Rapport de TER
Réalisation d'annuaires de composants logiciels

Auboin Nicolas, Bendavid Olivier, Haderer Nicolas, Pallet David

Encadrants du T.E.R : Huchard Marianne et Tibermacine Chouki
Intervenant : Jean Rémy Falleri, Rapporteur : Christophe Dony

Table des matières

1 Introduction

Lors de notre précédent semestre, nous avons été amenés à réaliser plusieurs petits projets, application directe des enseignements suivis. Dans l'un d'eux, nous avons travaillé à deux groupes en coopération. Cette dernière s'étant très bien déroulée, nous avons choisi de nous regrouper à nouveau pour ainsi former notre groupe de TER.

Appartenant tous les quatre au parcours "Génie Logiciel", nous avons logiquement orienté nos recherches sur des sujets s'y rapportant. Après concertation, nous avons finalement opté pour le sujet "Réalisation d'annuaires de Composants Logiciels". En effet, il est question de réaliser une hiérarchisation de composants logiciels. Un composant est une unité de réutilisation, n'étant alors défini que par ses services externes (ceux fournis et ceux requis). Le TER est donc totalement axé sur le Génie Logiciel basé composants.

Après avoir étudié la programmation procédurale, objet et plus récemment agent, nous avons tous été très enthousiastes de découvrir la programmation par composants, par le biais de ce TER.

Nous allons dans un premier temps énoncer le sujet du TER. Nous expliciterons alors brièvement l'objectif du projet. Nous poursuivrons par une analyse complète de la problématique induite par le principe de "composants logiciels" et nous détaillerons la partie théorique mise en application dans le projet. Nous ferons une rapide présentation du cadre du projet à travers les langages et les outils utilisés sans oublier l'organisation avec nos encadrants et celle au sein de notre groupe. A partir de là, nous pourrons détailler clairement le travail effectué allant de l'analyse à la conception du projet. Nous dresserons en outre un bilan du TER en comparant les objectifs du cahier des charges initial et le résultat obtenu. Nous y résumerons aussi les compétences assimilées durant le projet. Enfin, nous dresserons une conclusion globale de notre TER.

Sujet du TER :

"Le génie logiciel à base de composants permet de construire des applications par assemblage de briques logicielles disponibles sur étagères. Les composants exposent leurs services : les services qu'ils mettent à la disposition de leur environnement (services fournis) et les services qu'ils doivent trouver dans leur environnement pour fonctionner (services requis). Construire une

application dans ce paradigme consiste à trouver et à connecter (assembler) les composants compatibles (services fournis correspondant aux services requis) en vue de réaliser l'objectif final. Lors des évolutions ultérieures de l'application, de nouveaux composants peuvent être recherchés pour offrir de nouvelles fonctionnalités ou pour remplacer des composants en panne ou devenus obsolètes. Pour faciliter ces processus, nous proposons de construire des annuaires de composants de type "Pages jaunes" en ce sens qu'ils sont basés sur les services. Ces annuaires répertorieront les composants et les classeront afin d'accélérer les recherches soit lors des opérations de construction, soit lors des opérations de remplacement. Un intérêt de la classification est de présenter des composants proches du composant recherché lorsque celui-ci n'est pas disponible."

L'objectif du projet vise à servir le principe d'assemblage de composants par la création d'un annuaire de composants logiciels.

L'annuaire est une hiérarchisation de composants. La procédure d'assemblage l'utilisera pour déterminer les composants appropriés à la situation. L'objectif étant qu'à terme, l'annuaire fasse appel à un composant de manière dynamique. Par exemple, le composant doit pouvoir être remplacé en cas de panne, de problème d'un composant ou encore de mise à jour.

Pour déterminer la compatibilité entre composants, il faut regarder leurs interfaces. Plus en détails, il faut étudier chaque fonctionnalité de ces interfaces. Les fonctionnalités sont elles-mêmes décrites par une signature (un nom et les paramètres d'entrée et de sortie). Pour obtenir les diverses compatibilités possibles, il est indispensable d'avoir une hiérarchisation des types de paramètres. Un sous-type étendra le type d'origine et offrira donc des possibilités de compatibilité.

Les enseignants chercheurs avec qui nous travaillerons sur ce projet ont créé une méthode pour mettre en évidence les conditions de compatibilités entre les composants. Comme nous pourrons le voir au fil du développement du rapport, nous avons activement participé à l'avancée des recherches, entraînant des modifications de conception tout au long du projet.

2 Contexte du projet

Pour mieux comprendre le sujet, nous expliciterons brièvement le paradigme de développement par composants logiciels. Dans le monde actuel du développement logiciel, une notion a vu le jour depuis l'apparition des langages objets. Cette notion est la réutilisabilité du code. Idée phare des langages objet, la réutilisation n'a pas toujours été une des préoccupations majeures des développeurs. Le projet dans son ensemble a l'ambitieuse idée de mettre en avant l'aspect réutilisation des divers modules proposés par les développeurs. Ainsi, on peut définir un composant comme un élément qui, en s'assemblant avec d'autres, forme un logiciel à part entière. Pour déterminer quel composant correspond à quelle demande, les composants sont définis par une boîte noire o seuls sont visibles les services fournis et les services requis (à la manière d'un composant électronique). A partir de cela, la réalisation d'un composant revient à faire un assemblage de "briques logicielles".

Dans ce qui suit, nous expliciterons les divers éléments qui ont composé notre environnement de développement durant notre TER. Nous y aborderons les langages de programmation employés, les outils utilisés, pour finalement dresser une courte description du cadre de travail général.

2.1 Langages utilisés

Java

Java est un langage de programmation orienté objet répandu dans le monde du développement logiciel. Il nous a logiquement été proposé par nos encadrants en raison de certaines de ses indéniables qualités : Tout d'abord, il est multiplateformes, le code Java est portable aussi bien sur Linux que Windows ou encore MacOs.

D'autre part, il dispose de plusieurs environnements de développement intégré efficaces, dont Eclipse, que nous avons utilisé tout au long de notre TER (voir la partie Outils utilisés). Enfin, sa syntaxe est simple, et l'approche objet est très nettement marquée (tout est objet hormis les primitives, tels que les nombres entiers, les nombres réels etc.). Cet aspect se révèlera majeur dans notre projet. La stagiaire ayant déjà travaillé sur ce sujet durant son stage de Master 2 avait elle aussi utilisé Java, cet aspect ne fut pas cependant déterminant dans le choix de langage de programmation, comme nous pourrons le voir au fil du rapport. Lors de notre projet, nous avons été

amené à utiliser la JavaDoc [8] et les forums de la communauté JavaFr pour la résolution de certains de nos problèmes [9].

XML

L'XML ou "Extensible Markup Language" est un langage de balisage générique. Celui-ci permet ainsi de créer ses propres balises et attributs, permettant la réalisation claire et efficace d'un document hiérarchisant diverses notions. Dans le cadre de notre projet, nous utilisons le XML dans 2 types de documents.

Le premier est un modèle de composants très proche du modèle Fractal (http://fractal.ow2.org), lui-même basé sur l'XML (que l'on nommera méta-modele Pivot). Ce document définit l'architecture des composants, explicitant uniquement les composants, leurs interfaces et leurs rôles. La notion d'interface pouvant ici être assimilée à un point d'accès au composant, les différents rôles des interfaces décrivent les services requis et fournis par le composant. (voir Annexe document Fractal/XML).

Le second est le métamodèle Cocola (voir Annexe metamodèle Cocola) créé avec EMF (voir Outils utilisés : EMF). Il décrit les informations du premier modèle ainsi que toutes les interactions entre les constituants des composants (classes, interfaces, méthodes, attributs).

XMI

Basé sur le langage XML, XMI ou "XML Metadata Interchange" est une spécification de l'OMG qui permet de stocker et d'échanger des modèles objets sous la forme de documents XML. Dans notre TER, on génère un modèle respectant les spécifications du métamodèle. Ce modèle détaille les composants ainsi que tous les éléments qui les composent, et cela, de manière hiérarchisée. En guise de vérification, pour pouvoir s'assurer que les éléments contenus dans le modèle sont exacts, nous le sérialisons en XMI .(Voir modeleCocolaXMI)

Fractal

Lui aussi basé sur le langage XML, Fractal ADL ou "Architecture Description Language" est un langage extensible permettant de définir des architectures de composants. Il offre une syntaxe abstraite pour la définition

des attributs, les interfaces, les liens ... pour les composants.[5](Voir modele-Fractal)

2.2 Outils utilisés

Eclipse

Eclipse est un environnement de développement intégré. Comme la plupart des IDE, il offre divers conforts au programmeur : coloration syntaxique, détection d'erreurs de syntaxe, auto complétion, documentation des fonctions [2] etc. Mais sa principale caractéristique vient de son développement axé autour de la notion de plugin. En effet, grâce à ce système de plugin, nous avons pu aisément accéder à un ensemble de fonctionnalités qui se sont avérées être très pratiques. Nous avons pu utiliser entre autres les plugins EMF, Erca, Minjava que nous allons détailler ci-après.

EMF

EMF ou "Eclipse Modeling Framework" est un Framework permettant la réalisation d'applications basé sur une structure de modèle de données. Il permet donc, après création d'un métamodèle, de générer un ensemble de classes, utilisables pour réaliser une application. Ce métamodèle définit les règles que devront respecter les modèles résultants. Ainsi, nous avons créé un métamodèle respectant les exigences de nos encadrants (le métamodèle Cocola). La première phase de notre TER est donc un projet EMF basé sur ce métamodèle. Nous avons utilisé EMF pour générer un ensemble de classes et méthodes permettant la création de modèles décrivant les composants logiciels [3].

MinJava

MinJava est un plugin Eclipse réalisé par Jean Rémy Falleri, doctorant travaillant actuellement au LIRMM dans le Génie Logiciel. Ce plugin permet de réaliser une opération dite de rétro-conception (ou "reverse engineering"). En fournissant un package (.jar) ou un ensemble de fichiers ByteCode (.class), Minjava va générer un modèle décrivant l'ensemble des éléments (classes, méthodes, attributs etc.) Durant notre TER, nous avons été amenés à utiliser MinJava. Cependant, nous avons finalement préféré créer notre propre système de "reverse engineering" plus épuré, et en corrélation parfaite avec nos attentes (basé sur le package reflect [10]).

Erca

Autre plugin réalisé par Jean Rémy Falleri, Erca est un framework qui facilite l'utilisation de l'analyse de concepts formels et dispose d'une technique de groupement ordonné.

2.3 Cadre

Dans ce qui suit, nous énoncerons les divers facteurs formant le cadre de notre projet. Nous parlerons ainsi des réunions avec nos encadrants, les réunions du groupe de TER pour finir sur l'organisation globale que nous avons adopté en fonction des circonstances.

Réunions encadrants

Nous avons eu des réunions avec nos encadrants environ une fois par semaine. Dans un premier temps, nous avons discuté des concepts et défini plus précisément le cadre de notre travail. Une fois celui-ci clairement établi et compris par tous, les réunions se sont plus orientées vers une exposition de notre état d'avancement et la résolution de problèmes spécifiques.

Réunions de travail

Une fois le cadre de travail clairement défini, nous nous sommes scindés en deux groupes. Nous avons ainsi principalement travaillé en binôme, chaque binôme étant chargé de l'une des deux phases du projet. Cependant, nos parties étant étroitement liées, nous n'avons pas manqué de nous réunir régulièrement (à raison d'une fois par semaine). Nous y avons donc fait l'état de nos remarques ou problèmes, mais nous nous sommes aussi concertés sur les différents points du projet à aborder durant la prochaine réunion avec les encadrants.

Organisation

L'objectif principal du TER, consiste en la réalisation d'une application capable de hiérarchiser un ensemble de composants et ainsi de pouvoir effectuer des remplacements de composants. Nous avons par ailleurs dut effectuer ces réalisation en respectant les exigences précisées dans le cahier des charges. Ce dernier s'est d'ailleurs affiné tout au long du projet, principalement d au fait que le domaine des composants logiciels est relativement récent.

De par les outils utilisés, il nous a été impossible de travailler sur les ordinateurs de la faculté, nous avons donc travaillé sur nos ordinateurs personnels. Durant ce TER, nous avons été suivi par Marianne Huchard et Chouki Tibermacine. Trois semaines après le début du TER, Jean Rémy Falleri, doctorant spécialisé dans le domaine du Génie Logiciel est venu se joindre à nos encadrants.

Une fois les concepts du projet distinctement fixés, nous avons travaillé de manière autonome. Les rendez-vous avec les encadrants ont permis de s'assurer que toutes les contraintes et exigences demandées étaient respectées. Si l'ambiance générale durant les réunions a toujours été très bonne, celle-ci n'a cependant pas du tout influencée l'intransigeance de nos encadrants quant à leurs attentes sur notre travail.

Nous avons d par ailleurs analyser le travail existant réalisé par Nour Alhouda Aboud, durant son stage de Master 2 [1]. Le choix nous a été donné de reprendre ou non son travail. Après analyse du travail existant (voir partie : Travail effectué, Premiers pas), il nous a semblé plus intéressant de recréer totalement le projet.

Nous avons également travaillé en nous aidant d'articles de recherches. Ces articles, qui nous ont été fourni pour la plupart au début du TER nous ont permis de nous familiariser avec les notions clés des composants logiciels. Plus tard dans le projet, ils nous ont de surcroît servi de référence pour vérifier nos travaux (Voir articles utilisés).

Le projet fut partagé en deux phases. Ces dernières étant fortement dépendantes l'une de l'autre, il nous a fallu travailler en synergie pour mener à bien le projet. Le projet que nous avons réalisé sera très probablement repris par d'autres personnes. En conséquence, nous avons tâché de présenter et commenter au mieux le programme réalisé. Cela donne au groupe des repères pour facilement se replonger dans le projet, ou permet encore de prévoir les futurs éléments à coder. Ainsi, le TER fait appel à des qualités d'analyse, de rigueur et de clarté dans un environnement de collaboration.

3 Analyse de la problématique

Dans le répertoire de composants étudié, il y a plusieurs composants incluant des services fournis et des services requis. Si le répertoire a une structure plate, l'utilisateur doit revoir le répertoire entièrement, ou se restreindre à un ensemble de services obtenus par une requête basée sur des mots-clefs, mais n'a pas d'information claire pour retrouver le composant pertinent pour une connexion ou une substitution. Dans le développement ou la maintenance de tâches, il serait utile de savoir de manière efficace :

1. Quelles sont les relations de substitution entre composants
2. Quels sont les composants se comportant de la même manière, pour trouver rapidement une substitution approximative
3. Quels sont les nouveaux composants qu'il serait pertinent d'ajouter au référentiel afin de rendre plus faciles les futurs travaux de développements et les tâches de remplacement.

