloatDomainSimilarityFct	$s(x,y) = 1 - \frac{\lvert x-y \rvert}{D_{max} - D_{min}}$	valeurs numériques dont le domaine est borné par D_{max} et D_{min}
IntToleranceSimilarityFct FloatToleranceSimilarityFct	$s(x,y) = \begin{cases} 0 \text{ si } \lvert x-y \rvert \geq d \\ 1 - \frac{\lvert x-y \rvert}{d} \text{ sinon} \end{cases}$ $d \geq 0$, est appelé la tolérance	valeurs numériques
ListSimilarityFct	$s(x,y) = 1 - \frac{\lvert pos(x) - pos(y) \rvert}{l}$ l longueur du domaine de valeurs pos donne la position d'une valeur sur le domaine.	tout type de valeurs dont le domaine est une liste

Tableau IV-2 : fonctions élémentaires de similarité dans CBR*Tools

De même, nous proposons une hiérarchie de fonctions d'agrégation (cf. Figure IV-31). Les fonctions d'agrégation prennent en paramètres les différentes similarités calculées pour les sous-indices d'un indice composé, et un vecteur de poids. Ce vecteur peut être constitué, suivant l'application, des poids liés aux cas sources, au cas cible ou à la similarité. L'expression des fonctions d'agrégation est précisée dans le Tableau IV-3.

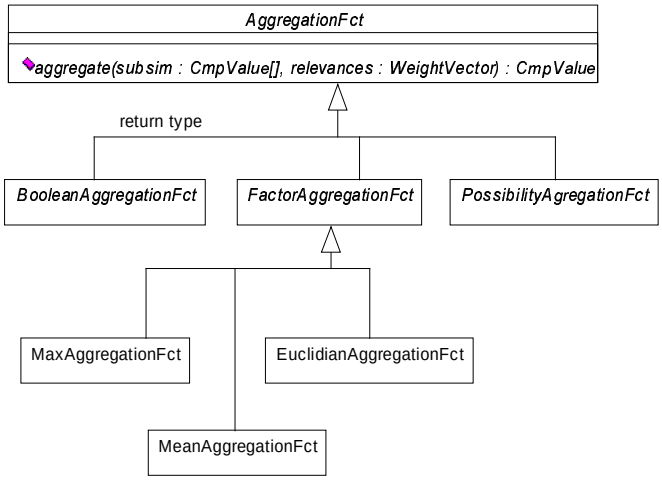

Figure IV-31 : extrait de la hiérarchie des fonctions d'agrégation

Classes	*Fonctions*
MeanAggregationFct	$g(v,w) = \frac{\sum_{i=1}^{n} v_i . w_i}{\sum_{i=1}^{n} w_i}$
EuclidianAggregationFct	$g(v,w) = \frac{\sqrt{\sum_{i=1}^{n} v_i^2 . w_i^2}}{\sqrt{\sum_{i=1}^{n} w_i^2}}$
MaxAggregationFct	$g(v,w) = \max_{i=1..n}(w_i v_i)$

Tableau IV-3 : fonctions d'agrégation de CBR*tools

2 Modèles à objets ouverts pour l'indexation par situations comportementales

L'intérêt de notre modélisation est de proposer un cadre général pour la modélisation du RàPC qui est enrichi par des représentations et des mécanismes spécifiques à l'indexation par situations comportementales. Ainsi, il est possible d'utiliser ce type d'indexation au sein d'une architecture *cohérente* pour construire des systèmes de raisonnement complets. Pour la modélisation de ce type d'indexation, nous avons introduit 12 nouveaux points d'ouverture (cf. chapitre III, §2.3.2) venant compléter les points d'ouverture généraux du RàPC. Ces nouveaux points d'ouverture ne concernent que les axes de la représentation des cas et de l'organisation de la mémoire. En suivant ces deux axes de variabilité, nous montrons comment les points d'ouverture existants sont utilisés et nous introduisons également les modèles à objets ouverts spécifiques pour ce type d'indexation. Les

différents concepts de notre modèle d'indexation (cf. chapitre II, §2) sont en effet représentés sous forme d'objets et de hiérarchies extensibles. Un enregistrement est un objet composé de différents objets chroniques. Une situation comportementale est composée de deux objets, l'un pour la partie instantanée, et l'autre pour la partie comportementale. La classe de ce dernier est une façade offrant ainsi une interface de manipulation simplifiée des indices comportementaux. Enfin, les comportements élémentaires, les fonctions de localisation et les contraintes temporelles sont représentés par des objets issus de hiérarchies extensibles de classes. Cette modélisation met en œuvre les patrons *Façade*, *Composite*, *Itérateur*, *Patron de méthode*, *Fabrication* et *Stratégie*.

2.1 Représentation des cas

Nous introduisons tout d'abord la modélisation des données d'observation puis celle des cas. Pour notre modèle d'indexation, un cas comprend une situation comportementale qui référence les données brutes. Enfin, les patrons de cas potentiels permettent de créer dynamiquement de nouveaux cas à partir des données brutes.

2.1.1 Représentation des données d'observation

Dans notre modèle d'indexation par situations comportementales, les données d'observation sont structurées en chroniques et en enregistrements. Nous présentons la représentation à objets de ces deux concepts, tout en insistant sur les propriétés qui faciliteront leur utilisation dans les mécanismes de l'indexation : encapsulation de l'hétérogénéité des chroniques et représentation d'un instant de référence.

2.1.1.1 *Chroniques*

Une chronique représente une succession de données ordonnées suivant leur occurrence ou leur date de mesure. Nous définissons alors une interface abstraite de manipulation d'une chronique (point d'ouverture TimeSeries) semblable à un tableau (accès aux données par leur position) dont la représentation concrète peut prendre plusieurs formes (cf. Figure IV-32). La hiérarchie des chroniques montre les différents types de chroniques possibles et notamment la chronique avec prévision (classe ForecastTS). Ce type de chroniques permet de prendre en compte les prévisions des observations (comportements futurs estimés) de manière transparente, tout en facilitant leur mise à jour par composition d'une chronique d'observations et d'une chronique de prévisions. Cette modélisation suit les principes du patron de conception *Composite* dans la mesure où la composition de ces deux sous-chroniques reste masquée. Toutefois, il n'est pas utile dans ce cas présent de modéliser une structuration hiérarchique (comme pour les indices, les bases de cas et les similarités).

Enfin, la hiérarchie des chroniques peut être étendue grâce à la définition de nouvelles classes, pour permettre la représentation de types particuliers de chroniques (domaine spécifique de valeurs, sémantique particulière des événements). Toutefois, cette modélisation reste simple et présente l'intérêt de mettre en évidence nos besoins pratiques dans le cadre de l'indexation par situations comportementales. Nous laissons en perspective l'extension de ce modèle, afin de permettre l'utilisation de travaux existants comme ceux relatifs au système de gestion de bases de données TEMPOS (Canavaggio, 1997 ; cf. chapitre II, §1.2.2.2).

