JC2001 Besançon, 25 et 26 octobre 2001

Formalisation de la relation Tout-Partie : application à l'assemblage des composants logiciels

Nicolas Belloir, Jean-Michel Bruel et Franck Barbier

Université de Pau et des Pays de l'Adour
LIUPPA
B.P. 1155, 64013 Pau Cedex France
{belloir,bruel,barbier}@univ-pau.fr

Résumé

Les méthodes d'analyse et de conception ainsi que les notations normalisées comme UML, n'ont pas suivi la montée en puissance du développement de logiciels basés sur les composants. L'assemblage de composants logiciels reste un problème complexe et délicat pour les concepteurs. Dans ce cadre, nous travaillons sur une étude fondamentale de la composition/combinaison, au travers notamment, de l'étude exhaustive du concept plus général de relation Tout-Partie. Via ce type de relation, nous souhaitons élaborer une méthode de combinaison de composants logiciels. Nous cherchons à aboutir à un environnement supportant le développement d'applications à base de composants et ce, de la phase d'analyse à la phase de test. Cet article présente un panorama relatif sur l'assemblage de composants, synthétise le travail formel sur lequel d'autres efforts de recherche portent actuellement, et trace les grandes lignes de nos travaux futurs.

Mots-clés : Composant logiciel, Orienté objet, Composabilité, Relation Tout-Partie.

---

1. Introduction

Un des objectifs de l'ingénierie logicielle orientée objet est de permettre la réutilisation de parties de logiciel déjà développées [23]. Force est de constater que cet objectif n'est pas totalement atteint. Une des raisons est que la réutilisation est envisagée trop tard dans le cycle de vie du logiciel, i.e. au niveau de l'implémentation. En plus de cela, l'héritage des classes présente des défauts en ce qui concerne les objectifs de réutilisabilité, à cause, pour l'essentiel, de leur faible degré de granularité. Des structures plus conséquentes doivent être candidates à la réutilisation [7].

L'ingénierie logicielle orientée composant (*Component Based Software Engineering*, CBSE) est née de ce constat d'échec relatif. A cela, plusieurs raisons : d'une part, la composition d'objets réutilisables va au–delà du simple mécanisme d'héritage. D'autre part, les composants sont des entités généralement de taille plus importante que les objets et sont donc envisagés plus tôt dans le cycle de vie. Enfin, les applications basées composants sont plus canoniques [15] que les applications objets (existence de modèles de composants et de composabilité). La maturité des architectures distribuées du type CORBA ou les approches de type modèles de composants (EJB, CCM, ...) ont d'ailleurs contribué à ce développement en fournissant des modèles standards de composants [22].

Cependant, même si des progrès ont été faits par rapport à l'ingénierie logicielle orientée objet, la réutilisabilité reste envisagée trop tard dans le cycle de vie [21]. C'est pourquoi, nous pensons qu'un objectif primordial aujourd'hui est d'intégrer la réutilisabilité dès la phase d'analyse. A ce titre, des travaux sont menés sur deux fronts. En effet des propositions affluent au niveau de la phase de conception et s'attachent à mieux définir les notions clés comme celles de composants et de connecteurs (e.g., [33], [21]). D'autre part, des recherches sont menées sur les techniques de composition, par exemple les langages de description d'architecture (ADL [20]), les frameworks [15], les patterns [7], etc. Nous nous intéressons pour notre part à la relation de composition au niveau de l'analyse et plus particulièrement à la formalisation de la relation Tout-Partie que l'on retrouve dans de nombreuses méthodes sous des appellations diverses et controversées [12]. En effet, un certain nombre de travaux significatifs centralisent la connexion entre composants sur un point névralgique, souvent appelé *connecteur*. Cependant, son fonctionnement est encore mal défini et il n'y a pas de réel consensus sur ses attributs, ses fonctionnalités, etc. Et si au niveau de la phase d'analyse des avancées ont été réalisées en termes de spécification de la relation de composition, nous pensons qu'un travail important doit être mené pour faire le lien avec ces composants connecteurs. Au niveau de la phase d'analyse, des propriétés sont définies pour une relation de composition donnée. Or, à notre connaissance, il n'existe pas de méthode permettant de conserver ces spécifications automatiquement lors du passage à la phase de conception.

Cet article est organisé de la façon suivante : nous allons montrer dans la section 2 les différentes approches portant sur la composition logicielle, donnant au passage quelques définitions permettant de bien situer notre problématique. Nous montrerons que de notre point de vue, le maillon faible provient des limites de la relation de composition au niveau de la phase d'analyse, notamment au niveau de la composition dans UML. La section 3 présentera succinctement notre vision de la relation de composition, au travers de l'étude formelle d'une relation plus globale : la relation Tout-Partie. Nous présenterons dans la section 4 les directions dans lesquelles nous allons travailler et les principaux objectifs que nous nous fixons, avant de conclure en section 5.