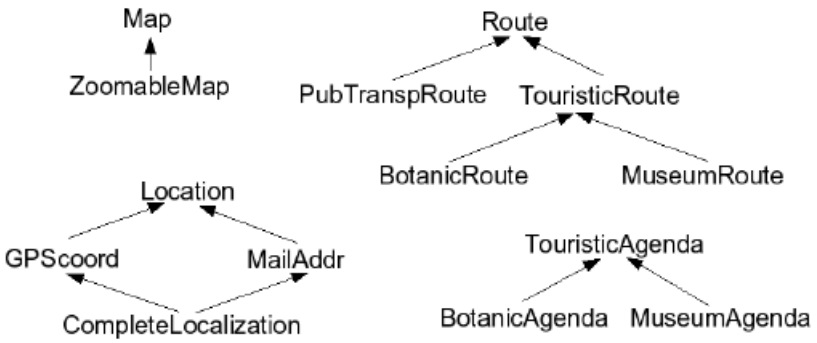


Figure 1 :*Hierarchie de type des composants de Route*

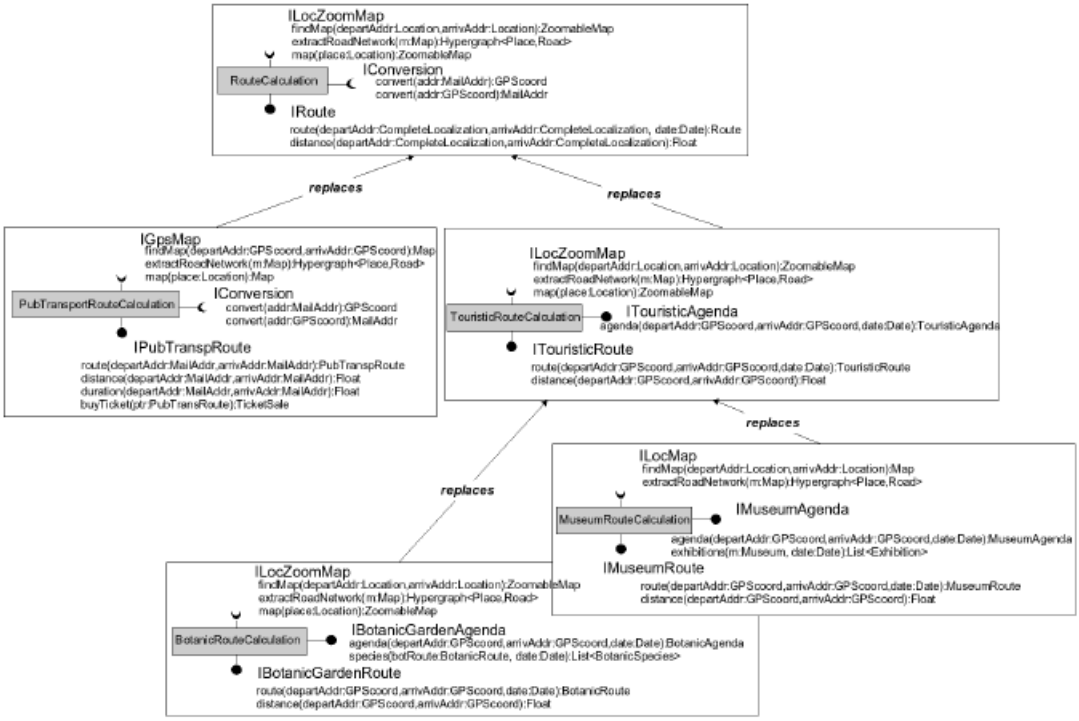


Figure 2 :*Classification des composants de Route*

Examinons la structure complète que nous proposons de construire afin de répondre à notre problématique. Premièrement, il s'agit de réaliser une classification des composants pouvant se remplacer d'un point de vue sémantique. Par exemple , MuseumRoutealculation est classifié comme pouvant se substituer à TouristicRouteCalculation.

Deuxièmement, nous voyons émerger de nouveaux types de composants comme RouteCalculation qui généralise tous les autres composants implémentés. En définissant un ensemble composé de types plus général plutôt que plus spécialisé, cela nous garanti que cet ensemble sera plus réutilisable et modifiable.

Nous représenterons les composants sur le principe de substitution qui établit qu'un composant peut se substituer à un autre s'il requiert moins et qu'il fournit plus.

- **Requiert moins signifie en substitution :**
  - généralisation des paramètres d'entrée dans les fonctions fournies (ou retrait des paramètres d'entrée)
  - généralisation des paramètres de sortie dans les fonctions requises
  - on a moins de fonctions requises
  - on a moins d'interfaces requises
- **Fournit plus signifie en substitution**
  - spécialisation des paramètres de sortie dans les signatures fournies
  - spécialisation, des paramètres d'entrée dans les signatures requises(ajouter des paramètres)
  - on a plus de fonctions fournies
  - on a plus d'interfaces fournies

L'application de cette règle de substitution utilisée dans l'encodage, dit que ne pas avoir un p requis peut être considéré comme la possible substitution d'avoir un p requit. Si un composant a p, il peut être remplacé par un composant qui ne nécessite pas p parce qu'il possède deja cette opération.

4 Travail effectué

Cette section présente le travail réalisé durant notre projet. Pour chaque phase du projet, nous exposons d'abord la démarche qui nous a amené au développement final puis nous explicitons le travail ainsi obtenu.

4.1 Premiers pas

Les premiers entretiens avec nos encadrants nous ont permis d'appréhender au mieux les notions relatives aux "composants logiciels". Pour se faire, les encadrants nous ont nottament fourni un article traitant des composants logiciels ainsi que sa traduction partielle en français [7, 6]. Nous nous sommes par ailleurs appuyé sur la thèse de Luc Fabresse pour nous familiariser avec le vocabulaire spécifique aux composants logiciels [4]. Une fois ces principes assimilés, nous avons pu discuter ensemble de la manière dont il fallait aborder le sujet. En effet, si le sujet du TER semble être défini, la manière pour parvenir à son développement n'était quant à elle, pas du tout fixée. Le sujet devait à la base correspondre à la continuité du travail effectué par une stagiaire au LIRMM : Nour Alhouda Aboud. Cette dernière avait donc réalisé un mémoire [1], détaillant son programme appliquant une méthode de construction d'annuaires de composants par classification. Il nous a alors été demandé d'analyser le travail effectué afin d'en déduire si nous devions ou non reprendre son projet. Après avoir examiné en détail le programme, nous avons, avec l'accord de nos encadrants, décidé de ne pas reprendre le programme et d'en réaliser un entièrement par nous même. En effet, si l'efficacité du programme n'est pas à remettre en doute, la manière dont le programme a été réalisé ne correspondait cependant pas à certaines attentes requises.

4.2 Analyse de l'existant

Dans son programme, Nour Alhouda Aboud utilise 2 sortes de modèles (au format XML), qu'elle a réalisé manuellement. Le premier permet la description de chaque composant (1 document par composant). (Voir Annexe description de composant Nour) Le second sert à hiérarchiser les interactions entre les divers éléments d'un composant (voir Annexe description des données Nour Alhouda Aboud).

Il est nécessaire que le premier modèle soit réalisé manuellement par le développeur du module ou par la personne souhaitant diviser le module en composants. En effet, ce document ne se base sur aucune donnée, et le frac-

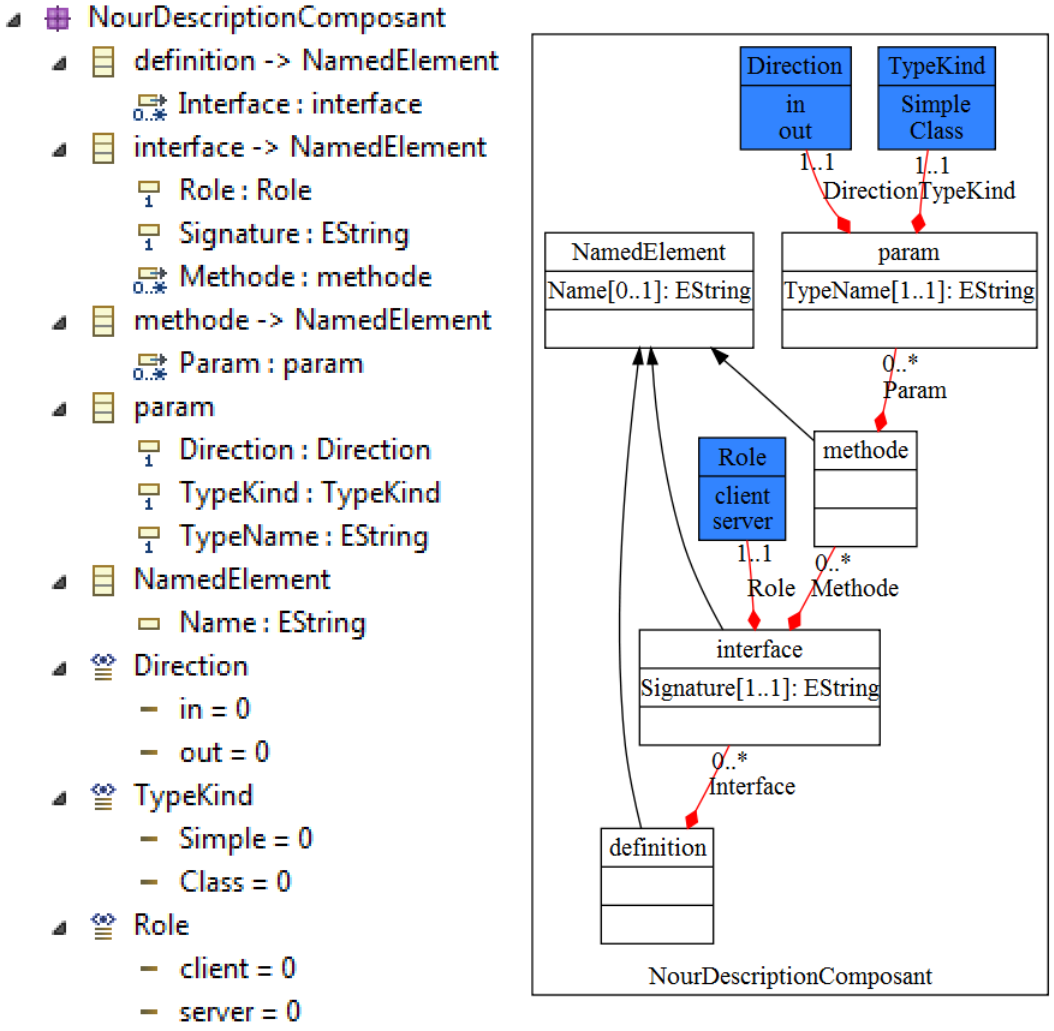
tionnement en composants est totalement subjectif. Néanmoins, le modèle décrivant les interactions des éléments d'un module est automatisable.

Pour clarifier notre analyse et mettre en contraste le travail de Nour Alhouda Aboud et le nôtre, nous allons faire ici la comparaison des métamodèles associés aux modèles réalisés par Nour et les nôtres. Nous nous sommes basés sur les modèles qu'elle a créés pour en déduire ses métamodèles. Ci-dessous un exemple de modèle de composant créé par Nour Alhouda Aboud :