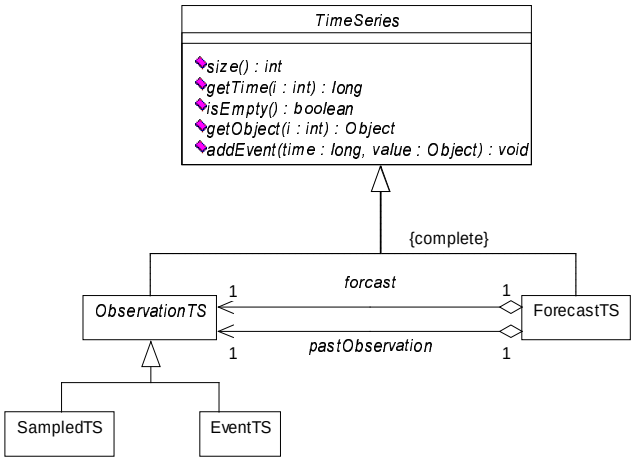

Figure IV-32 : hiérarchie des chroniques

2.1.1.2 *Enregistrements et curseurs*

Un enregistrement regroupe un ensemble de chroniques correspondant aux variables d'observation du processus, pour une période de temps définie et pour un contexte donné (cf. Figure IV-33). Chaque chronique (classe TimeSeries) est accessible depuis l'enregistrement (point d'ouverture Record) qui joue alors le rôle d'une façade (patron *Façade*) pour simplifier et prétraiter l'ajout de mesures dans les chroniques. Le contexte de l'enregistrement est utilisé pour exprimer des indices instantanés qui sont utilisés dans la définition des situations comportementales (cf. section suivante). Le contexte repose alors sur le point d'ouverture général CoumpoundIndice.

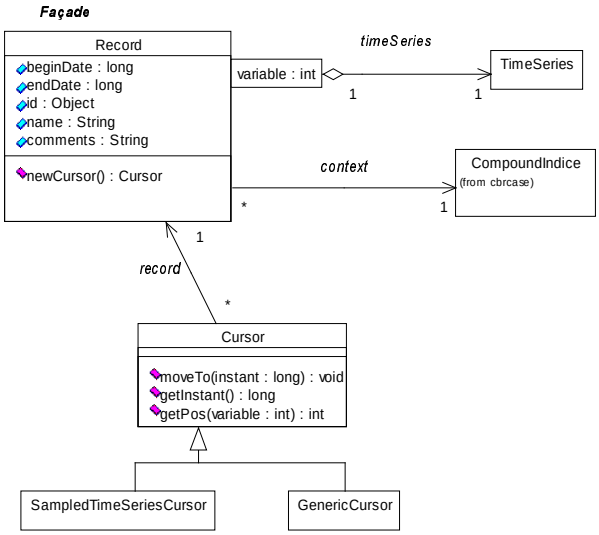

Figure IV-33 : représentation des enregistrements

De plus, dans l'indexation par situations comportementales, il est souvent nécessaire de référencer un instant précis à l'intérieur d'un enregistrement. Cet instant est représenté par un objet nommé *curseur* (point d'ouverture Cursor) qui détient cet instant ainsi que la position de la dernière donnée de chaque chronique ayant une date inférieure ou égale à l'instant de référence (application de l'opérateur Pos sur l'instant de référence, cf. chapitre II, §2.1.3.2). La sauvegarde de ces positions

permet d'éviter de les calculer à chaque manipulation des chroniques. Un curseur peut être déplacé et doit maintenir la cohérence des positions des données des chroniques qu'il référence. Un curseur est un objet abstrait permettant la réalisation de différents types de curseurs pour tirer parti des propriétés des chroniques utilisées. En particulier, la représentation et le déplacement d'un curseur deviennent très simples lorsque toutes les chroniques d'un enregistrement sont échantillonnées à la même fréquence (classe SampledTimeSeriesCursor). Toutefois, il est possible de définir un curseur ne prenant aucune hypothèse spécifique (classe GenericCursor). Enfin, un enregistrement définit une méthode de création d'un nouveau curseur (méthode newCursor suivant le patron *Fabrication*) permettant aux sous-classes de Record de définir le type de curseur par défaut à employer.

2.1.2 Représentation des cas et des situations comportementales

Conformément au modèle d'indexation par situations comportementales, une situation est composée d'une partie instantanée et d'une partie purement comportementale. Nous respectons cette séparation et la situation d'un cas est alors spécialisée (classe TimeExtendedSituation) pour contenir deux composants (cf. Figure IV-34) : l'un pour les indices comportementaux (classe BehaviouralSitiation) et l'autre pour les indices instantanés (classe InstantaneousSituation). Les indices instantanés comprennent principalement les indices du contexte de l'enregistrement auquel est liée une situation. Chaque composant représentant des indices hérite de la classe CompoundIndice pour respecter l'accès uniforme aux indices que nous avons définis. La classe CbrCase est également spécialisée (classe TimeExtendedCase) de manière à indiquer le type du cas (concret ou potentiel) et son origine (par exemple le nom du patron utilisé pour sa création), et pour référencer une situation comportementale.

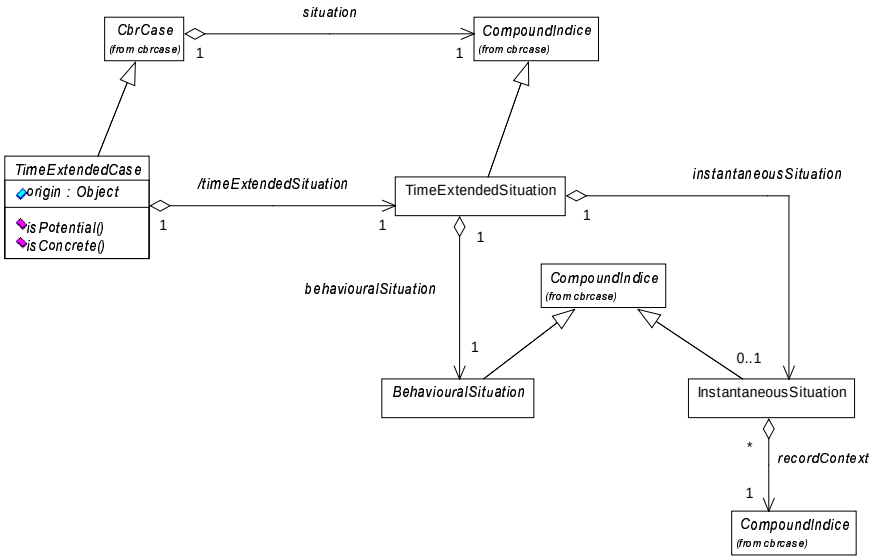

Figure IV-34 : représentation des cas et structuration des indices

La partie comportementale est à son tour décomposée pour obtenir une représentation ouverte des concepts nécessaires (cf. Figure IV-35) : les contraintes temporelles (classe IntervalContraint), les comportements élémentaires (classe RelevantSequence) et les contraintes de localisation (classe SequenceLocation). Suivant le patron *Façade*, la classe BehaviouralSituatuion donne l'accès à ces différents composants tout en simplifiant leur manipulation. En particulier, cette classe doit être spécialisée dans une application concrète pour réaliser les interfaces fonctionnelles des indices (consultation, sélection, modification et analyse). Les comportements élémentaires permettent de sélectionner les données issues d'une chronique et transforment ces données en indices (la classe

RelevantSequence hérite de CoumpoundIndice). De plus, la partie comportementale référence de manière indirecte, sur le plan du typage, les comportements élémentaires. Ainsi, les comportements élémentaires peuvent être regroupés et structurés, de manière spécifique pour chaque application, au sein d'indices composés intermédiaires.