2. Limites de l'approche actuelle

Cette section donne un bref aperçu des travaux menés sur la composition logicielle au niveau des applications basées composants. La composition logicielle peut être envisagée à plusieurs niveaux dans le cycle de vie du logiciel. Nous allons nous attacher à deux d'entre eux. Le premier, que nous appelons phase d'analyse, est la phase durant laquelle l'ingénieur spécifie le logiciel à un haut niveau d'abstraction. Cette phase permet de communiquer avec le destinataire du logiciel afin qu'il valide les besoins. L'utilisation de notations standardisées telles qu'UML aide à cette tâche. La seconde phase du cycle de vie à laquelle nous allons nous intéresser est celle que nous appelons phase de conception. Il s'agit de la phase durant laquelle est conçue l'architecture logicielle de l'application. Elle permet notamment de spécifier comment les composants à utiliser s'intègrent dans un environnement particulier (cf. Figure 1).

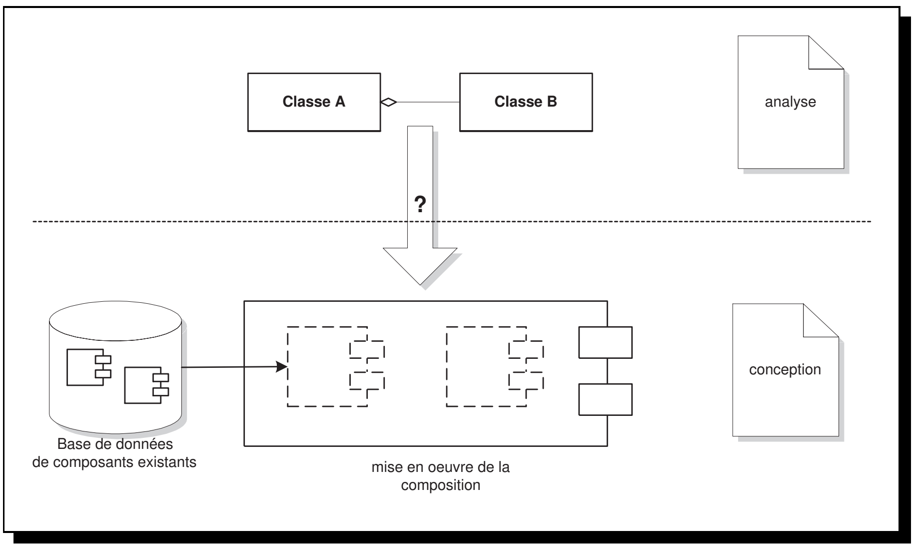

FIG. 1 – *La composition dans le cycle de vie du logiciel*

2.1. Conception des architectures logicielles

La phase de conception d'une application basée composant a pour objectif de définir l'architecture logiciel du système étudié. Shaw et Garlan [31] décrivent une architecture logicielle de la manière suivante :

Software architecture *[is a level of design that] involves the description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns.*

Les éléments autour desquels les systèmes sont construits sont les *composants*. Ces derniers sont des unités compilables [20], cachant les détails de leur implémentation [21], et qui interagissent avec le monde extérieur à l'aide de leurs *interfaces*. Ils peuvent être déployés indépendamment par une tierce personne [33], ce qui permet un haut pouvoir de réutilisation.

Les interfaces sont un élément essentiel de cette composition. En effet, en plus de leur fonction de dialogue avec le monde extérieur, elles spécifient les services fournis par le composant [20]. Malgré leur rôle informatif, elles donnent difficilement une bonne connaissance du comportement détaillé du composant [9]. On peut juste noter des apports tels que ceux du langage Eiffel et de la programmation par contrat. Au delà des simples interfaces qu'ils offrent, les composants Eiffel fournissent des règles de composabilité à travers l'expression de leurs pré–conditions, post–conditions et invariants [23]. Notons également que les modèles de composants, tels que les EJB (Enterprise Java Beans) [32] ou le CCM (CORBA Component Model) [25], proposent des approches intéressantes en terme de composition. Ils mettent notamment en œuvre des notions telles que le connecteurs ou les interfaces (voir plus bas). Cependant, ces approches sont très fortement orientées implémentation et ne traitent pas la composition au niveau de l'analyse. Nous pensons de plus qu'ils ne prennent pas assez compte du cas où les composants à assembler ne sont pas créés via le modèle lui-même (cas des *Component Off The Shelf* – COTS).

Si on regarde au niveau conceptuel comment assembler deux composants, une première approche assez naïve consiste à relier une interface d'un composant fournissant un service à un autre composant nécessitant ce service. La représentation de cette approche est typiquement un schéma dans lequel des carrés (matérialisant les composants) sont reliés entre eux par des liens (représentants les interfaces). Cependant deux problèmes se posent. Le premier apparaît lors de la phase suivante du cycle de vie du logiciel. En effet, lors de l'implémentation, on peut se trouver confronté à des composants écrits dans différents langages de programmation, ou ayant des interfaces incompatibles. Or, la phase de conception décrit bien qu'il faut faire interagir deux composants mais ne dit pas comment. Le second problème est que lors de la phase d'analyse, des contraintes ont pu être posées et cette simple représentation ne permet pas de les satisfaire. Pour ces raisons, cette approche est insuffisante.