Listing 1 – exemple de modèle de composants Nour

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<definition name="AdultOrder1">
    <interface name="I1" role="client" signature="Interface1">
        <methode Name="Create">
            <param Direction="in" TypeKind="Class" TypeName="Information"/>
            <param Direction="in" TypeKind="Class" TypeName="BankIdentity"/>
            <param Direction="in" TypeKind="Class" TypeName="Country"/>
            <param Direction="out" TypeKind="Class" TypeName="Customer"/>
        </methode>
        <methode Name="Modify">
            <param Direction="in" TypeKind="Class" TypeName="Information"/>
            <param Direction="out" TypeKind="Class" TypeName="Customer"/>
        </methode>
    </interface>
    <interface name="I6" role="client" signature="Interface6">
        <methode Name="Add">
            <param Direction="in" TypeKind="Class" TypeName="Product"/>
            <param Direction="out" TypeKind="Simple" TypeName="void"/>
        </methode>
        <methode Name="remove">
            <param Direction="in" TypeKind="Class" TypeName="Product"/>
            <param Direction="out" TypeKind="Simple" TypeName="void"/>
        </methode>
    </interface>
</definition>
```

Nous avons alors à partir de ces exemples, créé le métamodèle correspondant.



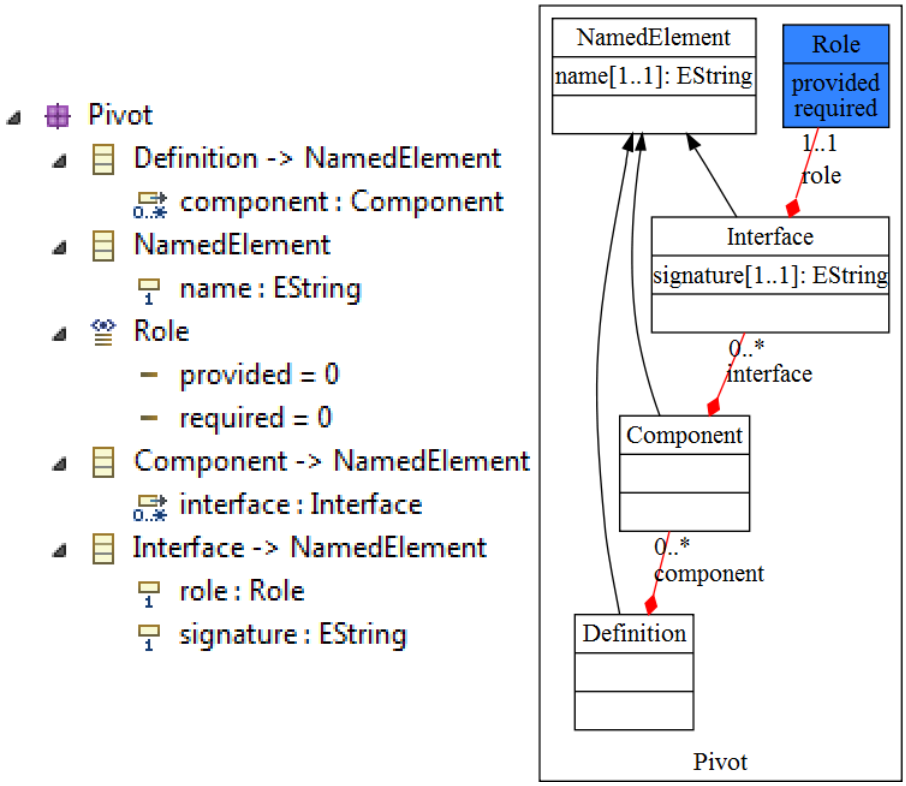
Métamodèle Composant Nour Alhouda Aboud

Ainsi, le méta-modèle de description des composants de Nour Alhouda Aboud comprend un élément définition dont l'attribut name correspond au nom du composant. Un composant contient un ensemble d'interfaces (les services des composants). Ces interfaces possèdent un nom, un rôle (si le service est requis ou fournis) et une signature (qui correspond au nom complet intégrant le nom du package). Une interface comprend un ensemble de méthodes. Chacune de ces méthodes inclut des paramètres. Ces paramètres possèdent une direction (”in” pour un paramètre d'entrée, ”out” pour un paramètre de sortie), un TypeKind indiquant si le paramètre est une classe ou une primitive (”Class” et ”Simple”) et un TypeName correspondant au

nom du type.

Le méta-modèle de description des données du module contient un ensemble de types nommés (la correspondance entre les 2 modèles se fait entre cet attribut et l'attribut TypeName du premier modèle). Chaque type peut contenir un ensemble de sous-types et cela de manière récursive.

Nous avons donc étudié les méthodes utilisées par Nour Alhouda Aboud pour décrire les composants et leurs interactions. Nous en avons déduit plusieurs faiblesses et avons donc, avec l'aide de nos encadrants, réalisé notre métamodèle de description des composants.



Métamodèle Pivot

Notre métamodèle de description des composants que nous appellerons par la suite méta-modèle pivot a été créé en se basant sur celui de Fractal ADL. Ce langage de description de composants tend à se répandre de plus en plus dans la communauté des développeurs. Le méta-modèle pivot reprend donc la majeure partie des principes de Fractal ADL en limitant toutefois

quelques-unes de ses fonctionnalités. Par exemple, le méta-modèle pivot ne permet pas, contrairement à Fractal ADL, de définir un composant comme la spécialisation d'un autre.

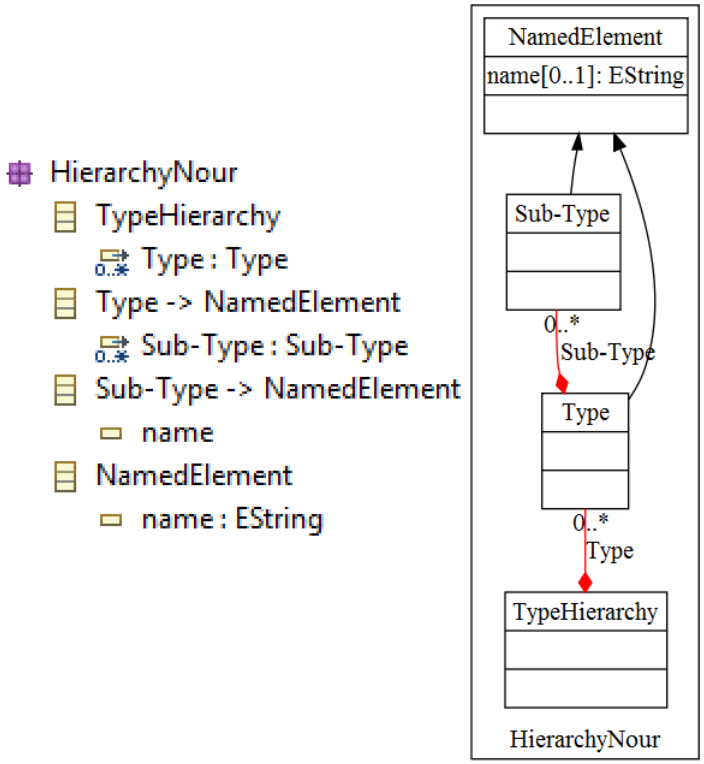
Par rapport au méta-modèle de Nour Alhouda Aboud, le méta-modèle pivot ne permet la description que d'un seul composant par modèle. Les roles "client" et "server" des interfaces deviennent "required" et "provided". Enfin, la partie décrivant l'ensemble des méthodes et les attributs les composants a été supprimée du modèle. En effet, ces éléments peuvent être directement extraits du module programmé à décomposer.

Le métamodèle hiérarchisant les données réalisé par Nour Alhouda Aboud n'est de surcroît, pas optimal. Son modèle ne spécifie à aucun moment qu'un sous-type est lui mème un type. D'après le méta-modèle de descriptions, seul un paramètre peut être typé, alors que dans notre modèle tous les éléments peuvent l'être. L'imbrication des sous-types ne se prète pas du tout à de possibles mises à jour. En effet, soit 2 types A et B, avec A qui étend B. Si l'on souhaite ajouter un type intermédiaire entre le type A et B, cela n'est pas aisé. Les modèles n'ayant pas été réalisés à partir d'un méta-modèle mais manuellement, ce genre de considération n'a pas d ètre pris en compte. Ci-dessous un exemple de modèle hiérarchique créé par Nour Alhouda Aboud.

Listing 2 – exemple de modèle hiérarchique de types, Nour Alhouda Aboud

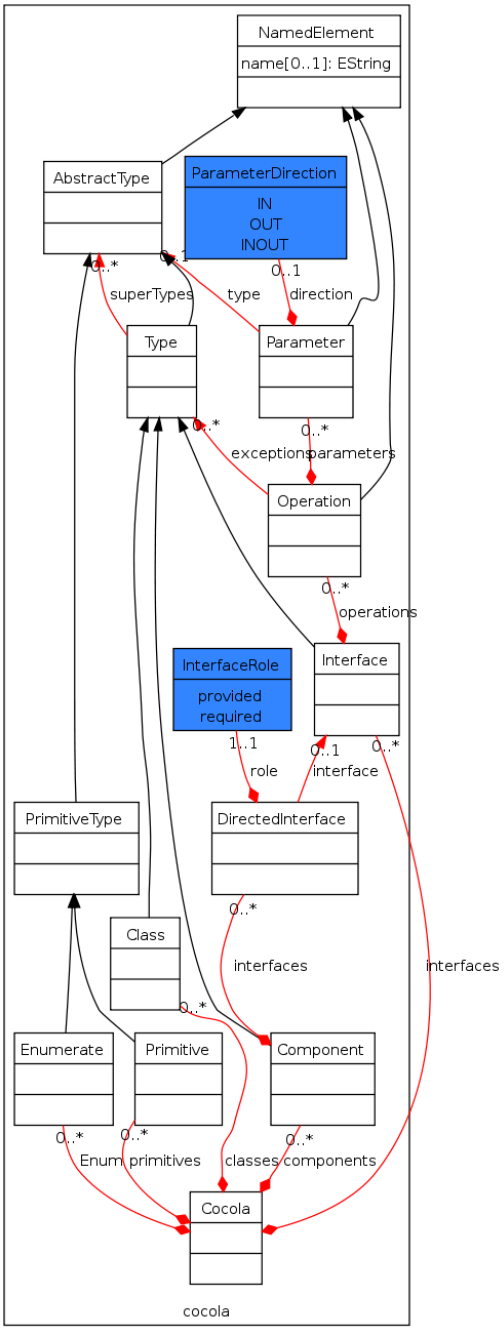
```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <type-hierarchy>
3   <type name="name">
4     <sub-type name="name">
5       <sub-type name="name">
6       </sub-type>
7     </sub-type>
8   </type>
9 </type-hierarchy>
```

A partir de celui-ci, nous en avons déduit le métamodèle correspondant.



Metamodèle Hierarchie Nour Alhouda Aboud

Au lieu de réaliser un métamodèle équivalent au métamodèle de description des données de Nour Alhouda Aboud, nous avons élaboré un méta-modèle général. Ce métamodèle va nous permettre de produire un modèle, en fusionnant les informations récupérées par l'analyse du module programmé et le modèle de composants. Nous l'appellerons par la suite métamodèle cocola (en référence au nom de l'ancien projet). Nous avons donc au final, limité au strict minimum les informations décrites par le métamodèle de composants. Cela représente en effet les seuls facteurs à établir manuellement. L'ensemble des autres informations se retrouvent modélisées via le métamodèle cocola.



Métamodèle cocola

Les deux représentations ci-dessus du métamodèle cocola explicitent clairement la structure des modèles générés. Ainsi, un modèle Cocola disposera d'un ensemble de composants, d'interfaces et de classes, un composant est lui-mème un ensemble d'interfaces orientées. Une interface orientée est une interface auquel on a ajouté un rôle (provided ou required). Une interface est un ensemble d'opérations (ou fonctions). Ces opérations comprennent un ensemble de paramètres pouvant ètre des paramètres d'entrée (IN), de sortie (OUT) ou encore les deux simultanément (INOUT). Les composants, interfaces, classes et opérations sont tous des Types. Un Type peut posséder un ensemble de superTypes étant des types abstraits (Ce qui permet à un type d'étendre un type primitif). Les primitives et les énumérations sont des types primitifs. Ces derniers, sont des types au même rang que les classes ou les interfaces. Tous ces types appartiennent à la famille des AbstractType. Un AbstractType a la particularité d'être un élément nommable (il possède un attribut name).

A partir de ces 2 méta-modèles nous disposons de bases solides pour réaliser un programme permettant d'organiser des composants logiciels.

Comme nous l'avons précisé un peu plus tôt, il est impossible d'automatiser la création du document décrivant les composants (basé sur le méta-modèle Fractal ou pivot). Il existe néanmoins des outils, comme Fractal ADL par exemple, pour faciliter la création d'un modèle de composants. Le modèle décrivant les interactions des éléments du module programmé pris en entrée (basé sur le métamodèle Cocola) peut par contre quant à lui, être généré à l'aide d'un programme que nous avons nous même mis en place. Ainsi, notre analyseur de code source permet la réalisation du modèle. Cette analyse nous a confortés dans notre choix de ne pas reprendre le travail réalisé par Nour Alhouda Aboud. Les différences entre les méta-modèles existant et ceux que nous avons réalisé changent totalement la réalisation de notre projet.

A partir de notre analyse initiale, nous avons eu les bases pour réfléchir à la conception du programme chargé de réaliser un système d'annuaire de composants logiciels.

Véritable coeur du projet, le métamodèle Cocola a donné lieu à de nombreuses discussions et modifications au fil du projet. Ce métamodèle sert de "grammaire" (ou syntaxe abstraite) aux modèles que va générer la première phase du programme.

La partie développement du projet comprend deux grandes phases. La première, est de former le modèle précis explicitant les composants d'un programme (donné en entrée) ainsi que tous les éléments les composant, de manière organisée. La seconde, est d'analyser ce modèle pour en dégager toutes ses possibilités (Mise à jour, remplacement d'un composant etc.).

Pour mieux comprendre le processus de fonctionnement du projet dans son ensemble, un schéma récapitulatif du programme réalisé est présent ci-dessous.

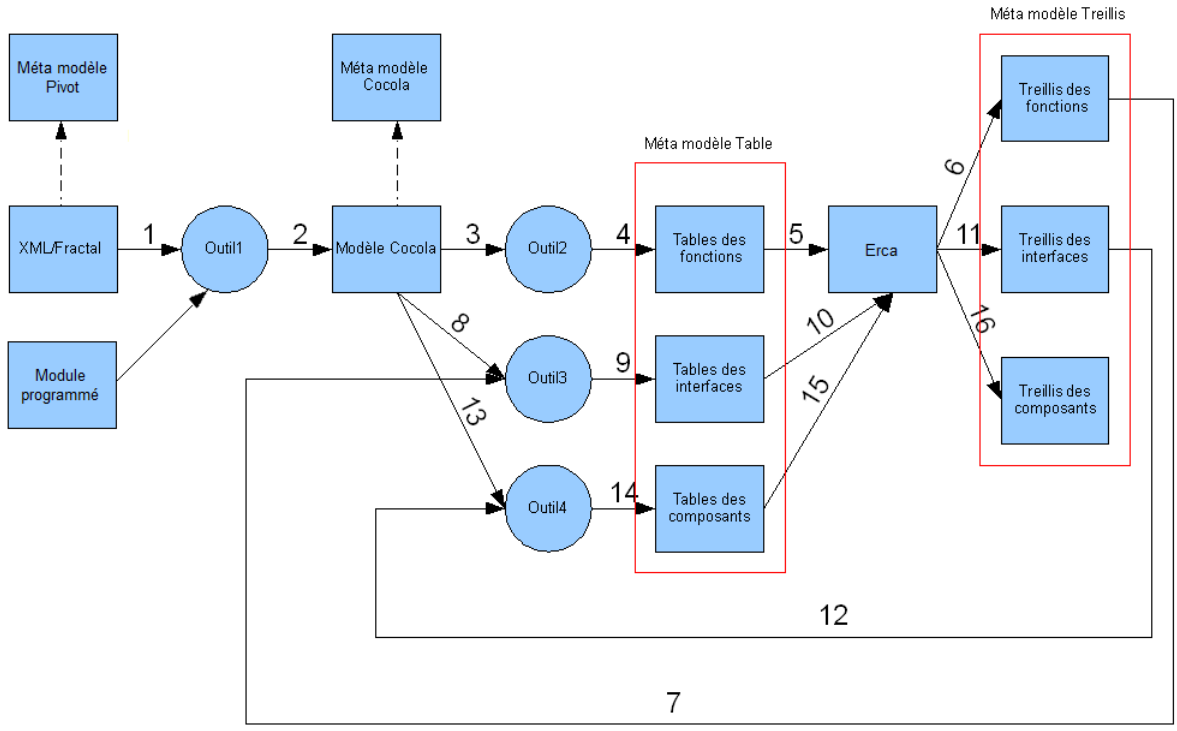


Schéma général du programme réalisé

4.3 Principe général

La phase 1 correspond à l'outil 1, la phase 2, aux outils 2,3 et 4.

L'outil 1 va donc prendre en entrée un ensemble de fichiers décrivant l'agencement des composants et leurs services. Ces fichiers sont écrits en respectant un modèle de composant similaire à Fractal ADL (Voir Annexe Fractal), écrit en XML. Notons que l'outil est aussi bien compatible avec le modèle Pivot créé qu'avec Fractal ADL. Il va en outre prendre un module développé : ensemble de fichiers ByteCode (fichiers binaires compilés). Ce module peut être une archive de type .jar ou un répertoire. L'outil va effectuer une opération de rétro-conception (ou "reverse engineering") pour retrouver l'ensemble des éléments qui le composent. En sortie, l'outil 1 va générer un modèle Cocola exposant les composants dans leurs détails (l'outil 1 aura fait la correspondance entre les fichiers de type Fractal et le ByteCode). Les classes utilisées y sont présentées, ainsi que les relations de SuperTypage associés. Les opérations contenues dans le ByteCode et les types des paramètres (en entrée et en sortie) y sont aussi présents. Ce modèle respecte le métamodèle Cocola. Nous sérialisons alors le modèle en XMI. Cela nous permet de vérifier le modèle et d'en garder une trace écrite. Le modèle sérialisé peut servir d'entrée à la seconde phase du projet, ce qui permet à l'outil 2 de ne pas réapeller un modèle déja généré. Dans le cas d'une nouvelle opération, le modèle non sérialisé sera directement utilisé.

L'outil 2 va donc prendre le modèle Cocola explicitant la hiérarchisation des composants logiciels.Notons qu'un composant contient des interfaces (notions pouvant ètre comparées à des points d'accès aux composants). Et que ces interfaces contiennent des opérations (ou fonctions). Pour pouvoir avoir une vue d'ensemble du composant, il faut logiquement partir du plus particulier et remonter pas à pas au plus général. La première étape est donc de décrire les opérations. En sortie, il va générer un tableau révélant les relations possibles au niveau des opérations (ou fonctions). Ce tableau, comme les tableaux issus des outils 3 et 4 se basent sur la méthode créée par nos encadrants.Celle-ci est, à son tour, basée sur un formalisme. Ce tableau sera donné en paramètre au plugin Erca, qui génèrera un treillis reflétant la structure ainsi dévoilée. Ensuite, de la mème façon, l'outil 3 va prendre le treillis généré antérieurement et le modèle cocola pour créer les tableaux des interfaces. Les treillis générés par Erca avec ces tableaux serviront d'entrée à l'outil 4 qui générera les tableaux des composants pour finalement obtenir les treillis des composants.

Ces treillis une fois créés peuvent directement servir à déterminer quel composant peut se substituer à un autre.

5 Détails du développement

Cette partie expose le développement réalisé durant le TER. Pour chacune des deux phases du projet, nous expliciterons tout d'abord les différentes étapes d'avancement jusqu'à aboutir à la dernière version. Nous énoncerons par ailleurs les aspects plus techniques et utiliserons enfin un exemple contrait que nous déploierons du début à la fin pour illuster le fonctionnement de notre programme.

5.1 Phase 1

L'outil 1 prend en paramètre un fichier de type modèle de composant Fractal. Fractal décrit un module avec des balises XML. Voici quelques unes des plus utilisées :

- Balise "definition" représente le module, extends permet la déscription de relations de super/sous composants
- Balise "component" correspond à un composant Attribut "name" : le nom du module
  - Attribut "name" : le nom du composant
- Balise "interface" définit un point d'accès au composant (un service du composant)
  - Attribut "name" : le nom de l'interface
  - Attribut "role" : le sens de ce service, server : le service est fourni, client : le service est requis
  - Attribut "signature" : la signature de l'interface (sa localisation dans le module)

Ci-dessous le fichier fractal de notre exemple Route.

Listing 3 – exemple Route Fractal