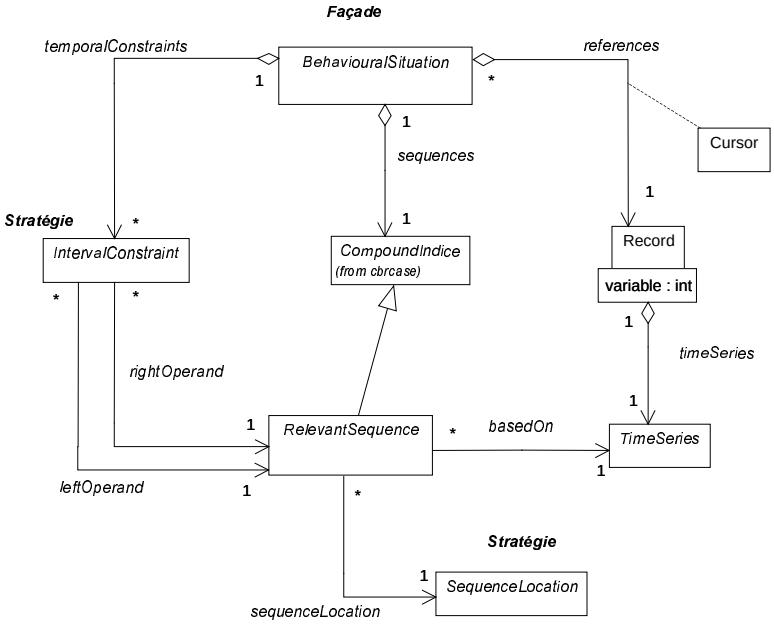

Figure IV-35 : décomposition de la partie comportementale

La représentation et le calcul de la satisfaction d'une contrainte temporelle sont délégués à un objet (classe IntervalConstraint) suivant le patron *Stratégie*. A titre d'exemple, nous avons représenté dans la Figure IV-36 trois contraintes temporelles portant sur la précédence de deux comportements élémentaires (classe IntervalAfter et IntervalBefore) et sur l'inclusion temporelle de comportements élémentaires (classe IntervalDuring). Cette hiérarchie est extensible et permet ainsi la définition des types de contraintes voulus.

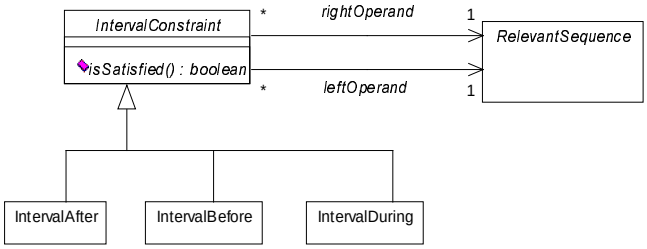

Figure IV-36 : hiérarchie des contraintes temporelles

Suivant le patron *Stratégie*, la gestion d'une contrainte de localisation est également déléguée à un objet implantant une méthode qui retourne un comportement élémentaire homologue dans une composante comportementale donnée. Différents types de contraintes de localisation ont été définis dans notre modèle d'indexation et sont ainsi représentés comme des sous-classes concrètes de la hiérarchie (cf. Figure IV-37) : localisation par temps écoulé (classe ByElapsedTimeSequenceLocation), par position (classe ByPositionSequenceLocation), par valeur (classe ByValueSequenceLocation) et enfin par référence (classe ByReferenceSequenceLocation). Il est donc possible de représenter de manière uniforme les types de contraintes existants et de nouveaux types peuvent être aisément définis.

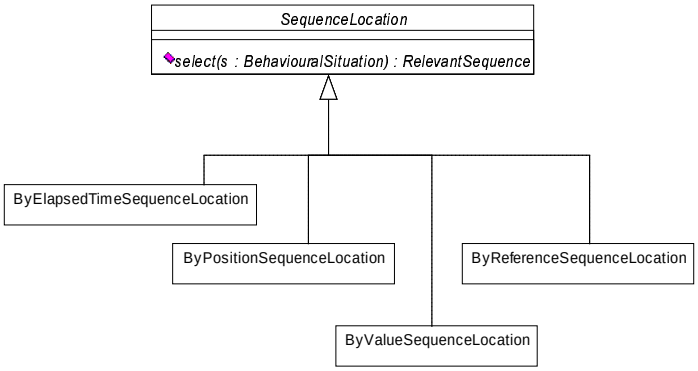

Figure IV-37 : hiérarchie des contraintes de localisation

2.1.3 Représentation des patrons de cas potentiels

La gestion d'un patron de cas potentiels est déléguée à une classe offrant deux méthodes qui peuvent être spécialisées suivant le patron *Patron de méthode* (cf. Figure IV-38) :

- prédicat d'applicabilité du patron (méthode isApplicable),
- instanciation du patron à un instant de référence dans un enregistrement (curseur) pour la création d'un cas potentiel (méthode instanciate).