Pour pallier à ce manque, différentes propositions ont été formulées. Toutes ont un point commun : la nécessité de *décrire la relation de composition* entre deux – ou plusieurs – composants. Un certain nombre de chercheurs défendent l'approche des *connecteurs*. Les connecteurs sont des blocs de construction architecturale utilisés pour modéliser les interactions entre les composants et les règles qui gouvernent ces interactions [20]. [21] et [30] parlent de "médiateurs pour l'interconnexion". Les divergences entre ces approches portent sur la forme que doivent prendre ces connecteurs et sur la nature de leur rôle. En effet, certains les considèrent uniquement comme des entités abstraites, n'ayant pas de forme exécutable [20] qui donc, n'ont de sens qu'au niveau de la conception. Au contraire, [6] se positionne sur le fait que si les connecteurs ne sont pas des éléments exécutables, cela peut impliquer deux choses. Il peut y avoir une perte d'information entre la conception et le développement (par exemple perte d'information sur les relations entre éléments du système) car l'information spécifiée au niveau du connecteur n'est alors pas reportée. Il se peut également que certaines informations vont devoir se retrouver au sein même du composant (le comportement du connecteur se retrouve au niveau du composant) ce qui implique sa modification.

Une autre notion proposée est celle de *glue* [29], aussi appelée *wrapper* dans [34] et de *script* pour mettre en œuvre la notion de connecteur lors de la phase d'implémentation. La glue a pour rôle d'enrober le composant à intégrer de manière à rendre possible la mise en relation d'interfaces incompatibles. Le langage de script a pour rôle de décrire la manière dont les composants sont reliés entre eux.

La Figure 2 montre un exemple de connecteur dans une relation de composition. Dans cette relation, le composé est un *site web* et les composants sont le composant *base de données* et le composant *script*. Un connecteur a été modélisé afin de faire le lien entre les deux composants. Cette figure illustre un autre point peu clair de la relation de composition : quels types d'interfaces faut-il rendre accessibles depuis l'extérieur du composé? Deux approches se présentent : la première consiste à modéliser de nouvelles interfaces propres au composé et à cacher celles des composants. Cette approche présente l'avantage de

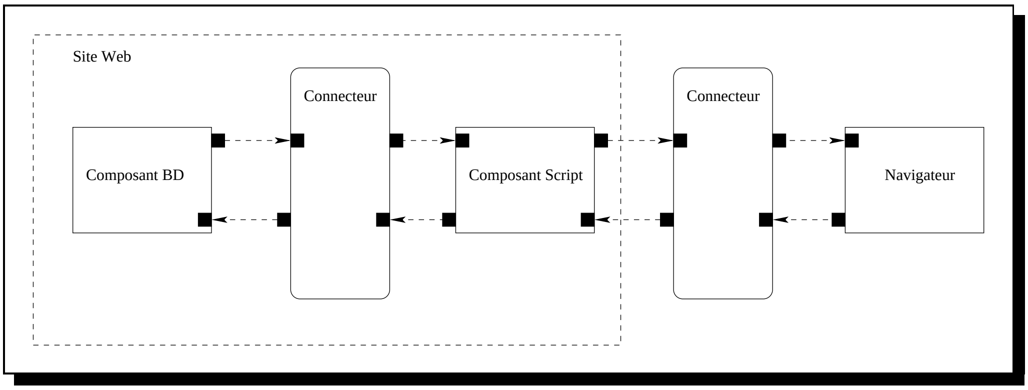

FIG. 2 – *Représentation d'une relation de composition pendant la phase de conception*

rendre le composé totalement opaque en cachant totalement ses composants. La seconde approche consiste à modéliser de nouvelles interfaces pour le composé lorsque cela est nécessaire mais à rendre les interfaces "utiles" des composants accessibles depuis l'extérieur du composé. Cette approche, une fois implémentée, présente l'avantage de réduire la charge de calcul grâce à l'optimisation des accès aux interfaces des composants.

Les différentes manières de décrire la relation de composition au niveau conceptuel sont révélatrices de la difficulté que représente l'implémentation fiable et correcte de cette relation.

2.2. La relation Tout-Partie

Nous pensons qu'une des raisons pour lesquelles la mise en œuvre au niveau conceptuel de la relation de composition est difficile, est que celle-ci est mal maîtrisée au niveau de la phase d'analyse. Conceptuellement, la relation de composition se modélise par la notion d'agrégation (cf. Figure 3 pour un exemple en UML). En effet, celle-ci permet de modéliser la relation de composition comme un objet de haut niveau mettant à disposition de clients, un ensemble d'interfaces fournissant ou nécessitant des services. Ces interfaces peuvent être propres à l'objet de haut niveau ou bien être celles des objets composants, mais sont vues de la même manière par les clients : il y a encapsulation [19, p.15]. Or, l'agrégation est une forme de relation Tout-Partie [5, p.38] :

Aggregation *is often referred to as a 'whole-part'[...] where the whole, the aggregate, is made up of its parts.*

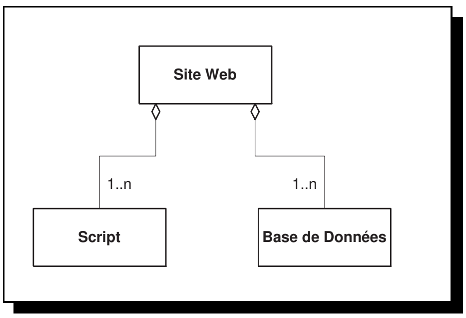

FIG. 3 – *Représentation d'une agrégation en UML*

Nous allons étudier dans la section suivante la relation Tout-Partie, qui est la forme abstraite de la relation de composition en génie logiciel. Dans le but d'illustrer le besoin d'une spécification plus spécialisée que celle trouvée dans UML, nous utilisons quatre exemples typiques, chacun illustrant une caractéristique particulière et représentative de la relation Tout-Partie : *OEUF–Jaune*, *PIÈCE–Mur*, *COMMANDE–Ligne de commande*, *SECTION–Paragraphe*.