```
1  <?xml version="1.0" encoding="ISO-8859-1" ?>
2  <!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal
3  ADL 2.0//EN" "classpath://org/objectweb/fractal/adl/xml/basic.dtd">
4  <definition name="routeExample">
5    <component name="BotanicRoute">
6      <interface name="IBotanicGardenRoute" role="provided" signature="Service"/>
7      <interface name="IBotanicGardenAgenda" role="provided" signature="Service"/>
8      <interface name="ILocZoomMap" role="required" signature="Service"/>
9    </component>
10   <component name="MuseumRoute">
11     <interface name="IMuseumRoute" role="provided" signature="Service"/>
12     <interface name="IMuseumAgenda" role="provided" signature="Service"/>
13     <interface name="ILocMap" role="required" signature="ServiceService"/>
14   </component>
15   <component name="TouristicRoute">
16     <interface name="ITouristicRoute" role="provided" signature="Service"/>
17     <interface name="ITouristicAgenda" role="provided" signature="Service"/>
18     <interface name="ILocZoomMap" role="required" signature="Service"/>
19   </component>
20   <component name="PubTransportRoute">
21     <interface name="IGPSMap" role="required" signature="Service"/>
22     <interface name="IConversion" role="required" signature="Service"/>
23     <interface name="IPubTranspRoute" role="provided" signature="Service"/>
24   </component>
25   <component name="Route">
26     <interface name="IMap" role="required" signature="Service"/>
27     <interface name="IRoute" role="provided" signature="Service"/>
28   </component>
29 </definition>
```

Le modèle de composant Pivot est proche du modèle Fractal mais restreint quelque peu ses fonctionnalités. Ce modèle de composants doit être en accord avec le métamodèle Cocola. Il doit par ailleurs être créé manuellement par le développeur du module, ou éventuellement par l'utilisateur du programme. En effet, cette opération ne peut être automatisée. La notion de composant et d'interface étant abstraite (et donc absente dans le code), il est impossible de déterminer de manière automatique la structure de ce document.

Avec ce document, nous possédons toutes les informations "abstraites" liées aux composants et à leurs services. L'outil 1 va déduire le reste en analysant les composants dont il est question. L'outil 1 étant un projet EMF basé sur le métamodèle Cocola, une gamme de méthodes est à notre disposition. Grâce à elle, nous avons pu ajouter nos éléments (composants, interfaces, classes, opérations etc.) en analysant le document de type Fractal et les fichiers de ByteCode. Pour analyser le document de type Fractal, nous avons réalisé un "parser" XML basique, qui, à la reconnaissance d'une balise, va ajouter l'élément correspondant au modèle Cocola. Les attributs de l'élément sont ajoutés de la même façon.

Pour analyser les fichiers de ByteCode, il a premièrement fallu réaliser un système efficace de parcours de fichiers. En effet, le module à analyser pris en entrée par l'outil 1 peut être de diverses formes. Nous avons donc traité le cas o celui-ci était un fichier archive de type .jar mais aussi le cas où le module se situe dans un répertoire. Dans ce cas-là, nous effectuons un parcours des répertoires et sous-répertoires récursivement. Seules les classes se trouvant dans des fichiers de ByteCode (.class) sont ajoutées à la liste des classes. L'ajout se fait de manière à garder l'arborescence complète de la classe, à la manière des packages Java (de type repertoire1.sousrepertoire1.NomDeMaClasse.class). Si une archive de type .jar est trouvée, elle est "dézippée" et l'ensemble des classes contenues sont ajoutées. Avec l'aide d'EMF, nous engendrons une instance du modèle basé sur le métamodèle Cocola. Nous y ajoutons les éléments récupérés dans les fichiers de type Fractal.

Par ailleurs, nous disposons de l'ensemble des classes contenues dans le module programmé pris en entrée. Nous allons donc analyser ces classes, pour en dégager toutes les informations nécessaires. Nous avons pris comme convention de définir le Type void, comme superType général. Dans le langage Java, toutes les classes descendent du superType "java.lang.Object" (hormi les types primitifs). En conséquence, si une classe ne dispose pas d'autre superType que "java.lang.Object", le superType void lui sera imposé. Cette

convention permet d'avoir la racine void commune à toutes les classes. Dans une première phase, nous allons ajouter au modèle l'ensemble des classes existantes. Nous allons alors analyser les fichiers de ByteCode en utilisant le package reflect. Pour chaque classe ou interface java pouvant être une interface (au sens service du terme), nous associons une interface. Ensuite, pour chaque interface créée (correspondant à une classe ou interface java), nous récupérons l'ensemble de ses opérations. Enfin, pour chaque opération, on récupère l'ensemble des types de ses paramètres (d'entrées et de sorties).

Dans une deuxième phase, nous allons regarder les relations de spécialisations entre les classes. Si une classe ou une interface en étend une autre, nous l'indiquons dans le modèle. L'avantage issu des fonctions mises à notre disposition par EMF est qu'il va lier dynamiquement les éléments entre eux. Lors de l'analyse, on va signaler que tel type étend tel autre.

Cette manière de faire assure une bonne cohérence des données. En effet, un élément peut faire référence à un autre élément présent dans le modèle. On évite ainsi les contradictions. Dans le cas o une classe n'est pas présente dans le modèle, le programme effectue une résolution pour déterminer son package d'origine. Lorsque la procédure est terminée, nous fournissons le modèle cocola en entrée à l'outil 2. Dans l'exemple Route qui réunit la plupart des cas particuliers que l'on peut rencontrer, un seul est resté insoluble. En effet, comme on peut le voir sur le schéma modelisant l'exemple 3, CompleteLocalization dispose d'un héritage multiple sur les classes GPSCoord et MailAddr. Le langage Java ne gère pas l'héritage multiple directement (contrairement à C++ par exemple). Ainsi, nous aurions pu "simuler" un héritage multiple (avec une interface et une composition) mais le resultat n'aurait probablement pas été le même. Nous avons préféré ne pas représenter cet héritage multiple dans les sources java et de l'ajouter simplement dans le modèle. Ci-dessous le modèle généré de l'exemple Route (sérialisé en XMI).

Listing 4 – exemple Route, modèle Cocola généré

```
1  <?xml version="1.0" encoding="ASCII"?>
2  <cocola:Cocola xmi:version="2.0"
3  xmlns:xmi="http://www.omg.org/XMI" xmlns:cocola="cocola">
4    <components name="BotanicRouteCalculation">
5      <interfaces interface="//@interfaces.0"/>
6      <interfaces interface="//@interfaces.1"/>
7      <interfaces role="required" interface="//@interfaces.2"/>
8    </components>
9    <components name="MuseumRouteCalculation">
10     <interfaces interface="//@interfaces.3"/>
11     <interfaces interface="//@interfaces.4"/>
```

```
12      <interfaces role="required" interface="//@interfaces.5"/>
13    </components>
14    <components name="TouristicRouteCalculation">
15      <interfaces interface="//@interfaces.6"/>
16      <interfaces interface="//@interfaces.7"/>
17      <interfaces role="required" interface="//@interfaces.2"/>
18    </components>
19    <components name="PubTransportRouteCalculation">
20      <interfaces role="required" interface="//@interfaces.8"/>
21      <interfaces role="required" interface="//@interfaces.9"/>
22      <interfaces interface="//@interfaces.10"/>
23    </components>
24    <primitives name="void"/>
25    <primitives name="java.util.List"/>
26    <primitives name="java.lang.Float"/>
27    <primitives name="float"/>
28    <classes name="BotanicAgenda" superTypes="//@classes.16"/>
29    <classes name="BotanicRoute" superTypes="//@classes.17"/>
30    <classes name="BotanicSpecies" superTypes="//@primitives.0"/>
31    <classes name="CompleteLocalisation" superTypes="//@classes.5 //@classes.8"/>
32    <classes name="Exhibition" superTypes="//@primitives.0"/>
33    <classes name="GPSCoord" superTypes="//@classes.7"/>
34    <classes name="HyperGraph" superTypes="//@primitives.0"/>
35    <classes name="Location" superTypes="//@primitives.0"/>
36    <classes name="MailAddr" superTypes="//@classes.7"/>
37    <classes name="Map" superTypes="//@primitives.0"/>
38    <classes name="Museum" superTypes="//@primitives.0"/>
39    <classes name="MuseumAgenda" superTypes="//@classes.16"/>
40    <classes name="MuseumRoute" superTypes="//@classes.17"/>
41    <classes name="PubTranspRoute" superTypes="//@classes.14"/>
42    <classes name="Route" superTypes="//@primitives.0"/>
43    <classes name="Ticketsale" superTypes="//@primitives.0"/>
44    <classes name="TouristicAgenda" superTypes="//@primitives.0"/>
45    <classes name="TouristicRoute" superTypes="//@classes.14"/>
46    <classes name="ZoomableMap" superTypes="//@classes.9"/>
47    <classes name="Date" superTypes="//@primitives.0"/>
48    <interfaces name="IBotanicGardenRoute">
49      <operations name="route">
50        <parameters name="out0-0" type="//@classes.1" direction="OUT"/>
51        <parameters name="in0-0-0" type="//@classes.5"/>
52        <parameters name="in0-0-1" type="//@classes.5"/>
53        <parameters name="in0-0-2" type="//@classes.19"/>
54      </operations>
55      <operations name="distance">
56        <parameters name="out0-1" type="//@primitives.3" direction="OUT"/>
57        <parameters name="in0-1-0" type="//@classes.5"/>
58        <parameters name="in0-1-1" type="//@classes.5"/>
59      </operations>
60    </interfaces>
61    <interfaces name="IBotanicGardenAgenda">
62      <operations name="agenda">
63        <parameters name="out1-0" type="//@classes.0" direction="OUT"/>
64        <parameters name="in1-0-0" type="//@classes.5"/>
65        <parameters name="in1-0-1" type="//@classes.5"/>
66        <parameters name="in1-0-2" type="//@classes.19"/>
67      </operations>
68      <operations name="species">
69        <parameters name="out1-1" type="//@primitives.1" direction="OUT"/>
70        <parameters name="in1-1-0" type="//@classes.1"/>
71        <parameters name="in1-1-1" type="//@classes.19"/>
72      </operations>
73    </interfaces>
```