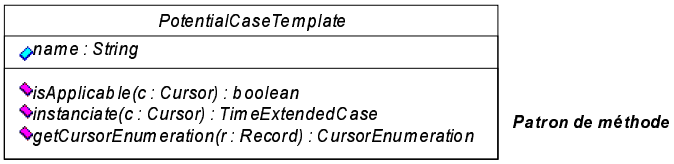

Figure IV-38 : représentation des patrons de cas potentiels

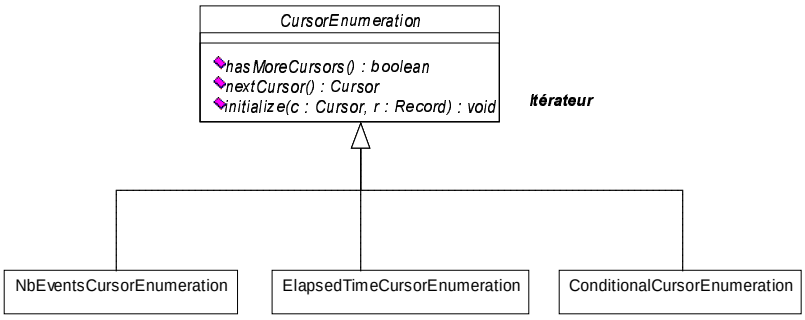

Figure IV-39 : hiérarchie d'algorithmes d'itération sur les enregistrements

Enfin, l'instanciation des cas potentiels nécessite le parcours des enregistrements, c'est-à-dire le calcul au sein d'un enregistrement d'un ensemble d'instants de référence pour lesquels les conditions d'application d'un patron sont valides. Ce parcours peut être contrôlé par un client de l'interface du patron, tout en appelant le prédicat d'applicabilité sur chaque instant. Toutefois, nous proposons d'appliquer le patron de conception *Itérateur* en encapsulant la logique du parcours dans un objet autonome (classe CursorEnumeration). Cette modélisation permet alors de concevoir des itérations adaptées à un patron particulier et aux chroniques des enregistrements considérés, tout en respectant une interface d'itération standard. A titre d'exemple, nous proposons trois types de parcours généraux qui permettent de déplacer un curseur en fonction : d'un nombre d'événements produits sur une variable (classe NbEventCursorEnumeration), d'une durée (classe ElapsedTimeCursorEnumeration) ou d'une condition à satisfaire sur l'une des variables (classe ConditionalCursorEnumeration). Un patron de

cas potentiels encapsule alors le choix du parcours approprié grâce à une méthode de création respectant le patron de conception *Fabrication* (méthode getCursorEnumeration).

2.2 Organisation de la mémoire

Nous rappelons que l'organisation de la mémoire comprend l'organisation des cas et l'organisation de l'indexation. Toutefois, l'indexation par situations comportementales introduit deux concepts spécifiques : les patrons de cas potentiels et les enregistrements. La mémoire d'un système de RàPC est alors étendue (classe TimeExtendedMemory) de manière à intégrer deux points d'ouverture qui gèrent l'accès à ces éléments (cf. Figure IV-40) : la base de patrons de cas potentiels et la base d'enregistrements (respectivement TemplateBase et RecordBase). Le point d'ouverture Memory de l'axe « gestion du raisonnement » est alors spécialisé pour intégrer la déclaration de ces composants. Ainsi, il est possible d'organiser les patrons et les enregistrements suivant les besoins d'une application, tout en offrant une interface d'accès standard. De plus, la mémoire joue encore son rôle de façade (patron *Façade*) pour gérer et simplifier l'accès et surtout l'ajout de nouveaux patrons et enregistrements. Par exemple, la mémoire peut être chargée de propager l'ajout d'un nouvel enregistrement dans les index pour qu'il puisse être utilisé lors d'un prochain raisonnement.

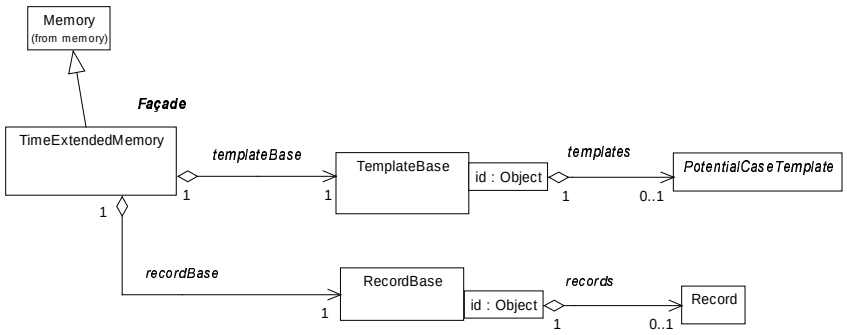

Figure IV-40 : extension de la mémoire

L'organisation des cas proprement dite s'appuie sur le point d'ouverture CaseBase. Dans le cadre de notre modèle d'indexation, une configuration typique de ce point d'ouverture consiste à définir une base composée, formée d'une base de cas volatile pour les cas potentiels et d'une base de cas persistante pour les cas concrets (cf. Figure IV-41). La base de cas volatile comprend les cas potentiels instanciés temporairement au cours d'un raisonnement. Lors de la phase d'apprentissage, ces cas seront, soit effacés, soit intégrés dans la base de cas persistante sous la forme de cas concrets. Le suivi du cycle de vie des cas est ainsi facilité. Toutefois, la structuration des bases de cas n'est pas restreinte à cet exemple, et le cadre de composition permet de définir d'autres structurations.

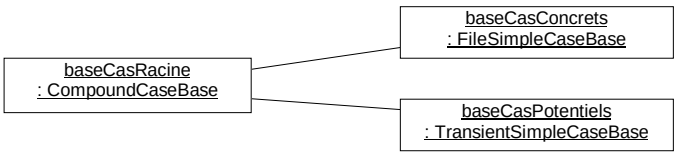

Figure IV-41 : structuration typique des bases de cas concrets et potentiels

Pour l'organisation de l'indexation, nous enrichissons tout d'abord la hiérarchie d'index avec un index spécifique pour la recherche des cas potentiels (classe PotentialCaseKnnIndex). Cet index (héritant de KnnIndex) permet d'effectuer une recherche par plus proches voisins en parcourant un ensemble d'enregistrements et de patrons de cas potentiels pour lesquels il classe les meilleurs cas

suivant leur restriction (utilisation d'un filtre d'indices pour construire dynamiquement la restriction). Les meilleurs cas potentiels sont alors retournés complètement instanciés.

La Figure IV-42 donne un exemple typique d'utilisation de cet index. Tout d'abord, l'index templateIndex recherche les patrons applicables pour le cas cible courant, le résultat est alors sauvé dans une variable nommée Template. L'index d'embranchement (nommé a) lance également la recherche sur la partie instantanée de la situation limitée au contexte des enregistrements (index recordIndex). Ce résultat est alors transmis à l'index chargé d'instancier les patrons (index p). Cet index interprète les identificateurs de sous-ensemble du résultat courant comme des identificateurs d'enregistrement. De même, le résultat de la variable Template est récupéré pour identifier les patrons à utiliser. Un enregistrement ou un patron représente en effet un sous-ensemble de cas en compréhension. Enfin, les meilleurs cas sont retournés avec un résultat de type {CaseSimIndexResult}. Cette modélisation est très flexible car elle permet d'opérer le filtrage des patrons et des enregistrements avec différents index simples ou même composés.