Par exemple, dans la relation *PIÈCE–Mur*, le même mur peut être composant de plusieurs pièces (dans la suite, nous appelons cette caractéristique *partageabilité*). Il est clair que cette caractéristique n'est pas vrai pour la relation *OEUF–Jaune*, qui doit donc appartenir à un type différent de relation Tout-Partie. En regardant la relation *COMMANDE–Ligne de commande*, on se rend compte que détruire une commande entraîne la destruction des lignes de commande (dépendance de cycle de vie) alors que casser un oeuf n'entraîne pas la destruction de son jaune. Ainsi, des Parties survivent parfois au Tout, et parfois non. Pour la relation *SECTION–Paragraphe*, un paragraphe donné peut être séparé de la section donnée pour être dynamiquement réalloué dans une autre section. Faire de même avec le mur d'une pièce ou avec le jaune d'un oeuf est évidemment impossible. En d'autres termes, il est nécessaire d'avoir à disposition des notations variées, pour pouvoir exprimer les relations ayant une sémantique particulière. La relation Tout-Partie doit permettre de modéliser des relations de composition ayant des propriétés bien différentes. Nous allons présenter dans la section suivante notre modèle de relation Tout-Partie.

3. Un modèle pour la relation Tout-Partie

UML se pose en tant que standard industriel de modélisation des applications basées composants. Il fournit pour cela des archétypes graphiques d'assemblage de composants ([27], pp 3-159). Cependant, leur sémantique est suffisamment mal définie pour engendrer une recherche sur le sujet. Nous étudions et nous critiquons, au niveau modélisation, la relation Tout-Partie telle qu'elle est implantée dans UML. Nous caractérisons des propriétés essentielles (dites premières) et des propriétés secondaires. Nous pensons qu'une bonne caractérisation au niveau de la phase d'analyse fournira au niveau de la phase de conception des techniques de combinaisons de composants plus sûres.

3.1. Limites de la relation Tout-Partie dans UML 1.4

Dans UML 1.4, la relation Tout-Partie est mise en œuvre par les notions d'*agrégation* et de *composition*. Cependant, de nombreux travaux (e.g. [11]) ont montré que ces notions sont incomplètes, ambiguës, et erronées. L'étude de la documentation d'UML [26] montre d'ailleurs plusieurs incohérences. Par exemple, la représentation graphique du méta-schéma d'UML n'est pas en accord avec les définitions données par la documentation elle-même. La documentation précise d'ailleurs que les définitions traitant de la relation de composition sont parfois "légères" ([27], pp 2-8). Enfin, en plus de ces incohérences, [2] a prouvé l'inconsistance des représentations graphiques de la relation d'agrégation et de composition, à savoir le losange blanc et le noir. Ces trois exemples sont illustratifs de la non maîtrise de cette relation au niveau de la phase d'analyse en UML. C'est pourquoi nous pensons que de plus amples études sont nécessaires afin de mieux maîtriser la modélisation de la relation Tout-Partie. Ces études devront porter, en premier lieu, sur la relation Tout-Partie en génie logiciel puis être appliquées à UML afin de fournir des outils de modélisation cohérents et explicites.

3.2. Propositions de modèle de la relation Tout-Partie

L'étude de la relation Tout-partie ne se cantonne pas au génie logiciel. De nombreux travaux ont été menés sur ce sujet que ce soit en linguistique et philosophie [35], en ontologie [28], en mathématiques [18] et [1], en base de données [4], ou encore en ingénierie des connaissances [24]. L'étude approfondie de ces travaux nous a permis d'observer la récurrence et la reconnaissance de certaines propriétés. Nous proposons de classer ces propriétés afin de permettre l'identification des différentes formes de relation Tout-Partie. [16] propose un paradigme de relation Tout-Partie générique en se basant sur le standard ISO [13]. Nous reprenons ce principe de relation générique en définissant, pour la relation Tout-Partie, des propriétés primaires, secondaires et dérivées (voir [3] pour les définitions formelles). La Figure 4 présente ces propriétés sous forme de tableau.

Pour qu'une relation soit de type Tout-Partie, il faut qu'elle possède l'ensemble des propriétés primaires, les propriétés secondaires permettant de spécialiser le type de relation Tout-Partie. Les propriétés déduites le sont des primaires et secondaires. [12] donne de nombreuses informations sur la relation Tout-Partie.