```
74  <interfaces name="ILocZoomMap">
75    <operations name="findMap">
76      <parameters name="out2-0" type="//@classes.18" direction="OUT"/>
77      <parameters name="in2-0-0" type="//@classes.7"/>
78      <parameters name="in2-0-1" type="//@classes.7"/>
79    </operations>
80    <operations name="extractRoadNetWork">
81      <parameters name="out2-1" type="//@classes.6" direction="OUT"/>
82      <parameters name="in2-1-0" type="//@classes.9"/>
83    </operations>
84    <operations name="map">
85      <parameters name="out2-2" type="//@classes.18" direction="OUT"/>
86      <parameters name="in2-2-0" type="//@classes.7"/>
87    </operations>
88  </interfaces>
89  <interfaces name="IMuseumRoute">
90    <operations name="route">
91      <parameters name="out3-0" type="//@classes.12" direction="OUT"/>
92      <parameters name="in3-0-0" type="//@classes.5"/>
93      <parameters name="in3-0-1" type="//@classes.5"/>
94      <parameters name="in3-0-2" type="//@classes.19"/>
95    </operations>
96    <operations name="distance">
97      <parameters name="out3-1" type="//@primitives.3" direction="OUT"/>
98      <parameters name="in3-1-0" type="//@classes.5"/>
99      <parameters name="in3-1-1" type="//@classes.5"/>
100   </operations>
101 </interfaces>
102 <interfaces name="IMuseumAgenda">
103   <operations name="agenda">
104     <parameters name="out4-0" type="//@classes.11" direction="OUT"/>
105     <parameters name="in4-0-0" type="//@classes.5"/>
106     <parameters name="in4-0-1" type="//@classes.5"/>
107     <parameters name="in4-0-2" type="//@classes.19"/>
108   </operations>
109   <operations name="exhibitions">
110     <parameters name="out4-1" type="//@primitives.1" direction="OUT"/>
111     <parameters name="in4-1-0" type="//@classes.10"/>
112     <parameters name="in4-1-1" type="//@classes.19"/>
113   </operations>
114 </interfaces>
115 <interfaces name="ILocMap">
116   <operations name="findMap">
117     <parameters name="out5-0" type="//@classes.9" direction="OUT"/>
118     <parameters name="in5-0-0" type="//@classes.7"/>
119     <parameters name="in5-0-1" type="//@classes.7"/>
120   </operations>
121   <operations name="extractRoadNetWork">
122     <parameters name="out5-1" type="//@classes.6" direction="OUT"/>
123     <parameters name="in5-1-0" type="//@classes.9"/>
124   </operations>
125 </interfaces>
126 <interfaces name="ITouristicRoute">
127   <operations name="route">
128     <parameters name="out6-0" type="//@classes.17" direction="OUT"/>
129     <parameters name="in6-0-0" type="//@classes.5"/>
130     <parameters name="in6-0-1" type="//@classes.5"/>
131     <parameters name="in6-0-2" type="//@classes.19"/>
132   </operations>
133   <operations name="distance">
134     <parameters name="out6-1" type="//@primitives.3" direction="OUT"/>
135     <parameters name="in6-1-0" type="//@classes.5"/>
```