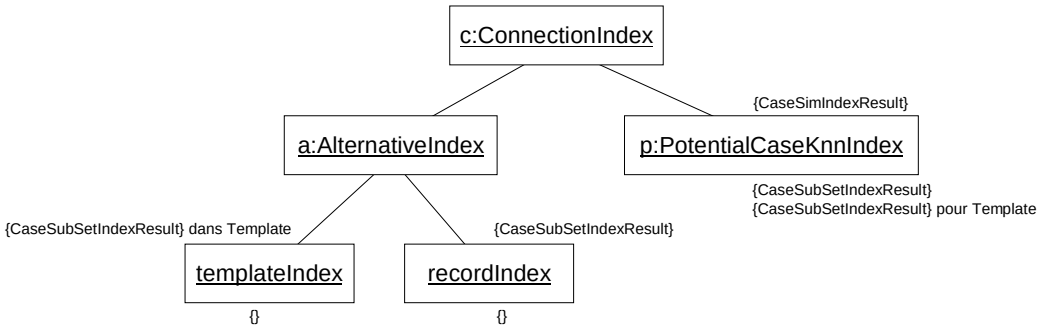

Figure IV-42 : structuration typique des index pour l'instanciation des patrons de cas potentiels

Nous introduisons également l'index BehaviouralKnnIndex qui effectue une recherche par plus proches voisins, tout en construisant la composante comportementale qui représente l'interprétation du cas cible par chaque cas (concret ou potentiel). Cet index possède le profil de paramètre {CaseSetIndexResult} et retourne un type de résultat spécifique {BehaviouralIndexResult} permettant d'accéder à cette interprétation pour chaque cas retourné. Pour permettre l'utilisation de ce type de résultat lors d'une phase de réutilisation, une nouvelle interface de sortie de la phase de recherche est également déclarée (classe BehaviouralRetrieveResult).

3 Réalisation et utilisation de CBR*Tools

Les points d'ouverture et les modèles à objets que nous avons présentés sont implantés dans notre plate-forme à objets nommée CBR*Tools. Cette plate-forme regroupe plus de 200 classes programmées dans le langage à objets Java. Nous présentons tout d'abord quelques points importants de la réalisation puis nous décrivons les principes d'utilisation de la plate-forme.

3.1 Réalisation de la plate-forme

La réalisation de CBR*Tools repose tout d'abord sur une structuration en sous-systèmes qui garde la décomposition en axes de variabilité pour faciliter la recherche des classes à utiliser. De plus, l'implantation en Java a été particulièrement étudiée pour permettre une utilisation aisée.

3.1.1 Structuration en sous-systèmes

Les classes de CBR*Tools sont regroupées en deux sous-systèmes principaux : le sous-système core ou *noyau* qui fournit les abstractions et les composants pour le RàPC de manière générale, et le sous-système time qui regroupe les classes spécifiques pour l'indexation par situations comportementales (cf. Figure IV-43).

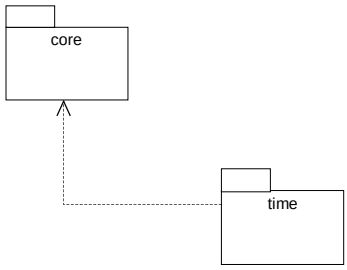

Figure IV-43 : sous-systèmes principaux de CBR*Tools

Le sous-système time comprend 37 classes et le noyau de CBR*Tools est composé de trois sous-systèmes spécifiques, permettant d'organiser les différentes classes suivant les axes de variabilité (cf. Figure IV-44) :

- reasoner (40 classes) : ce sous-système regroupe les classes d'objets de contrôle du raisonnement et les schémas de phase.
- cbrcase (28 classes) : ce sous-système contient les classes responsables de la représentation des cas : indices, types d'indices et vecteurs de poids.
- memory (109 classes) : enfin, ce sous-système, qui permet l'organisation de la mémoire, est lui-même décomposé en trois sous-systèmes différents, en plus des objets de façade contenus à son niveau :
  - casebase (16 classes) : classes pour les bases de cas,
  - index (40 classes) : classes gérant les index et les bases d'index,
  - similarity (50 classes) : classes nécessaires à la représentation des mesures de similarité : objets de contrôle, paramètres, domaines de valeurs, fonctions élémentaires de similarité et fonctions d'agrégation.

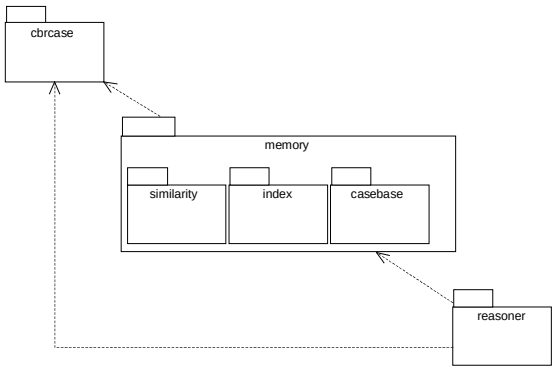

Figure IV-44 : organisation du sous-système core

3.1.2 Implantation en Java

L'implantation générale de CBR*Tools suit une méthodologie rigoureuse sachant que les classes offertes ont la possibilité d'être spécialisées et réutilisées : mécanismes d'assertion et d'exception pour détecter le plus tôt possible la violation de propriétés, conventions de nommage et

d'implantation. Par exemple, chaque point d'ouverture présenté au chapitre précédent est représenté par une hiérarchie de classes en trois niveaux :

- *interface* définissant les opérations publiques du point d'ouverture sans aucun code sous-jacent,
- *classes abstraites* proposant une implantation par défaut des parties constantes du point d'ouverture ainsi que des protocoles internes,
- *classes concrètes* réalisant des variations possibles et directement réutilisables du point d'ouverture.

Ainsi, il est possible de redéfinir complètement le comportement du point d'ouverture à partir de son interface. Les classes abstraites organisées en une hiérarchie sont alors des outils pour construire les variations concrètes du point d'ouverture. La distinction entre l'interface et les classes abstraites des points d'ouverture a été masquée dans les diagrammes de classes présentés dans ce chapitre, puisqu'elle n'apporte pas d'information pertinente au niveau conceptuel.

De plus, la représentation des indices a été particulièrement étudiée pour son intégration dans le langage Java. Ainsi, le modèle conceptuel présenté dans la Figure IV-15 a été adapté pour la réalisation. Les classes Indice et SimpleIndice sont avantageusement remplacées par la classe Object (la classe mère de toutes les classes en Java) permettant ainsi d'utiliser n'importe quelle classe d'objets pour des indices simples. Enfin, une classe abstraite (JavaClassIndice) héritant de l'interface d'un indice composé (CompoundIndice) fournit l'ensemble des opérations permettant d'utiliser une classe Java comme un indice composé. Ainsi, suivant la déclaration des indices effectuée, un attribut public ou un attribut encapsulé respectant les conventions de nommage des Java Beans (méthodes d'accès sous la forme get<nom attribut> et set<nom attribut> ; Beans, 1997), peut être reconnu comme un indice. Les attributs encapsulés permettent de plus la représentation d'indices calculés dynamiquement. La classe JavaClassIndice réalise ces opérations grâce aux fonctions de réflexivité de Java (sorte de métaniveau où les classes sont considérées comme des objets). Nous permettons également la spécialisation des classes d'indices ainsi que l'utilisation des types de base (entier et réel) tout en évitant la création d'objets pour les encapsuler. La classe suivante donne un exemple typique de déclaration d'indices dans le langage Java grâce à CBR*Tools :

```
public class CarCaseSituation extends JavaClassIndice {
   public String make;
   public float width;
   public float length;


   float getSurface() {return width*length;}


   static {
        ArrayIndiceDescriptor desc =
                new ArrayIndiceDescriptor("aid.cbr.samples.cars.CarCaseSituation");
        desc.addIndiceDescriptor(new IndiceDescriptor("make", StringType.STRING));
        desc.addIndiceDescriptor(new IndiceDescriptor("width", new RangeFloatType(60,73)));
        desc.addIndiceDescriptor(new IndiceDescriptor("length", new RangeFloatType(141,209)));
        desc.addIndiceDescriptor(new IndiceDescriptor("surface",new RangeFloatType(8460,15257)));
   }
}
```