Propriétés primaires	nature binaire, propriété émergente, propriété résultante, asymétrie au niveau des instances, anti-symétrie au niveau des types
Propriétés secondaires	encapsulation (visibilité), chevauchement des cycles de vie (9 cas), transitivité, partageabilité, séparabilité, mutabilité
Propriétés déduites	propagation des opérations, appartenance, abstraction, dépendance existentielle, propagation des opérations de destruction

FIG. 4 – *Propriétés primaires, secondaires et résultantes de la relation Tout-Partie*

Dans les sections suivantes nous allons décrire les propriétés primaires et secondaires et essayer de les illustrer. Nous ne traiterons pas les propriétés déduites dans cet article.

3.3. Propriétés primaires

Pour qu'une relation soit Tout-Partie, il faut qu'elle possède l'ensemble des propriétés suivantes : nature binaire, propriété émergente, propriété résultante, asymétrie au niveau des instances, anti-symétrie au niveau des types. Voyons les définitions de ces propriétés.

Nature binaire. La relation Tout-Partie est une relation binaire. Plusieurs points permettent d'affirmer cela. Notamment, [26] nous dit que : *"Only binary associations may be aggregations"*.

Asymétrie au niveau des instances. La définition de l'asymétrie est la suivante :

$$\textit{asymétrie} \iff \textit{irréflexivité} \wedge \textit{anti-symétrie}.$$

Dans [17, 8], il est démontré que la relation Tout-Partie est irréflexive au niveau des instances et non irréflexive au niveau des classes. Par exemple, si on prend la relation *BRANCHE D'ARBRE – Branche d'arbre* (une branche d'arbre est composée de branches d'arbres), elle est non-irréflexive au niveau des classes car la classe branche d'arbre est composée d'elle-même mais elle est irréflexive au niveau des instances car une branche d'arbre ne peut être composée d'elle-même.

Nous avons fixé l'anti-symétrie au niveau des instances car c'est ainsi une contrainte d'intégrité formalisant le fait que, un objet A, composé d'un objet B, ne puisse pas se retrouver être le composant de ce même objet B. Cette contrainte n'est pas exprimée au niveau des classes. Si on prend comme exemple la relation *BRANCHE D'ARBRE – Branche d'arbre*, l'anti-symétrie fixe le fait que si une branche A est composée d'une branche B, alors la branche B ne peut être composée de la branche A.

L'irréflexivité étant prouvée et l'anti-symétrie étant fixée, la relation Tout-Partie est asymétrique au niveau des instances.

Anti-symétrie au niveau des types. La relation Tout-Partie est utilisée en génie logiciel dans la phase d'analyse et dans la phase de conception. La phase d'analyse a pour objectif de définir une représentation intelligible des besoins et du fonctionnement de l'application, à un niveau abstrait. Cette représentation a pour objectif, notamment d'être claire, afin d'être comprise par le destinataire du logiciel. Ainsi, ce dernier peut valider l'analyse du système. Au niveau de la phase de conception, la relation Tout-Partie est un outil pour construire des modules logiciels ayant une forte cohésion et un faible couplage. Lors de ces deux phases, l'objectif de la relation Tout-Partie est de construire des modèles lisibles, donc le plus simple possible, ce qui doit permettre à terme une implémentation simple. Dans cette objectif, nous avons choisi de promouvoir l'anti-symétrie au niveau des types [12].

Propriétés émergentes et résultantes. Une propriété émergente est une caractéristique intrinsèque du composé, c'est à dire que sa valeur est déterminée indépendamment de ses composants [16]. Il s'agit par exemple du nom d'une pièce dans la relation *PIÈCE–Mur*.

Une propriété résultante est une caractéristique du composé dont la valeur dépend directement de ses composants [16]. Il s'agit par exemple de la surface d'une pièce dans la relation *PIÈCE-Mur*, celle-ci étant déterminée notamment par la longueur des murs qui la composent.

Une relation, pour être de type Tout-Partie, doit posséder au moins une propriété émergente (sinon, il s'agit d'une simple encapsulation) et au moins une propriété résultante (sinon, il s'agit d'une simple association).

3.4. Propriétés secondaires

Les propriétés secondaires d'une relation Tout-Partie sont les propriétés qui permettent de la caractériser. Ces propriétés sont les suivantes : transitivité, encapsulation, dépendance des cycles de vie, partageabilité, séparabilité et mutabilité. Dans cette section, nous nous intéressons peu à la transitivité et à l'encapsulation qui sont moins discriminatoires par rapport aux autres propriétés secondaires [11].

Transitivité. Il s'agit pour une relation Tout-Partie A–B et pour une relation Tout-Partie B-C, de permettre la relation Tout-Partie A–C. Seuls certains types de relation Tout-Partie sont transitifs [36]. Cela signifie que la transitivité n'est pas une propriété primaire mais une propriété secondaire. Par exemple, si on prend la relation *LABORATOIRE–Chercheur* et la relation *CHERCHEUR–Partie du Corps*, alors il n'est pas possible d'avoir la relation *LABORATOIRE–Partie du Corps*. Il n'y a pas transitivité.

Encapsulation. Un objet composant est encapsulé par un objet composé si et seulement si ce composé est le seul objet du système complet à pouvoir accéder à ce composant. Cela signifie que le composé doit fournir un ensemble d'interfaces qui lui sont propres, parfois redondantes avec les interfaces des composants qui sont invisibles de l'extérieur du composé. De plus amples détails sont donnés dans [2].