```
136       <parameters name="in6-1-1" type="//@classes.5"/>
137     </operations>
138   </interfaces>
139   <interfaces name="ITouristicAgenda">
140     <operations name="agenda">
141       <parameters name="out7-0" type="//@classes.16" direction="OUT"/>
142       <parameters name="in7-0-0" type="//@classes.5"/>
143       <parameters name="in7-0-1" type="//@classes.5"/>
144       <parameters name="in7-0-2" type="//@classes.19"/>
145     </operations>
146   </interfaces>
147   <interfaces name="IGPSMap">
148     <operations name="findMap">
149       <parameters name="out13-0" type="//@classes.9" direction="OUT"/>
150       <parameters name="in13-0-0" type="//@classes.5"/>
151       <parameters name="in13-0-1" type="//@classes.5"/>
152     </operations>
153     <operations name="extractRoadNetWork">
154       <parameters name="out13-1" type="//@classes.6" direction="OUT"/>
155       <parameters name="in13-1-0" type="//@classes.9"/>
156     </operations>
157     <operations name="map">
158       <parameters name="out13-2" type="//@classes.9" direction="OUT"/>
159       <parameters name="in13-2-0" type="//@classes.7"/>
160     </operations>
161   </interfaces>
162   <interfaces name="IConversion">
163     <operations name="convert">
164       <parameters name="out9-0" type="//@classes.5" direction="OUT"/>
165       <parameters name="in9-0-0" type="//@classes.8"/>
166     </operations>
167     <operations name="convert">
168       <parameters name="out9-1" type="//@classes.8" direction="OUT"/>
169       <parameters name="in9-1-0" type="//@classes.5"/>
170     </operations>
171   </interfaces>
172   <interfaces name="IPubTranspRoute">
173     <operations name="route">
174       <parameters name="out10-0" type="//@classes.13" direction="OUT"/>
175       <parameters name="in10-0-0" type="//@classes.8"/>
176       <parameters name="in10-0-1" type="//@classes.8"/>
177     </operations>
178     <operations name="duration">
179       <parameters name="out10-1" type="//@primitives.3" direction="OUT"/>
180       <parameters name="in10-1-0" type="//@classes.8"/>
181       <parameters name="in10-1-1" type="//@classes.8"/>
182     </operations>
183     <operations name="buyTicket">
184       <parameters name="out10-2" type="//@classes.15" direction="OUT"/>
185       <parameters name="in10-2-0" type="//@classes.13"/>
186     </operations>
187     <operations name="distance">
188       <parameters name="out10-3" type="//@primitives.3" direction="OUT"/>
189       <parameters name="in10-3-0" type="//@classes.8"/>
190       <parameters name="in10-3-1" type="//@classes.8"/>
191     </operations>
192   </interfaces>
193 </cocola:Cocola>
```

5.2 Phase 2

Cette phase consiste à déduire les relations entre les différentes fonctions, interfaces et composants d'une librairie de composants. Afin d'expliciter au mieux ces outils, nous suivrons étape par étape leurs utilisations autour de l'exemple : la librairie de composant "Route" vue dans la description de l'outil 1.

5.2.1 Classification des fonctions

Un composant possède des opérations, ces opérations sont contenues dans des interfaces. Les interfaces peuvent être fournies ou requises. Nous cherchons donc dans un premier temps à trouver les nouvelles signatures pouvant se substituer à celles déjà existantes pour les operations.

5.2.2 Classification des signatures requises (Operations) :

En se restreignant aux signatures des opérations requises, le principe de substitution implique pour le substitut :

- La spécialisation des paramètres d'entrée dans les signatures requises.
- La généralisation des paramètres de sortie dans les signatures requises[1]

Treillis de l'opération findMap initial

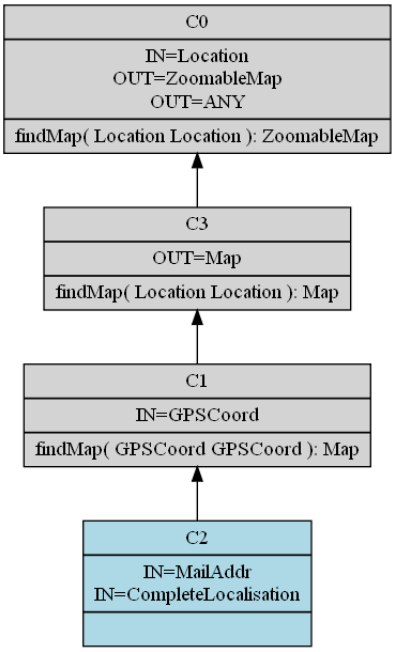
Ici nous avons trois concepts (signatures de fonctions) différents.

	IN=GPSCoord	IN=MailAddr	IN=CompleteLocalisation	IN=Location	OUT=Map	OUT=ZoomableMap	OUT=ANY
findMap(Location Location) : ZoomableMap				×		×	×
findMap(Location Location) : Map				×	×	×	×
findMap(GPSCoord GPSCoord) : Map	×			×	×	×	×

[1]Dans notre modèle, nous supposons que nous avons un seul paramètre de sortie.

Table de substituabilité des signatures de l'operation findMap

On constate que pour chaque signature, pour tous les paramètres d'entrée, on ajoute les types qui spécialisent ceux déjà connus par la signature initiale, et pour les paramètres de sortie, on ajoute ceux qui généralisent les autres.



Treillis de l'opèration findMap après recherche des substitutions de paramètres

Ici le concept C2 est nouveau, nous constatons que ses paramètres d entrée sont : MailAddr et CompleteLocalization. Dans la hiérarchie de type, MailAddr spécialise localisation. CompleteLocalization spécialise MailAddr et GPSCoord, on peut donc créer un nouveau concept pour findMap avec ces paramètres d'entrée.

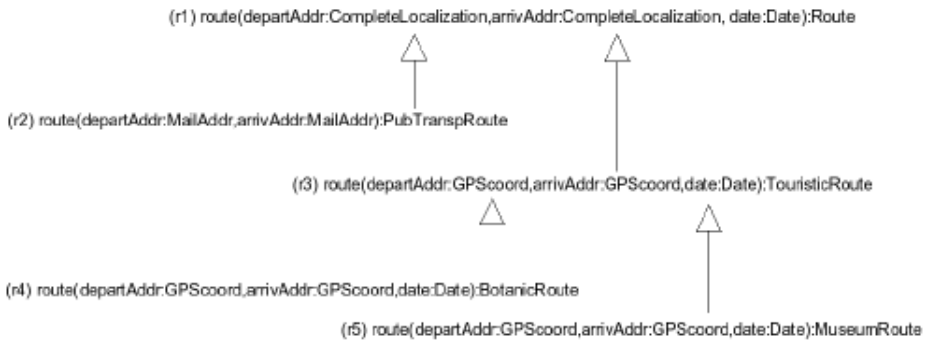
5.2.3 Classification des signatures fournies (Operations)

En se restreignant aux signatures des opérations fournies, le principe de substitution implique pour le substitut :

- La généralisation des paramètres d'entrée dans les signatures fournies (ou le retrait des paramètres d'entrée)
- La spécialisation des paramètres de sortie dans les signatures requises (dans notre modèle, nous supposons que nous avons un seul paramètre de sortie)

Concernant le retrait d'un paramètre, nous avons besoin de représenter une signature ne contenant pas le paramètre. Pour ce faire, nous ajoutons !p $\neg$*(non p)commeuntype*, *signifiantque*!*psesubstitueàp*.

Dans la figure 3, nous observons que nous avons ajouté le type !date. D'un point de vue fourni, si le paramètre date est supprimé de la première occurrence, nous perdons les propriétés initiales de construction de la fonction. D' un point de vue requis, nous perdons la factorisation avec CompLocalization et les paramètres de la méthode route.



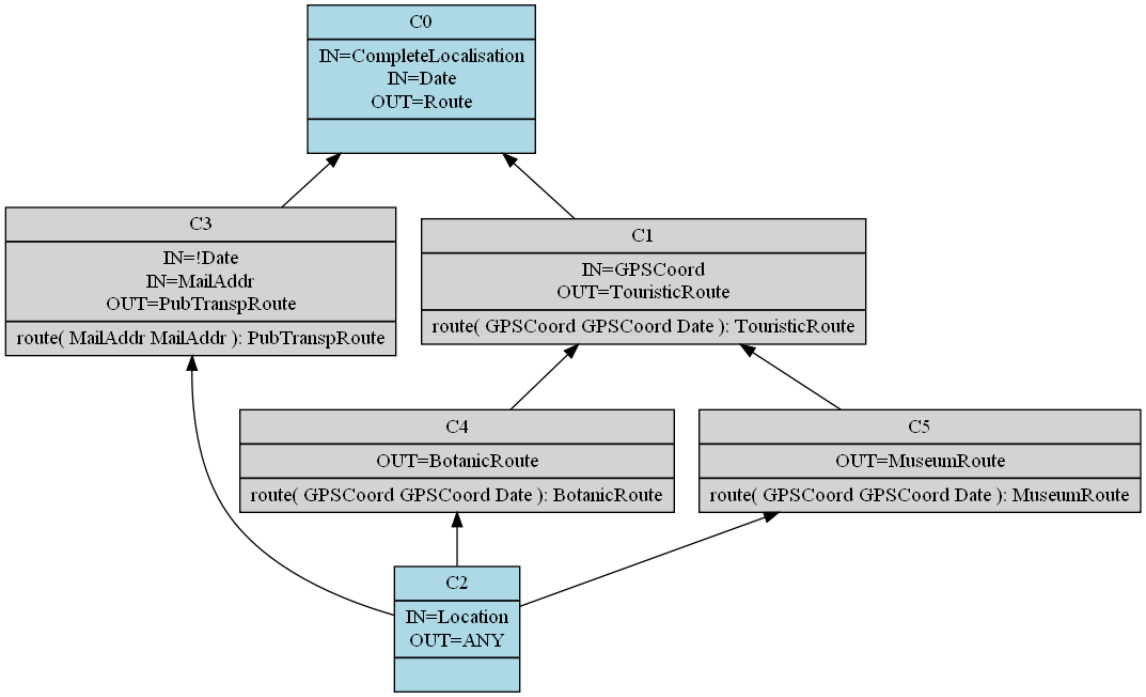
Treillis de l'operation route

Ici nous avons cinq concepts (signatures de fonctions) differents.

	IN=CompleteLocalisation	IN=GPSCoord	IN=Date	IN= !Date	IN=Location	IN=MailAddr	OUT=TouristicRoute	OUT=Route	OUT=BotanicRoute	OUT=ANY	OUT=MuseumRoute	OUT=PubTranspRoute
route(GPSCoord GPSCoord Date) : BotanicRoute	×	×	×				×	×	×			
route(GPSCoord GPSCoord Date) : MuseumRoute	×	×	×				×	×			×	
route(GPSCoord GPSCoord Date) : TouristicRoute	×	×	×				×	×				
route(MailAddr MailAddr) : PubTranspRoute	×		×	×		×		×				×

Table de substituabilité des signatures de l'oération route

On constate que pour chaque signature, pour tous les paramètres d'entrée, on ajoute les types qui généralisent ceux déjà connus par la signature initiale, et pour les paramètres de sortie, on ajoute ceux qui spécialisent les autres.



Treillis de l'opération route après recherche des substitutions de paramètres

On constate l'apparition de deux nouveaux concepts : C0 et C2. C0 prend en paramètre d'entrée un type Location. Ce concept est la factorisation de tous les autres concepts, c'est à dire que tous les concepts représentant les diverses signatures peuvent se substituer à la signature reprèsentée par le C0. C2 a pour paramétre d'entrée un type Location et en sortie le type ANY. Ce concept doit pouvoir se substituer à tous les autres concepts. Un premier problème nous est apparu lors de la déduction de la signature de ce concept. En effet, pour assurer que ce concept spécialise tous les autres on doit choisir le type de sortie le plus spécifique, or dans notre exemple cela voudrait dire qu'un type pourrait spécialiser les types BotanicRoute et MuseumRoute, cependant un tel type n'existe pas dans notre hiérarchie. C'est pour cela que nous avons introduit un nouveau type abstrait pouvant spécialiser tous les types contrairement à void qui les généralise tous.

5.2.4 Classification des Interfaces

L' assemblage des composants se base majoritairement sur une connexion d' interfaces compatibles, à un niveau d'abstraction plus élevé que les simples fonctionnalités. Les interfaces pourraient être organisées par spécialisation assez naturellement, en considérant les fonctionnalités incluses. Pourtant, en utilisant les abstractions découvertes lors de la construction des treillis des signatures de fonctionnalités, on va pouvoir découvrir des abstractions encore plus pertinentes au regard de la substituabilité ou de la connexion.

Dans la première version de notre logiciel, nous avions élaboré une seule classification pour les interfaces fournies ainsi que pour les interfaces requises. Cette manière de procéder n'était malheureusement pas efficace lorsque nous nous retrouvions confronté à une librairie comportant un grand nombre d' interfaces. Les tables générées étant trop volumineuses, nous avions une explosion combinatoire lors de la génération des treillis. Nous avons donc dû trouver un moyen pour pallier à ce problème.

5.2.5 Calcul des composantes connexes

Une interface est connectée à une ou plusieurs signatures, et une signature est incluse par une ou plusieurs interfaces. On peut s'apercevoir que l'ensemble nous donne un graphe biparti.

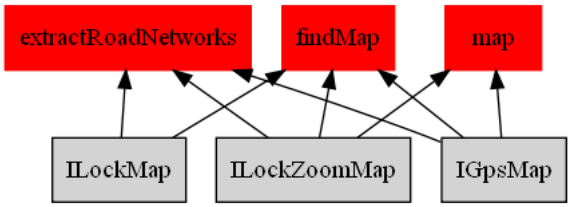


Figure 1 : *Composante connexe Interfaces Requise 1*

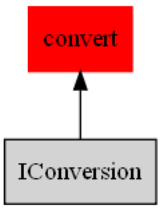


Figure 2 : *Composante connexe Interfaces Requise 2*

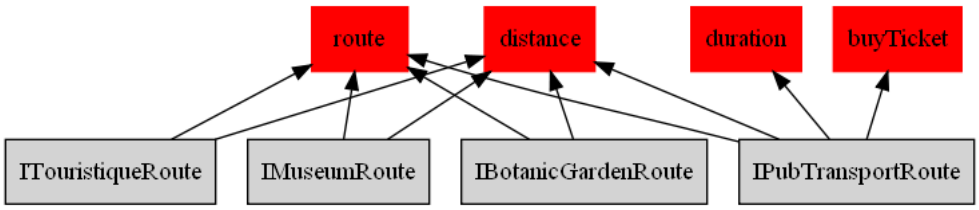


Figure 3 : *Composante connexe Interfaces Fournie 1*

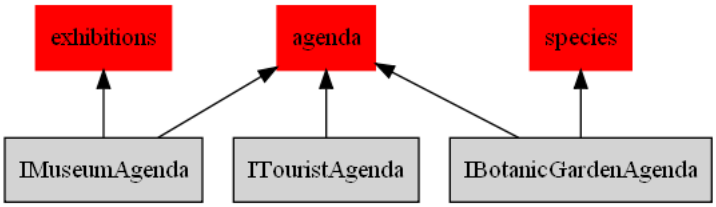


Figure 4 : *Composante connexe Interfaces Fournie 2*

Dans la figure 2, on peut s'apercevoir que l'interface IConversion possède seulement une signature *convert*, et que cette signature est incluse seulement par une interface. Cette interface est donc indépendante de toutes les autres, il n'est donc pas nécessaire de l'inclure dans le même treillis avec toutes les autres interfaces requises. Nous allons donc générer autant de treillis que de composantes connexes.

5.2.6 Classification des Interfaces Fournies

Voici les tables de substitution des interfaces fournies. Nous avons deux tables représentant les composantes connexes calculées précédemment. Les colonnes **C*i*** représentent les interfaces, et les lignes **L*i*** représentent les signatures des fonctions inclues dans les interfaces, ainsi que les signatures abstraites factorisées résultantes des treillis des signatures.
Nous avons une relation *(**Ci** × **Li**)* si :

1. *L'interface **Ci** possède directement la signature **Li***
2. *Le concept représentant la signature **Li** est un concept parent d'une des signatures inclues dans l'interface **Ci**. Par exemple, la signature **route(CompleteLocalisation Date) : Route** est la signature de plus haut niveau du treillis **route**(lien hyper). Donc toutes les signatures inclues dans les interfaces pourront se substituer à cette signature.*

	agenda(Location Date) : ANY	agenda(GPSCoord GPSCoord Date) : BotanicAgenda	agenda(GPSCoord GPSCoord Date) : MuseumAgenda	agenda(GPSCoord GPSCoord Date) : TouristicAgenda	exhibitions(Museum Date) : java.util.List	species(Route Date) : java.util.List	species(BotanicRoute Date) : java.util.List
ITouristicAgenda				×			
IMuseumAgenda			×	×	×		
IBotanicGardenAgenda		×		×			×

Figure 5 : *Table de substituabilité des interfaces fournie de la figure 4 (5.2.5)*

	route(Location) : ANY	route(CompleteLocalisation Date) : Route	route(MailAddr MailAddr) : PubTranspRoute	route(GPSCoord GPSCoord Date) : TouristicRoute	route(GPSCoord GPSCoord Date) : MuseumRoute	route(GPSCoord GPSCoord Date) : BotanicRoute	distance(Location) : float	distance(CompleteLocalisation) : float	distance(GPSCoord GPSCoord) : float	distance(MailAddr MailAddr) : float	duration(Location) : float	duration(MailAddr MailAddr) : float	buyTicket(Route) : ANY	buyTicket(PubTranspRoute) : Ticketsale
ITouristicRoute		×		×				×	×					
IBotanicGardenRoute		×		×		×		×	×					
IPubTranspRoute		×	×					×		×		×		×
IMuseumRoute		×		×	×			×	×					

Figure 6 : *Composante connexe interfaces fournie 3*(5.2.5)

Nous allons maintenant genérér les treillis des tables ci-dessus :

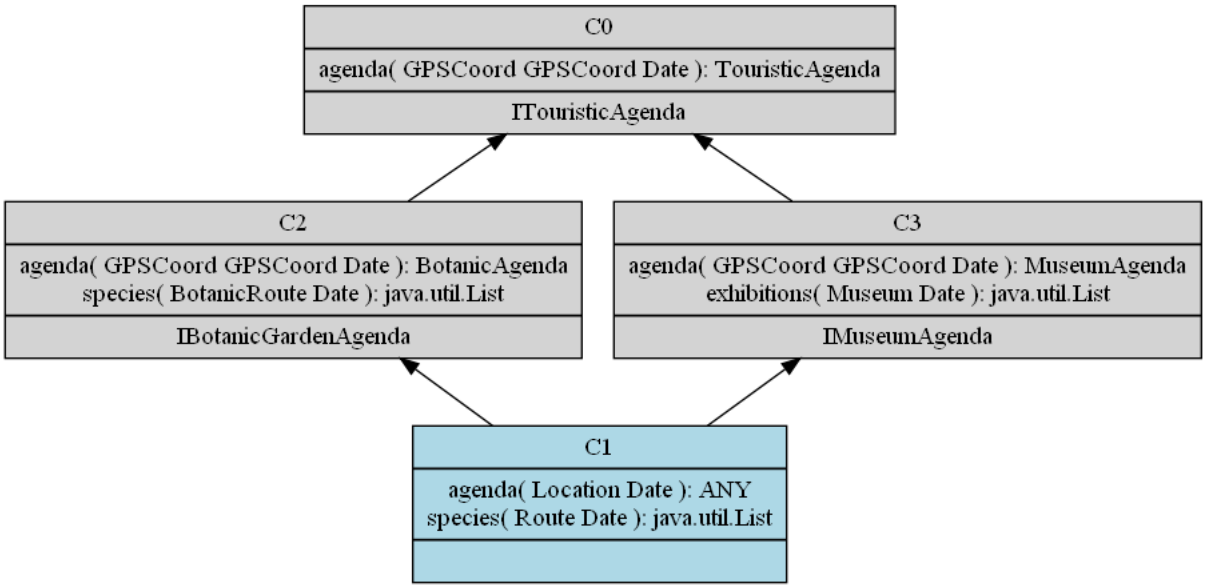


Figure 7 : Treillis de l'interface fournie de la composante figure 4

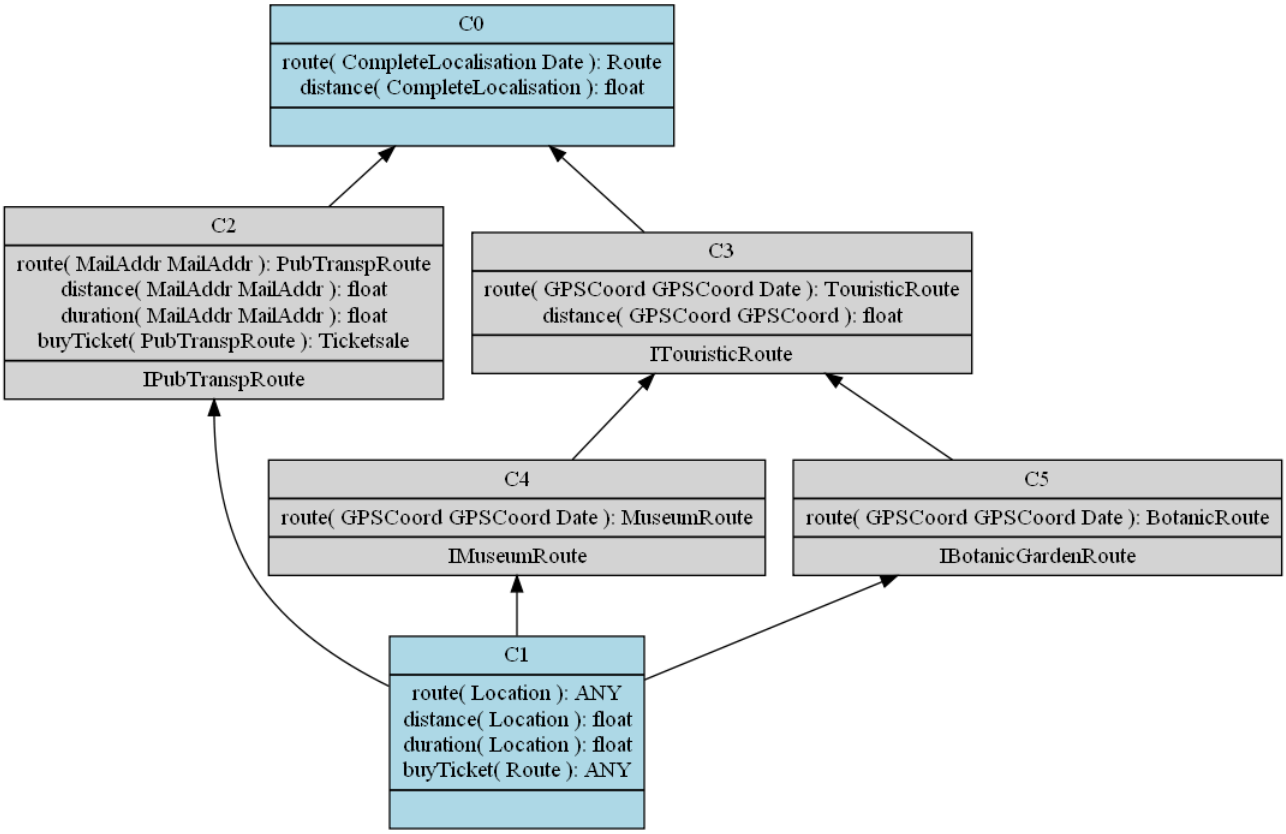


Figure 8 : Treillis de l'interface fournie de la composante figure 3

Ces treillis nous donnent une classification de substituabilité des interfaces. Prenons par exemple le treillis figure 8, le concept C4 et C5 correspondant respectivement aux interfaces IMuseumRoute et IBotanicGardenRoute peuvent se substituer au concept C3 correspondant à l'interface ITouristiqueRoute.

Nous voyons émerger deux nouveaux concepts. Le concept C0 et C1, représentant une factorisation possible des interfaces appartenant à la même composante connexe. En effet, ces nouvelles interfaces pourront être fournies à l'utilisateur afin qu'il puisse générer des interfaces plus spécialisées ou plus générales.
La génération de l'interface représenté par le concept C0 nous donnerait les signatures des fonctionnalités suivantes : *(route (CompleteLocalization CompleteLocalization Date) : Route* ***et*** *distance (CompleteLocalization CompleteLocalization) :float)*. Ce serait l'interface à laquelle toutes les autres pourraient se substituer.

5.2.7 Classification des interfaces requises

La génération des tables des interfaces requises s'effectue de la même manière que celles des interfaces fournies. Mis à part le fait que l'on réintroduise la même notion de non appartenance vu dans la génération des tables des signatures fournies. L'interface ILockMap ne possède pas de signature *map*. On en déduit donc que cette interface peut alors effectuer cette fonctionnalité.

	convert(CompleteLocalisation) : Location	convert(Location) : CompleteLocalisation	!convert	convert(GPSCoord) : MailAddr	convert(MailAddr) : GPSCoord
IConversion		×		×	×

Figure 9 : *Table de substituabilite des Interfaces Requise de la Figure 2*(**??**)

	findMap(CompleteLocalisation) : Map	!findMap	findMap(Location Location) : Map	findMap(GPSCoord GPSCoord) : Map	findMap(Location Location) : ZoomableMap	extractRoadNetWork(ZoomableMap) : HyperGraph	!extractRoadNetWork	extractRoadNetWork(Map) : HyperGraph	map(CompleteLocalisation) : Map	!map	map(Location) : Map	map(Location) : ZoomableMap
ILocMap			×		×			×	×	×	×	×
ILocZoomMap					×			×				×
IGPSMap			×	×	×			×			×	×

Figure 10 : *Table de substituabilité des interfaces requises de la figure 1*5.2.5

Nous allons maintenant générer les treillis des tables ci-dessus :

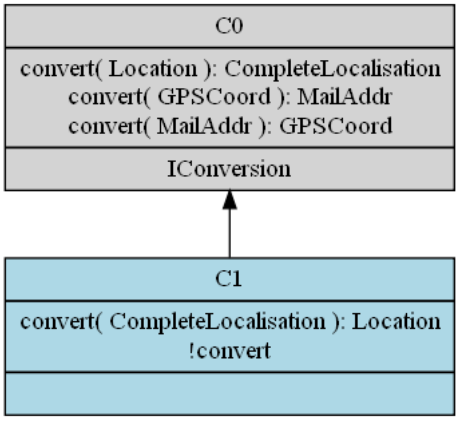


Figure 11 : Treillis de l'interface requise de la composante figure 2

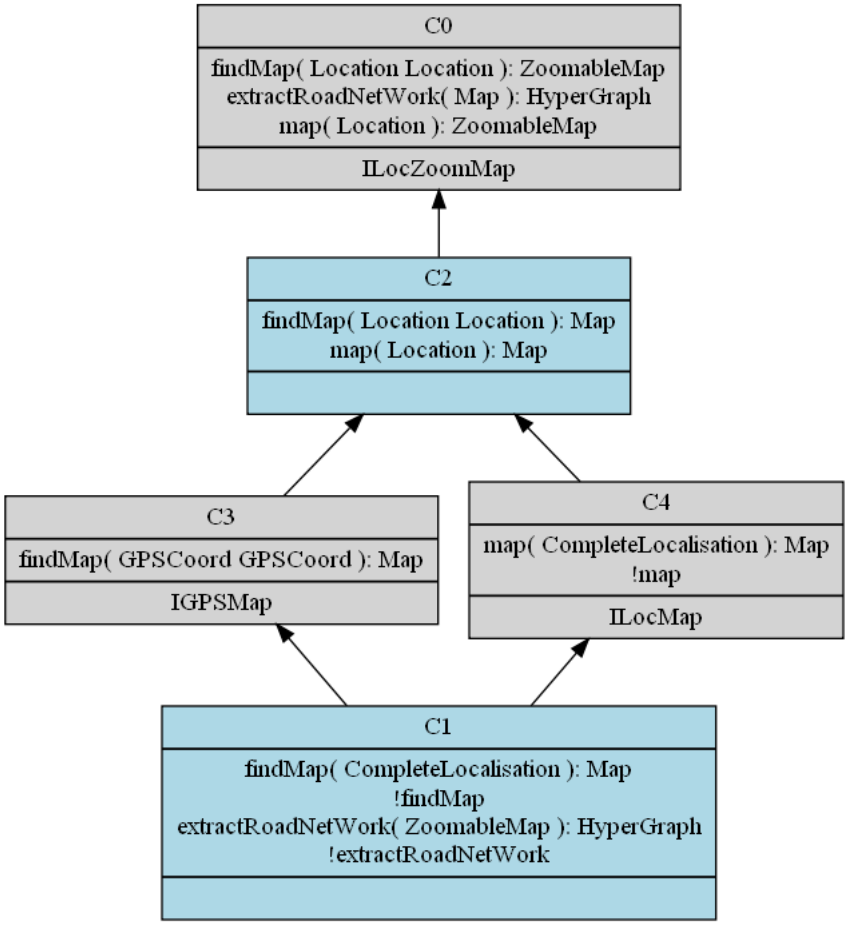


Figure 12 : Treillis de l'interface requise de la composante figure 1

L'interprétation du treillis des interfaces requises est la même que celle du treillis des interfaces fournies ainsi que la génération des nouvelles interfaces émergentes. La table figure 13 est un récapitulatif de toutes les nouvelles interfaces trouvées dans les différents treillis calculés auparavant.

0-InterfaceProC1	agenda(Location Date) : ANY,species(Route Date) : java.util.List, exhibitions(Museum Date) : java.util.List
1-InterfaceProC1	route(Location) : ANY,distance(Location) : float,duration(Location) : float, buyTicket(Route) : ANY
1-InterfaceProC0	route(CompleteLocalisation Date) : Route,distance(CompleteLocalisation) : float
0-InterfaceReqC1	()
0-InterfaceReqC2	findMap(Location) :Map, map(Location) : Map, extractRoadNetWork(Map) : HyperGraph
1-InterfaceReqC1	()

Figure 13 : nouvelle interface

5.2.8 Classification des composants

Nous allons d'abord proposer une solution pour construire le treillis des composants. La technique utilisée est la même que celle utilisée pour les interfaces. Les treillis des interfaces permettent d'enrichir la description du cadre formel qui sera utilisée lors de la construction des treillis des composants. Nous montrerons ensuite les utilisations possibles de ces treillis.

Les composants sont décrits par leurs interfaces requises et fournies. Cette information peut être organisée par spécialisation. Les composants peuvent bénéficier de la spécialisation à la fois des relations entre les interfaces existantes et celles obtenues à partir des treillis des interfaces requises et fournies, sans oublier le fait qu'un composant ne possède pas d'interface requise. On obtient ainsi un enrichissement de la description des composants et une classification plus précise.

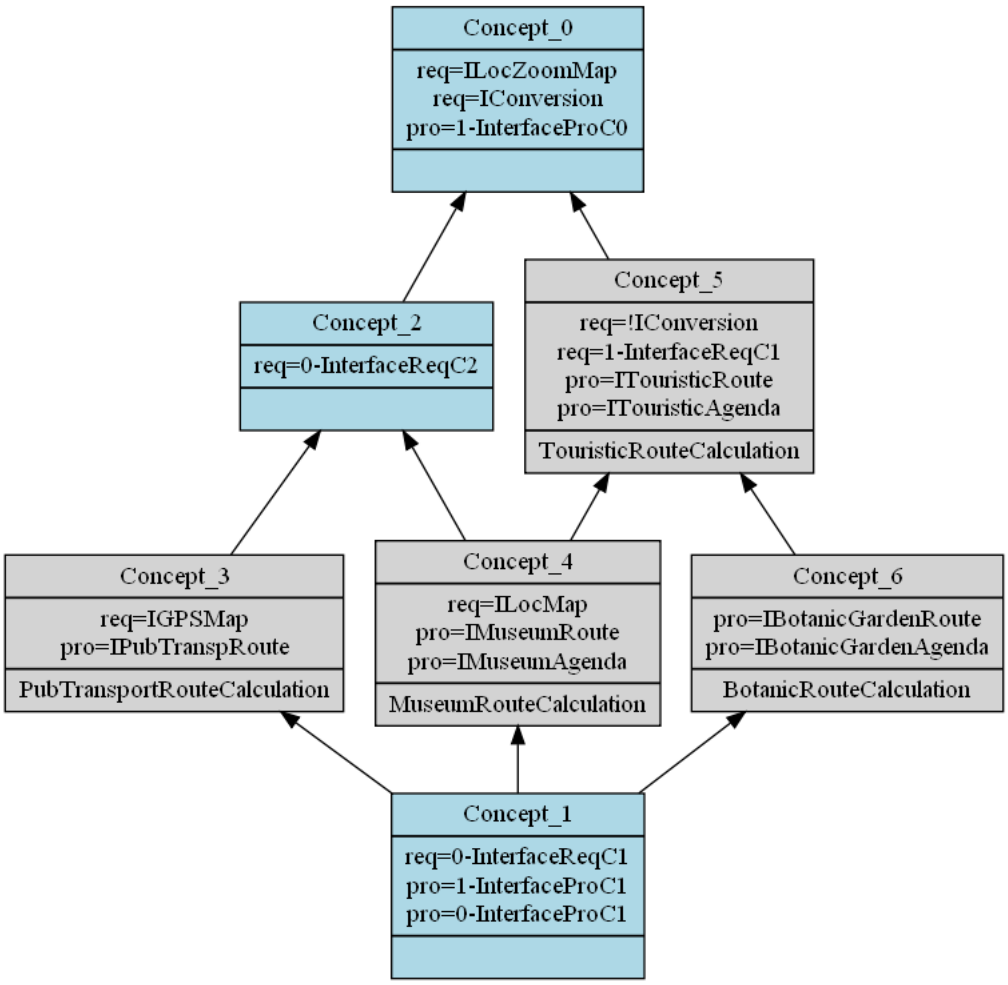
La première étape consiste à repérer les ensembles de composants (comme pour les interfaces 5.2.5). Les composants sont ainsi reliés par des fonctions qui sont elles-mêmes inclues dans d'autres composants. Dans le cas de

notre exemple, nous pouvons observer une seule composante connexe. Nous générerons donc un seul treillis.

- ***Un composant requiert une interface :***
  Soit **C** un composant, **Ireq** une interface requise et **Li** le treillis correspondant à líinterface **Ireq**. Nous utilisons aussi la notion de non-appartenance pour les interfaces requises. Si un composant ne possède pas l'interface **Ireq** ainsi quíaucune des interfaces appartenant à **L*i***. Nous en déduisons que **C** est déjà capable de faire les fonctionnalités données par **Ireq**. Nous avons donc une relation *(**C** × **!Ireq**)* et *(**C** × **Ireq**). Sinon nous avons une relation (**C** × **Ireq**) si :*
  - **Ireq** est déclarée par **C**
  - Le concept de **Ireq** dans **L*i*** est parent d'au moins un concept représentant une interface déclarée par **C**.
- ***Un composant fournit une interface :***
  Soit **C** un composant, **Ipro** une interface et **L*i*** le treillis correspondant à líinterface **Ipro**.
  *nous avons une relation (**C** × **Ipro**) si :*
  - **Ipro** est déclarée par **C**
  - Le concept de **Ipro** dans **L*i*** est parent d'au moins un concept représentant une interface déclarée par **C**.

	req=ILocZoomMap	req=0-InterfaceReqC1	req=0-InterfaceReqC2	req=IGPSMap	req= !IConversion	req=IConversion	req=1-InterfaceReqC1	req=ILocMap	pro=ITouristicRoute	pro=1-InterfaceProC1	pro=1-InterfaceProC0	pro=ITouristicAgenda	pro=0-InterfaceProC1	pro=IPubTranspRoute	pro=IMuseumRoute	pro=IMuseumAgenda	pro=IBotanicGardenRoute	pro=IBotanicGardenAgenda
TouristicRouteCalculation	×				×	×	×		×		×	×						
PubTransportRouteCalculation	×		×	×		×					×			×				
MuseumRouteCalculation	×		×		×	×	×	×	×		×	×			×	×		
BotanicRouteCalculation	×				×	×	×		×		×	×					×	×

Figure 13 : Table de substituabilité des composants



Treillis Composent

Dans ce treillis, nous pouvons voir l'apparition de trois nouveaux concepts : C0, C1 et C2. C0 possède deux interfaces requises, IlocZoomMap et Iconversion, et une interface fournie 1-InterfaceProC0. 1-InterfaceProC0 est l'interface fournie la plus générale, tous les composants peuvent l'utiliser. ILocZoomMap et Iconversion sont des interfaces requises et peuvent être aussi connectées à chaque composant. Ce concept est donc le plus général, il est la factorisation de tous les autres.

C2 généralise les concepts 3 et 4. De la même manière, cela implique que ses interfaces requises et fournies généralisent celles utilisées par les concepts

3 et 4. Quant à C1, il se trouve en bas de treillis. Il en est de même pour les interfaces ou les fonctions. Ce concept peut donc être substitué par tous les autres, il est le concept le plus specialisé.

5.3 Implementation

Dans cette section, nous allons brièvement parler de l'implémentation de la seconde phase du projet. Elle consiste en la construction de différentes tables et treillis déduits de l'analyse des divers composants, de leurs interfaces et leurs fonctions. Nous allons mettre en exergue le parallèle entre la partie théorique et la partie pratique.

Diagramme UML

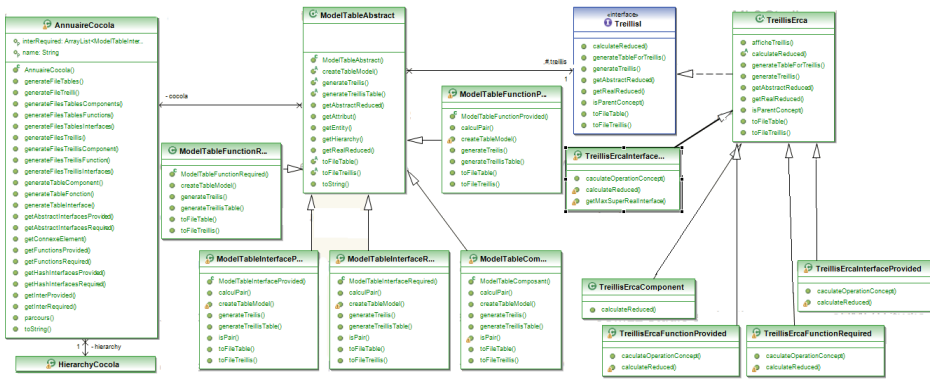


Diagramme Uml de la phase 2

HierarchyCocola

La classe HierarchyCocola se base sur le modèle Cocola. Elle va faciliter l'utilisation future de la hiérarchie de ce modèle, en l'intégrant dans une structure de données.

Les Attributs Principaux :

Listing 5 – Attributs HierarchyCocola