Pour finir, l'expression des indices dans une classe java étant considérée par CBR*Tools comme un cas particulier, d'autres classes abstraites permettent de représenter les indices de différentes manières : structure dynamique (table de hachage, tableau) et séquence d'une chronique (classe RelevantSequence).

Toutefois, nous pouvons regretter que Java n'offre pas en standard des moyens de représentation à objets des connaissances. Le niveau de réflexivité que nous avons utilisé est très limité, et ne permet

pas une personnalisation fine de la description des classes comme dans un langage muni d'un métaniveau (Jaczynski, 1993). La modélisation que nous proposons représente un bon compromis, mais nous laissons en perspective l'ajout de composants pour étendre et gérer la cohérence de la description des indices et de leurs facettes (cf. le système ROCADE, Fuchs, 1997).

3.2 Utilisation de CBR*Tools

L'utilisation de notre plate-forme est facilitée par son intégration au sein d'un environnement graphique permettant de visualiser et de manipuler directement les diagrammes de classes. Toutefois, le raisonnement à partir de cas est intrinsèquement un processus complexe, et nous avons dû introduire de nombreux points d'ouverture pour sa modélisation. Nous donnons alors deux critères d'estimation de la complexité d'utilisation de notre plate-forme pour réaliser un système de RàPC : l'effort minimum et les dépendances systématiques entre points d'ouverture.

3.2.1 Environnement d'utilisation

La plate-forme CBR*Tools peut bien évidemment être directement utilisée comme tout autre ensemble de classes en Java grâce à un éditeur ou à un environnement de programmation classique. Toutefois, nous proposons son utilisation au sein d'un atelier de génie logiciel permettant de visualiser et de manipuler directement l'intégralité des modèles UML présentés.

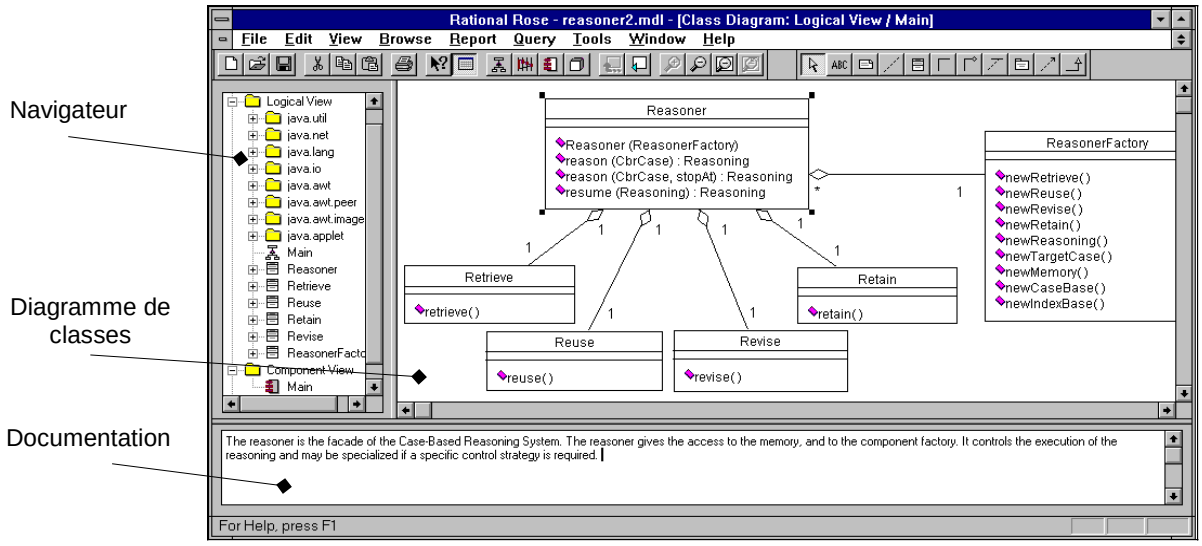

Figure IV-45 : interface d'utilisation de CBR*Tools avec Rational Rose

Durant la conception de CBR*Tools, nous avons en effet utilisé l'atelier de génie logiciel Rational Rose (Rose, 1998 ; Quatrani, 1998). Cet atelier offre une interface graphique (cf. Figure IV-45) permettant de naviguer dans les différents éléments conceptuels (classes, paquetages, scénarios), d'afficher ces éléments sous forme de diagrammes exprimés en UML, et de consulter les documentations associées à chaque élément (classe, attribut, méthode).

L'utilisation d'un tel outil ne s'arrête pas à la conception initiale de CBR*Tools mais a aussi pour but de fournir un environnement de conception permettant de faciliter l'utilisation de CBR*Tools. En effet, cet outil est complémentaire de notre approche reposant sur le concept de plate-forme à objets :

- *Il facilite la compréhension de la plate-forme*. Cet outil permet effectivement de manipuler visuellement les différents concepts de CBR*Tools ainsi que leurs interactions. Ces manipulations mènent non seulement à une meilleure compréhension de CBR*Tools mais

aussi à une meilleure conception par la mise en évidence des points difficiles le plus tôt possible dans le cycle de développement.

- *Il facilite l'extension de la plate-forme*. La spécialisation d'une classe de CBR*Tools se fait alors par simples manipulations graphiques : définition d'une nouvelle classe, déclaration de l'héritage, et déclaration des méthodes à spécialiser par un simple mécanisme de copier/coller.
- *Il permet de générer une documentation de qualité*. La documentation est importante pour une plate-forme, et cet outil permet d'exporter l'ensemble des diagrammes qui peuvent alors être inclus dans la documentation utilisateur.
- *Il facilite l'implantation*. A partir des diagrammes, les structures des classes sont automatiquement générées dans le langage choisi (Java) et doivent être complétées manuellement. Cette génération automatique évite donc un travail fastidieux qui est une source d'erreurs, pour pouvoir se concentrer sur la réalisation des algorithmes.
- *Il facilite le développement itératif*. Cet outil est également pourvu d'un module de ré-ingénierie (*reverse engineering*) particulièrement utile dans un processus itératif de conception et d'utilisation comme celui d'une plate-forme à objets. En effet, le mécanisme de génération et celui de la ré-ingénierie représentent les seuls moyens d'assurer la cohérence entre les diagrammes et la réalisation effective.