Dépendance de cycles de vie. Cette propriété permet de spécifier les relations existantes entre le cycle de vie du composé et le cycle de vie du composant. Plusieurs cas de figure sont possibles selon que le composant est créé avant, en même temps, ou après le composé, et qu'il est détruit avant, en même temps, ou après le composé. Il y a donc en tout 9 cas possibles.

Partageabilité. La partageabilité d'un composant est sa capacité à faire potentiellement partie de plusieurs composés. Il s'agit, par exemple, d'un même mur formant deux pièces différentes. La partageabilité peut être de niveau *local* ou de niveau *global*. La partageabilité de niveau local est la capacité pour un composant d'être partagé par plusieurs composés d'un même type. Son contraire est l'*exclusion locale*. Elle signifie l'incapacité d'un composant à être partagé par plusieurs composés de même type. La partageabilité de niveau global est la capacité pour un composant d'être partagé par plusieurs composés de types différents. Son contraire est l'*exclusion globale*. Elle signifie l'incapacité d'un composant à être partagé par plusieurs composés de différents types.

De ces définitions, nous pouvons édicter les propriétés suivantes :

$$\textit{non partage} \iff \textit{exclusion locale} \wedge \textit{exclusion globale}$$
$$\textit{partageabilité} \iff \textit{partageabilité locale} \vee \textit{partageabilité globale}$$
$$\textit{partageabilité} \iff \neg\ \textit{(non partage)}$$

Séparabilité et Mutabilité. La séparabilité est la capacité d'un composant à être séparé de son composé. Elle est souvent utilisée comme critère de classification pour les relations Tout-Partie. Cependant, elle est sujet à confusion à cause de son manque de distinction claire avec les autres propriétés de la relation Tout-Partie. Par exemple, dans la relation *VOILIER – Voile*, la voile peut être séparée du voilier.

La mutabilité est la capacité pour un composant, lié à un composé, d'évoluer. Au contraire, l'immutabilité est l'obligation pour un composant de rester le même (de ne pas évoluer) tout au long du cycle de vie du composé. Par exemple, dans la relation *EMBRYON–Cellule*, la cellule peut muter.

Nous avons formellement établi [2] que l'immutabilité implique l'inséparabilité et qu'inversement la séparabilité implique la mutabilité.

3.5. Exemples de relations Tout-Partie

La Figure 1 présente des exemples de relations Tout–Partie et illustre leurs caractéristiques. Seules les caractéristiques principales (dans le sens de la puissance d'illustration) ont été présentées dans ce tableau.

Nous voulons, avec cet exemple, illustrer certaines propriétés, dont nous avons parlé, telle que immutabilité ⇒ séparabilité. A terme, notre objectif est de réunir un nombre important d'exemples de manière à chercher des relations, ou des inconsistances, entre ces propriétés. Cela sera le point de départ pour des travaux futurs plus détaillés.

Exemples	Propriétés Primaires		Propriétés Secondaires			
	Emerg.	Résult.	Partag.	Sép.	Mut.	Dep. de vie
OEUF–Jaune	date de ponte	poids	×			$\overline{birth}$–$\|death$
COMMANDE–Lig. de com.	adresse	montant total		×	√	$\|birth$–$\overline{death}$
PIÈCE–Mur	nom	surface	√			$birth\|$–$\|death$
SECTION-Paragraphe	numéro	longueur			√	$birth\|$–$\|death$
VOILIER-Voile	nom	taille		√	√	$birth\|$–$\|death$
EMBRYON–Cellule	race	taille		×	√	$\overline{birth}$–$death\|$
LIVRE–Page	titre	longueur		×	×	$birth\|$–$\|death$

TAB. 1 – *Exemples de relation Tout–Partie et leurs propriétés. Dans la table,* $\overline{birth}$ *(resp.* $\overline{death}$*) signifie la coïncidence de la naissance (birth), respectivement de la mort (death) pour le Tout et les parties,* $|birth$ *(resp.* $|death$*) signifie l'antécédance de la naissance du Tout (respectivement la mort), et* $birth|$ *(resp. death) signifie l'antécédance de la naissance des parties (respectivement la mort).*

4. Futurs travaux

Les sections précédentes ont présenté et situé notre proposition de classification de la relation Tout-Partie en terme de propriétés primaires, secondaires et résultantes. En partant de cette classification, nous allons travailler plus particulièrement à établir formellement les relations existantes entre les différentes propriétés de la relation Tout-Partie. Nous essayerons ensuite à partir de là de définir une méthode permettant d'aider le concepteur à passer de la phase d'analyse à celle de conception de manière consistante.

En parallèle, et toujours dans l'idée d'assurer la consistance du système tout au long de son cycle de vie, nous travaillons sur l'intégration de tests réutilisables au sein des composants logiciels, afin de certifier leur composabilité et leur validité. Ces travaux rentrent dans le cadre d'un projet européen, Component+ [14], dans lequel nous sommes contractant principal. Ce projet, destiné à améliorer la testabilité des composants, fait intervenir un certain nombre de concepts plus généraux que le test lui-même et qui pourront donc nous faire avancer vers notre objectif. De plus, via les études de cas et les projets pilotes entrant dans le cadre de ce projet, nous espérons évaluer, à l'aide de nos partenaires, les gains obtenus en terme de réutilisabilité par la mise en œuvre de nos propositions.