```
1    //Le model Cocola correspondant la description de la
         librairie de composant
2    private Cocola model;
3
4    //Ces deux HashMap nous permettent d'avoir  les sous-Type
         et  les super-Type de chaques Type
```

```
5       private HashMap<String,ArrayList<String>>
            hierarchieTypeDown;
6       private HashMap<String,ArrayList<String>>
            hierarchieTypeUp;
7
8       //une classifcation direct des elements fournis et requis
9       private HashMap<String,ArrayList<String>>
            interfacesRequired;
10      private HashMap<String,ArrayList<String>>
            interfacesProvided;
11      private HashMap<String,ArrayList<StrOperation>>
            fonctionsProvided;
12      private HashMap<String,ArrayList<StrOperation>>
            fonctionsRequired;
13      private HashMap<String,ArrayList<String>>
            componentsProvided;
14      private HashMap<String,ArrayList<String>>
            componentsRequired;
15
16
17      //represente un graphe biparti entre les fonctions et les
            interfaces
18      private HashMap<String,ArrayList<String>>
            functiontoRequiredInterface;
19      private HashMap<String,ArrayList<String>>
            functiontoProvidedInterface;
20
21      //represente un graphe biparti entre les fonctions et les
            composants
22      private HashMap<String,ArrayList<String>>
            componentstoFunctions;
23      private HashMap<String,ArrayList<String>>
            functionstoComponents;
```

AnnuaireCocola

La classe AnnuaireCocola est la classe principale de la seconde phase du projet. Elle regroupe l'ensemble des fonctionnalités fournit par le logiciel. Et permet la connexion entre les outils 2, 3 et 4. (4.2).

Les Attributs Principaux :

Listing 6 – Attributs AnnuaireCocola