3.2.2 Estimation de la complexité d'utilisation de CBR*Tools

Dans cette section, nous donnons une estimation de la complexité d'utilisation de notre plate-forme. Tout d'abord, nous estimons l'effort minimum à réaliser pour construire des applications simples. Puis nous donnons les dépendances systématiques entre points d'ouverture qui sont dictées par la plate-forme.

3.2.2.1 *Effort minimum*

L'effort minimum représente les points d'ouverture à *spécialiser obligatoirement*, les autres points d'ouverture étant munis d'une valeur par défaut ou d'un ensemble de composants directement réutilisables par instanciation. Nous distinguons également les spécialisations guidées par l'application et les spécialisations nécessaires pour déclarer et construire le système avec la plate-forme. Pour le niveau « RàPC », un nouveau système doit obligatoirement définir les phases du raisonnement (recherche, réutilisation et apprentissage, la phase de révision étant souvent absente), la représentation des cas et des indices, et enfin une stratégie d'indexation qui utilise généralement une similarité (cf. Tableau IV-4). Il s'agit enfin de définir la fabrique du système de raisonnement qui permet de déclarer les composants à utiliser (ReasonerFactory). L'application donnée en exemple dans l'annexe C illustre cet effort minimum. Le système de raisonnement ainsi construit configure les 8 points d'ouverture obligatoires, tout en définissant une base de cas spécifique (la base de cas volatile par défaut n'étant pas adaptée).

	Axe « gestion du raisonnement »	*Axe « représentation des cas »*	*Axe « organisation de la mémoire »*	*Total*
Points d'ouverture orientés application	Retrieve, Reuse, Retain	CbrCase, Compoundndice	IndexBase, Similarity	*7*
Points d'ouverture orientés plate-forme	ReasonerFactory	-	-	*1*
Total	*4*	*2*	*2*	*8*

Tableau IV-4 : points d'ouverture à spécialiser obligatoirement pour une application au niveau « RàPC »

Pour une application utilisant l'indexation par situations comportementales, 14 points d'ouverture sont à spécialiser obligatoirement dont uniquement 5 sont nouveaux et spécifiques à ce type d'indexation. Il s'agit en effet de définir les enregistrements, la structure de la composante instantanée, les types de comportements élémentaires, au moins un patron de cas potentiels, un filtre d'indices pour construire la restriction et la déclaration du patron dans la base des patrons (cf. Tableau IV-5).

	Axe « gestion du raisonnement »	***Axe « représentation des cas »***	***Axe « organisation de la mémoire »***	***Total***
Points d'ouverture orientés application	Retrieve, Reuse, Retain	TimeExtendedCase, CompoundIndice, *BehaviouralSituation*, *RelevantSequence*, *Record*, *PotentielCaseTemplate*	IndexBase, Similarity, IndiceFilter, *TemplateBase*	*13*
Points d'ouverture orientés plate-forme	TimeExtendedReasonerFactory	-	-	*1*
Total	*4*	*6*	*4*	*14*

Tableau IV-5 : points d'ouverture à configurer obligatoirement pour une application au niveau « Indexation par situations comportementales »

3.2.2.2 *Dépendances systématiques entre les points d'ouverture*

Pour faciliter l'utilisation de la plate-forme et évaluer la complexité des relations entre les points d'ouverture de CBR*Tools, nous donnons la liste exhaustive des dépendances systématiques entre ces derniers qui sont imposées par la plate-forme. Ces dépendances sont de deux sortes :

- *dépendance déclarative.* La configuration d'un point d'ouverture (spécialisation ou instanciation) se traduit par une déclaration au sein d'un ou plusieurs points d'ouverture. Cette déclaration peut se faire par une simple affectation de paramètre ou par une spécialisation triviale. En d'autres termes, *si A dépend déclarativement de B, l'utilisation d'un nouveau type de A implique une déclaration dans B*. Ce type de dépendance est représenté par une flèche en pointillé de A vers B.
- *dépendance algorithmique*. La configuration d'un point d'ouverture entraîne une modification algorithmique d'un ou plusieurs autres points d'ouverture. Dans ce cas, il est nécessaire de spécialiser ces points d'ouverture. En d'autres termes, *si A dépend algorithmiquement de B, l'utilisation d'un nouveau type de A implique la programmation de B*. Ce type de dépendance est représenté par une flèche pleine de A vers B.

L'axe de gestion du raisonnement comprend le point d'ouverture ReasonerFactory qui est remplacé par TimeExtendedReasonerFactory pour l'indexation par situations comportementales. Ce point d'ouverture est central pour la déclaration des types d'objets à utiliser dans un système de raisonnement. Ainsi de nombreuses relations de dépendance pointent vers la fabrique (cf. Figure IV-46).

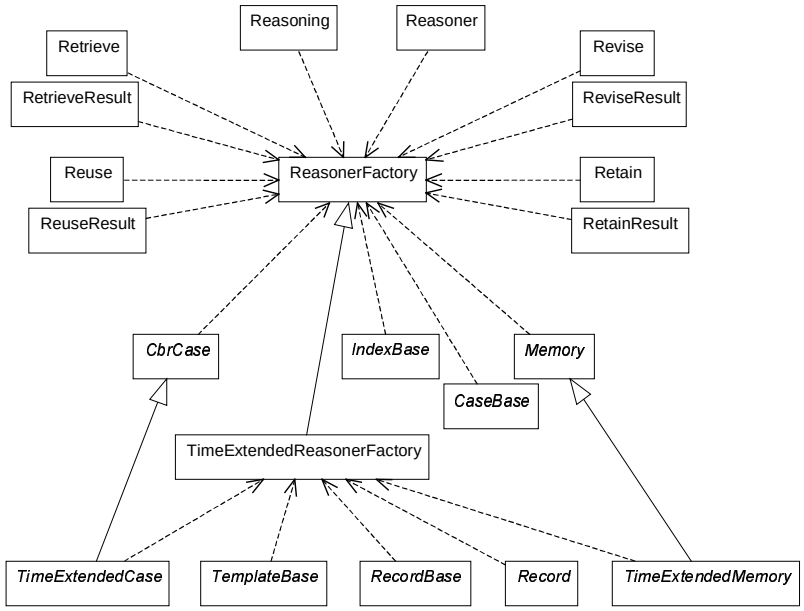

Figure IV-46 : dépendances de l'axe « gestion du raisonnement »

Pour l'axe de représentation des cas, la définition du patron de cas potentiels est un point d'ouverture important. Un patron de cas potentiels doit en effet utiliser l'ensemble des types d'objets servant à construire un cas (cf. Figure IV-47).