Enfin, nous étudions actuellement les différentes approches proposées en terme de *framework*, afin de rendre cohérent et homogène notre environnement de développement (faute de place, cette partie de notre travail n'a pas été détaillée dans cet article).

En termes de perspective, la contribution scientifique ici présentée a fait l'objet d'une réponse concernant la définition d'UML 2.0 [10], corrigeant en particulier quelques lacunes d'UML 1.4. La nouvelle caractérisation de la relation Tout–Partie que nous proposons, doit intégrer le futur méta–modèle UML, et donc améliorer la spécification de la relation de composition, fortement utilisée dans la connexion des composants logiciels, en analyse spécialement, mais aussi en implantation dans les "Component Diagram". Un enjeu du travail de recherche initié dans cet article vise à ce titre, à classifier et déterminer d'autres types de connecteurs reposant sur d'autres spécialisations possibles de Tout-Partie.

5. Conclusion

L'ingénierie logicielle basée composant connaît un engouement important, tant au niveau industriel qu'au niveau de la recherche. De nombreux travaux sont par ailleurs menés sur ce thème afin de définir des outils et des méthodes appropriées à cette problématique. On peut citer notamment les travaux menés sur la notion de composition (assemblage de composants, frameworks spécifiques à un domaine, patterns de conception, ...

Nous avons, à travers cet article, présenté le contexte des travaux d'une thèse qui démarre sur le sujet. Pour cela, nous avons montré les divergences existantes concernant la notion de relation de composition au niveau conceptuel. Nous pensons qu'une des raisons pour lesquelles il existe un flou à ce niveau vient du fait que celle-ci est sémantiquement mal définie, et notamment dans UML 1.4. A ce niveau, nous représentons la relation de composition comme un sous-type de la relation Tout-Partie. Afin de pallier à ce problème de caractérisation de la relation Tout-Partie, nous avons présenté dans la section 3 des travaux visant à mieux modéliser la relation de composition au niveau de l'analyse. Ces travaux montrent qu'un certain nombre de propriétés – ou de contraintes – peuvent être exprimées à ce niveau. Cependant, outre les compétences du concepteur, il n'existe pas à notre connaissance de méthodes et d'outils permettant d'assurer un passage consistant de la phase d'analyse à celle de conception. Cela présente, d'une part un risque de perte d'information qui pourrait se traduire par un système final au comportement différent de celui spécifié par l'analyste. D'autre part, cela présente un travail peu assisté qui se traduit par une charge de travail supplémentaire pour le concepteur. Nous pensons donc qu'il est nécessaire de proposer une méthode permettant de valider le passage de la phase d'analyse à celle de conception de manière consistante. Pour cela, nous nous basons sur une étude fondamentale de la relation de composition, au travers, notamment,de l'étude exhaustive du concept plus général de relation Tout–Partie ainsi que sur la participation à un projet européen portant notamment sur le test intégré au sein même des composants.

Remerciements

Les auteurs tiennent à remercier Annig Le Parc - Lacayrelle et Albert Royer pour leur collaboration ainsi que les relecteurs qui, par leurs remarques, ont permis d'améliorer la qualité de cet article.

Bibliographie

1. Artale (Alessandro), Franconi (Enrico), Guarino (Nicola) et Pazzi (Luca). – Part-Whole Relations in Object-Centered Systems: An Overview. *Data Knowledge Engineering*, vol. 20, n3, 1996, pp. 347–383.
2. Barbier (Franck) et Henderson-Sellers (Brian). – The Whole-Part Relationship in Object Modelling: A Definition in cOlOr. *Information and Sofware Technology*, vol. 43, n1, 2001, pp. 19–39.
3. Barbier (Franck), Henderson-Sellers (Brian), Opdahl (Andreas) et Gogolla (Martin). – *Unified Modeling Language: Systems Analysis, Design, and Development Issues*, chap. The Whole-Part Relationship in the Unified Modeling Language: A new Approach, pp. 185–209. – Hershey, PA, Idea Group Publishing, 2001, K. Siau and T. Halpin édition.
4. Bertino (Elisa) et Guerrini (Giovanna). – Extending the ODMG Object Model with Composite Objects. *In : Proceedings of OOPSLA'98*. pp. 259–270. – New-York, NY. 1998.
5. Cook (S.) et Daniels (J.). – *Designing Object Systems – Object-Oriented Modelling with Syntropy*. – Englewood Cliffs, New Jersey, Prentice Hall, 1994.
6. Ducasse (Stéphane) et Richner (Tamar). – Executable Connectors : Towards Reusable Design Elements. *Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97)*, 1997, pp. 483–499.
7. Gamma (Erich), Helm (Richard), Johnson (Ralph) et Vlissides (John). – *Design Patterns : Element of Reusable Object-Oriented Software*. – Addison-Wesley, 1995.
8. Halper (Michaël), Geller (James) et Perl (Yehoshua). – An OODB Part-Whole Model: Semantics, Notation and Implementation. *Data and Knowledge Engineering*, vol. 27, n1, 1998, pp. 59–95.
9. Heiler (Sandra). – Semantic Interoperability. *ACM Computing Surveys*, vol. 27, n2, 1995, pp. 271–273.
10. Henderson-Sellers (Brian) et Al., 1999. Proposition pour UML 2.0, pour plus de détails voir : http://cgi.omg.org/cgi-bin/doc?ad/99-12-11.
11. Henderson-Sellers (Brian) et Barbier (Franck). – Black and Whites Diamonds. *In : UML'99 – The Unified Modeling Language*, éd. par France (Robert) et Rumpe (Bernhard). pp. 550–565. – Springer-Verlag. October 1999.
12. Henderson-Sellers (Brian) et Barbier (Franck). – What is this thing called agreggation. *In proceedings of TOOLS EUROPE*, 1999, pp. 236–250.

13. ISO. – *Information Technology. Open Systems Interconnection - Management Information Services - Structure of Management Information - Part 7: General Relationship Model.* – ISO document n ISO/IEC 1065-7.2, ISO/IEC JTC1/SC21, 1995.
14. Projet Component+, En cours. Projet européen, pour plus de détail voir : http://www.component-plus.org/.
15. Johnson (Ralph E.). – Frameworks = Components + Patterns. *In : Communications of the ACM*, pp. 39–42. – 1997.
16. Kilov (Haim). – *Business Specification: The Key to Successful Software Engineering.* – Prentice-Hall, 1999.
17. Kolp (Manuel) et Pirotte (Alain). – An aggregation model and its C++ implementation. *In : Proceedings of the Int. Conf. on Object-Oriented Information Systems (OOIS'1997).* pp. 211–221. – Springer-Verlag. 1997.
18. Lano (Kevin). – *Formal Object-Oriented Development.* – London, Springer-Verlag, 1995.
19. Lewandowski (Scott M.). – Frameworks for Component-Based Client/Serveur Computing. *ACM Computing Surveys*, vol. 30, n 1, March 1998, pp. 3–27.
20. Medvidovic (Nenad) et Taylor (Richard N.). – A Classification and Comparaison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, vol. 26, n 1, january 2000, pp. 70–93.
21. Meijler (Theo Dirk) et Nierstrasz (Oscar). – *Cooperative Information Systems – Trends and Directions*, chap. Beyond Objects: Components, pp. 49–78. – San Diego, CA, Academics Press, 1998.
22. Merle (Philippe) et Riveill (Michel). – Des objets aux composants, 1999. pour plus de détail voir : http://www.essi.fr/ riveill/cours/car/07-99-slides-composants.pdf.
23. Meyer (Bertrand). – *Object-Oriented Software Construction.* – Prentice Hall, 1997.
24. Motschnig (Renate) et Kaasboll (Jens). – Part—Whole Relationship Categories and their Application in Object-Oriented Analysis. *5th International Conference on Information Systems Development: ISD 96*, september 1996.
25. OMG. – CORBA Component Model, april 1999. http://www.omg.org/.
26. OMG. – *OMG Unified Modeling Language Specification, verion 1.3.* – OMG document, Object Management Group, 1999.
27. OMG. – *OMG Unified Modeling Language Specification, verion 1.3.* – OMG document, Object Management Group, March 2000.
28. Opdahl (A.L.), Henderson-Sellers (Brian) et Barbier (Franck). – Ontological analysis of whole-part relationships in OO-models. *Information and Software Technology, 2001*, En parution.
29. Schneider (Jean-Guy) et Nierstrasz (Oscar). – *Software Architectures - Advances and Applications*, chap. Components, Scripts and Glue, pp. 13–25. – Springer-Verlag, Leonor Barroca, Jon Hall, and Patrick Hall Editions, 1999.
30. Shaw (Mary), DeLine (Robert), Klein (Daniel V.), Ross (Theodore L.), Young (David M.) et Zelzsnik (Gregory). – Abstractions for Software Architectures and Tools to Support Them. *IEEE Transactions on Software Engineering*, april 1995.
31. Shaw (Mary) et Garlan (David). – *Software Architecture : Perspectives on an Emerging Discipline.* – Prentice Hall, april 1995.
32. Sun. – Enterprise Java Beans Specifications, V1.1. http://java.sun.com/products/ejb/.
33. Szyperski (Clemens). – *Component Software – Beyond Object-Oriented Programming.* – New York, NY, Addison-Wesley, 1997.
34. Troya (J.M.) et Vallecillo (A.). – Controllers : reusable wrappers to adapt software components. *Information and Sofware Technology*, vol. 43, n 1, 2001, pp. 189–202.
35. Varzi (Achile). – Parts, Wholes, and Part-Whole Relations: The Prospects of Mereotopology. *Data and Knowledge Engineering*, vol. 20, n 3, 1987, pp. 417–444.
36. Winston (M.), Chaffin (R.) et Herrmann (D.). – A Taxonomy of Part-Whole Relations. *Cognitive Science*, vol. 11, 1987, pp. 417–444.