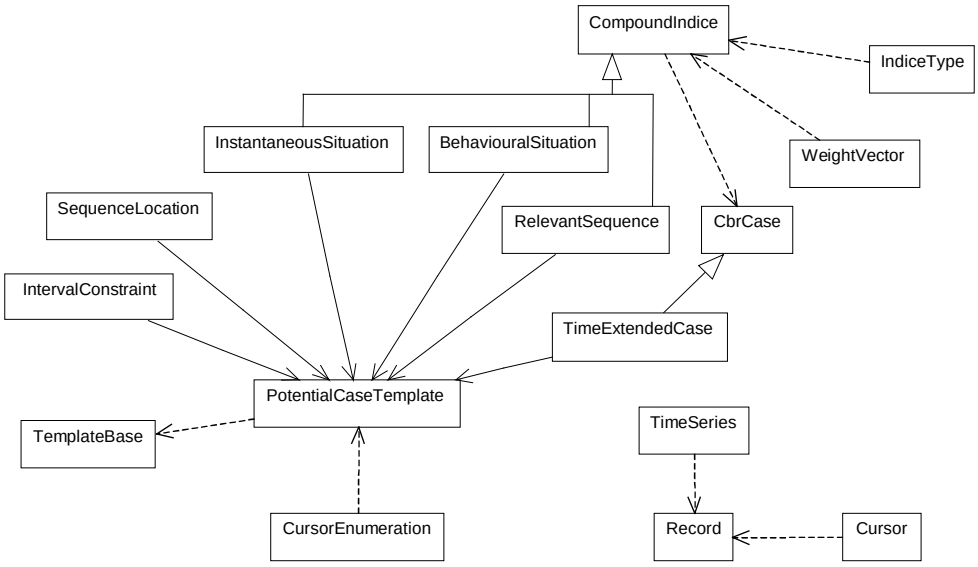

Figure IV-47 : dépendances de l'axe « représentation des cas »

Enfin, l'axe de l'organisation de la mémoire comprend surtout des dépendances déclaratives (cf. Figure IV-48). Toutefois, la définition d'un domaine de valeurs pour les similarités nécessite que des fonctions élémentaires de similarité ainsi que des fonctions d'agrégation soient définies sur ce domaine.

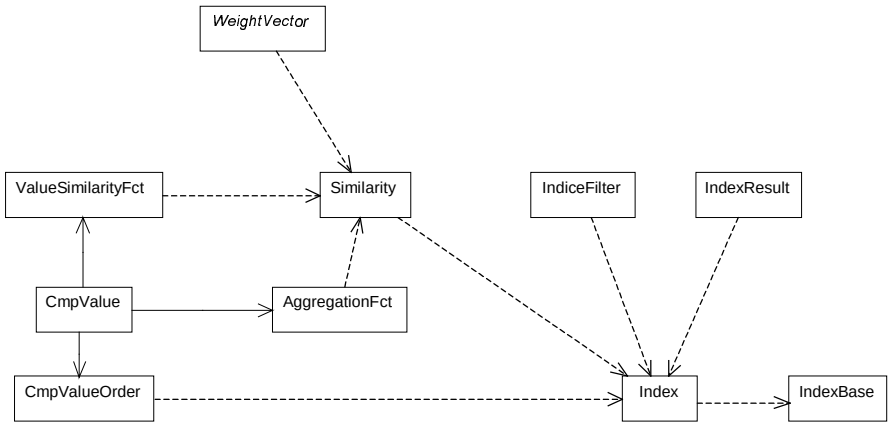

Figure IV-48 : dépendances de l'axe « organisation de la mémoire »

Nous concluons que la configuration d'un point d'ouverture entraîne peu de dépendances systématiques, surtout grâce à la centralisation de la plupart des déclarations dans la fabrique. Toutefois, ces dépendances ne prennent en compte qu'une partie des relations sémantiques qui peuvent exister entre les points d'ouverture. Ces relations, fortement dépendantes de l'application visée, peuvent devenir très complexes. Par exemple, l'introduction d'un nouveau contrôleur de raisonnement peut nécessiter la modification des types de résultats des phases. Pour produire ces résultats, les phases elles-mêmes sont susceptibles d'être modifiées ainsi que l'organisation de l'indexation de la mémoire et la représentation des cas. La documentation de telles dépendances dépasse le cadre de notre thèse, et nécessiterait une méthodologie de conception de systèmes RàPC couplée avec une documentation de haut niveau sur la plate-forme elle-même, par exemple sous la forme de scénarios d'utilisation (Johnson, 1992 ; Meusel *et al*., 1997 ; Froedlish *et al.*, 1997).

4 Conclusion

Suite à la présentation des points d'ouverture nécessaires au RàPC et à l'indexation par situations comportementales, nous avons proposé dans ce chapitre un ensemble de modèles à objets ouverts pour la conception d'une plate-forme en RàPC. Notre démarche originale tient principalement dans l'explication des décisions de conception en termes de patrons. Dans le domaine du génie logiciel, nous avons ainsi montré concrètement comment des opérations complexes, en l'occurrence un raisonnement, peuvent être modélisées avec des patrons, la difficulté majeure étant d'identifier les patrons à appliquer pour atteindre les objectifs de flexibilité.

Dans le domaine du RàPC, notre utilisation des patrons de conception présente deux apports importants. Premièrement, grâce au langage abstrait formé par les patrons et leur instanciation dans un contexte de RàPC, nous avons mis en évidence des objets-métier (schémas de phase, index, similarités, bases de cas) ainsi que leurs façons de collaborer. Les modèles obtenus sont ainsi réutilisables au niveau conceptuel pour guider la réalisation d'applications RàPC de manière générale.

Deuxièmement, ces modèles ont mené à la réalisation concrète d'un outil pour le raisonnement à partir de cas et plus particulièrement pour la gestion de l'indexation par situations comportementales. Cet outil, sous la forme d'une plate-forme à objets nommée CBR*Tools, offre un ensemble de composants réutilisables au sein de l'architecture abstraite modélisée. L'utilisation de la plate-forme est également facilitée par son intégration dans un atelier graphique où les diagrammes de classes sont accessibles et peuvent être spécialisés. Cet outil permet enfin la capitalisation d'objets-métier

notamment sous les formes d'index, de fonctions de similarité et surtout de schémas de phase de raisonnement.

CBR*Tools est implanté en Java et représente plus de 200 classes. Nous avons premièrement donné une estimation *a priori* de la complexité de son utilisation (effort minimum et dépendances systématiques). Deuxièmement, différentes applications ont été implantées avec CBR*Tools. Tout d'abord, des systèmes simples de raisonnement ont été réalisés. Le calcul d'un facteur de risque pour un assureur de voitures a nécessité la spécialisation d'uniquement dix classes (cf. annexe C). Nous avons également réalisé une application de recherche des cas similaires pour l'assistance à la conduite de la nutrition des plantes et une application d'aide à la navigation sur le Web (Broadway) que nous présentons dans le chapitre suivant. Cette dernière application nous a permis d'évaluer concrètement notre plate-forme (cf. chapitre V, §2.2 et §2.3.2)