```
1
2
3
4       private HierarchyCocola hierarchy;
5
6       //Pour chaque nom de fonction, nous avons associe un
            Treillis
7       private HashMap<String,ModelTableFunctionRequired>
            array_fonctionsRequired;
8       private HashMap<String,ModelTableFunctionProvided>
            array_fonctionsProvided;
```

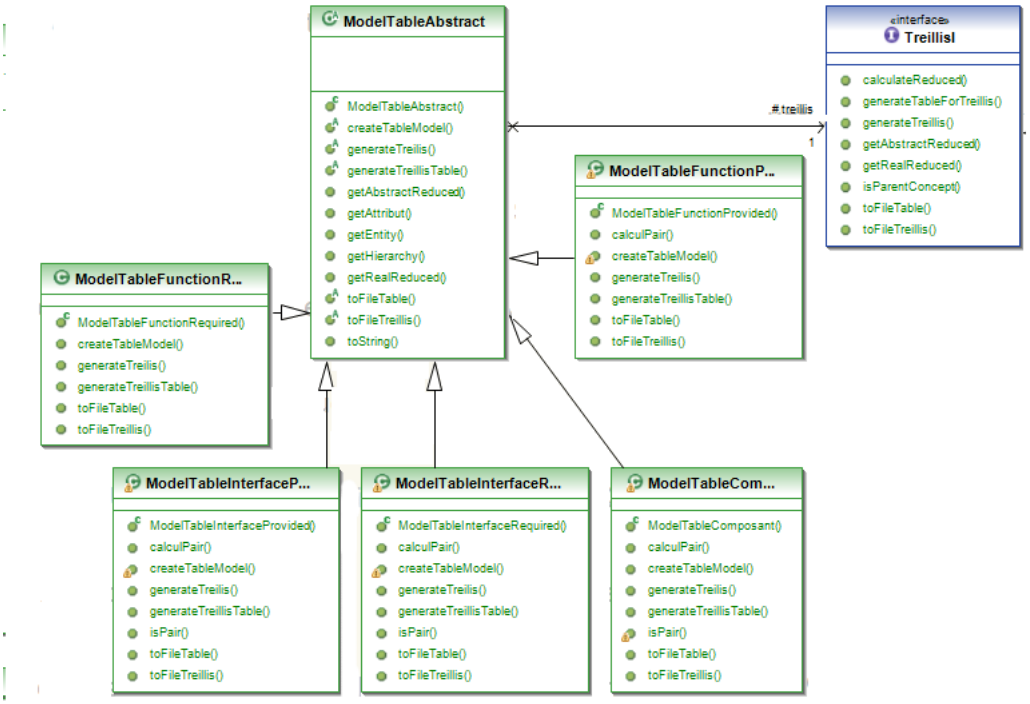
```
9
10      //Pour chaques interface, nous avons associe un Treillis
11      private HashMap<String,ModelTableInterfaceRequired>
            hashInterRequired;
12      private HashMap<String,ModelTableInterfaceProvided>
            hashInterProvided;
13
14      //Pour chaques cmoposant, nous avons associe un Treillis
15      private HashMap<String,ModelTableComposant> hashcomponent
            ;
```

Les méthodes principales :

Listing 7 – Methode AnnuaireCocola

```
1
2               //soit deux hashMap representant un graphe
                    biparti, cette fonction retourne des listes d
                    'elements connexes.
3               public ArrayList<ArrayList<String>>
                    getConnexeElement(HashMap<String,ArrayList<
                    String>> part1,
4               HashMap<String,ArrayList<String>> part2);
5
6               //genere les treillis des fonctions
7               public void generateTableFonction();
8
9               //genere les treillis des composents
10              public void generateTableComponent();
11
12              //genere les treillis des interfaces
13              public void generateTableInterface();
```

ModelTableAbstract



Modéle uml de la génération de tables

ModelTableAbstract est une classe abstraite permettant la génération d' une table servant à calculer un treillis. Elle est spécialisée par ModelTable-FunctionProvided, ModelTableFunctionRequired, ModelTableInterfaceProvided,ModelTableInterfaceRequired,ModelTableComponent. Les specialisations rempliront leur table selon leur rôle. Les treillis seront générés par un type implémentant l'interface ITreillis. La g'énération de treillis s'effectue par une librairie exterieur Erca (2.2), ce system facilite donc l'implementation d'un treillis autre que celui-ci.

Les Attributs Principaux :

Listing 8 – Attributs AnnuaireCocola

```
1
2     //Interface utilisee pour le calcul des treillis
3     protected TreillisI treillis;
4
5     //represente les lignes de la table
```

```
6       private ArrayList<String> entity;
7
8       //represente les colonnes de la table
9       private ArrayList<String> attributes;
10
11      //represente les relation entre les lignes et les
            colonnes
12      protected  int[][] table;
```

Les méthodes principales :

Listing 9 – Attributs AnnuaireCocola

```
1
2       //creation de la table ainsi que ses relations
3       public abstract void createTableModel();
```

ITreillis

Description :

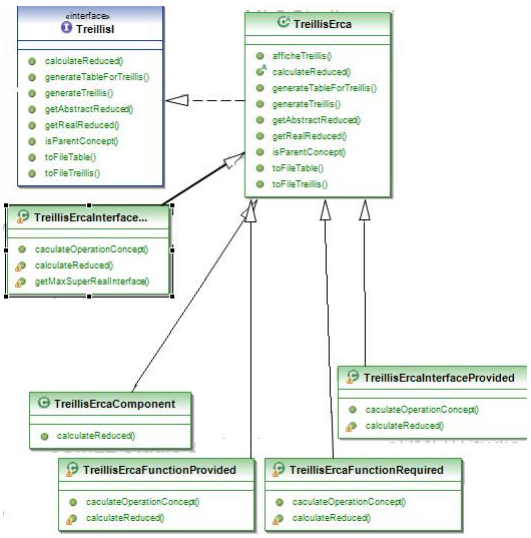
L'interface TreillisI possède toutes les signatures des fonctions dont à besoin la classe ModelTableAbstract pour faire appel au calcul des treillis et aux différents rèsultats obtenus. Ces méthodes devront être définies par l'implémentation du treillis.

Listing 10 – Attributs AnnuaireCocola

```
1
2 public interface TreillisI {
3
4       //genere la table servant a calculer le treillis a partir
            de la ModelTableAbstract
5       public void generateTableForTreillis(ModelTableAbstract
            table);
6
7       //genere le treillis
8       public void generateTreillis(ModelTableAbstract table);
9
10      //deduit les nouveaux concepts emergents.
11      public void calculateReduced(ModelTableAbstract table);
12
13      //retourne tous les concepts abstrait (nouveau concept
            calcule) et son numero de concept correspondant
14      public HashMap<String,String> getAbstractReduced(
            ModelTableAbstract table);
15
16      //retourne tous les concept reel (nouveau concept calcule
            ) et son numero de concept correspondant
17      public HashMap<String,String> getRealReduced(
            ModelTableAbstract table);
18
19      //renvoie vrai si c1 est un concept parent de c2
20      public Boolean isParentConcept(String c1,String c2);
```

```
21
22      //fonction a implementer pour avoir une sortie fichier de
            la table
23      public void toFileTable(ModelTableAbstract table,String
            path);
24
25      //fonction a implementer pour avoir une sortie fichier du
            treillis
26      public void toFileTreillis(ModelTableAbstract table,
            String path);
27
28          }
```

TreillisErca



Modéle uml de la génération de treillis

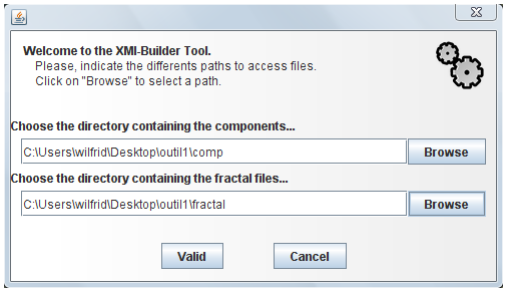
TreillisErca implemente l'interface ITreillis. Elle definie donc toutes les signatures inclusent dans l'interface.Pour le calcul des treillis, elle utilise la librairie ERCA (lien hyperText vers Erca). La deduction des nouveaux concept se fait par les specialisations de celle-ci. L'implement par Erca permet d'obtenir une sortie fichier des tables en latex, et une sortie fichier en .dot permettant la géération de **svg***(Scalable Vector Graphics)*.

6 Mode d'emploi

Annuaire de Composant Logiciel (ACL) est un logiciel permettant de réaliser l'analyse de librairies de composants, développé par des etudiants dans le cadre d'un projet de Master1.

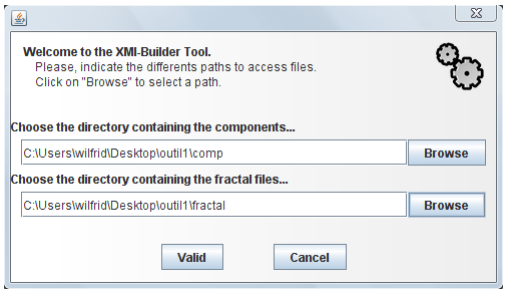
Il s'agit d'un annuaire permettant de consulter et de rechercher des informations sur les composants d'une librairie. Par ailleurs, il met à jour les nouvelles relations de subtitution ou de connexion entre les composants, les interfaces et les fonctions.
ATTENTION : Actuellement, cette interface est encore en cours de développement, il ne s'agit pas de la version finale.

6.1 Lancement



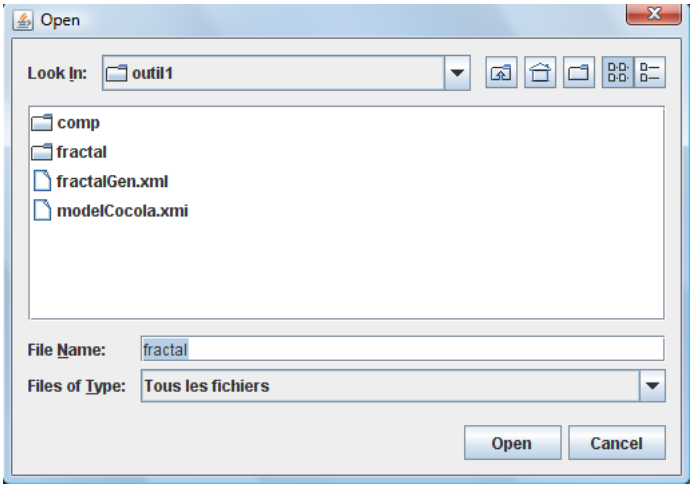
Fenêtre générale de sélection des répertoires

Cette fenêtre prend deux répertoires en entrée. Le premier va contenir l'ensemble des composants. Le second comprendra un ensemble de fichiers Fractal (ou Pivot) décrivant l'architecture des composants. En sortie, l'outil 1 générera un modèle respectant le métamodèle Cocola. Notons qu'il n'est pas obligatoire de passer systématiquement par l'outil 1 à chaque utilisation du programme. Il est en effet possible de charger un modèle Cocola au format XMI généré avec l'outil 1 lors d'une précédente utilisation.



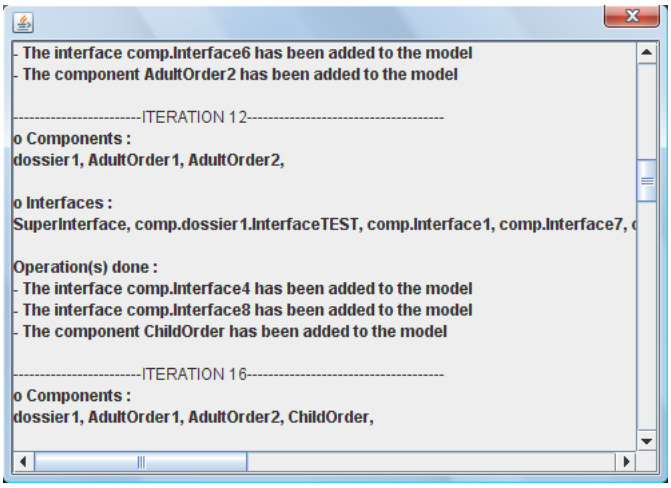
Fenêtre générale de sélection des répertoires

A partir de la fenêtre principale de l'outil, pour déterminer les chemins d'accès aux répertoires, une fenètre de type "JFileChooser" apparaît lorsque l'utilisateur presse un des boutons "Browse". Une fois ces deux chemins renseignés, la procédure de création du modèle pourra commencer en appuyant sur le bouton "Valid".



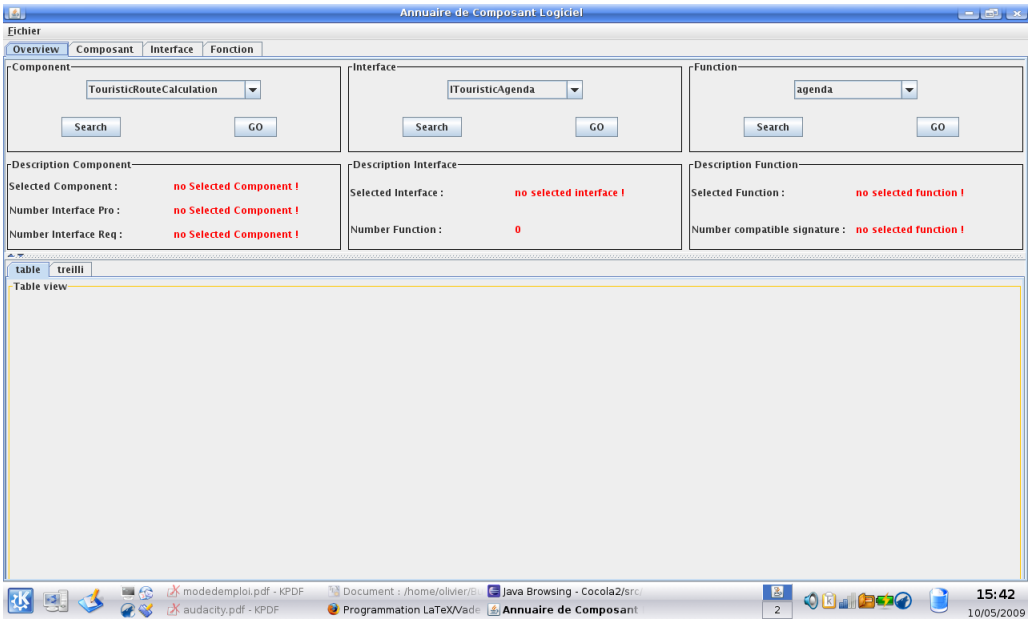
Fenêtre de sélection des répertoires

Ainsi, le parcours et l'analyse des différents fichiers se trouvant dans ces deux dossiers s'effectuent. Analyse qui, une fois terminée, est accompagnée d'une troisième fenêtre dîte "console" qui permet de voir les divers traitements effectués. Enfin, le bouton "Cancel" permet tout simplement de fermer l'outil 1.



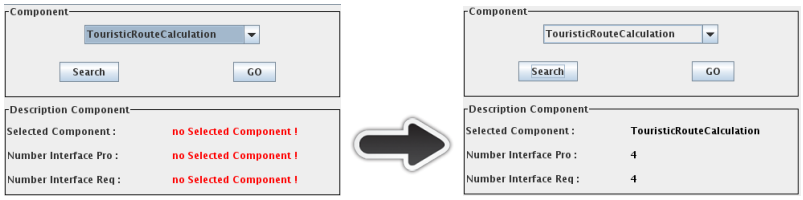
Fenêtre console affichant les traitements effectués

6.2 Onglet Overview

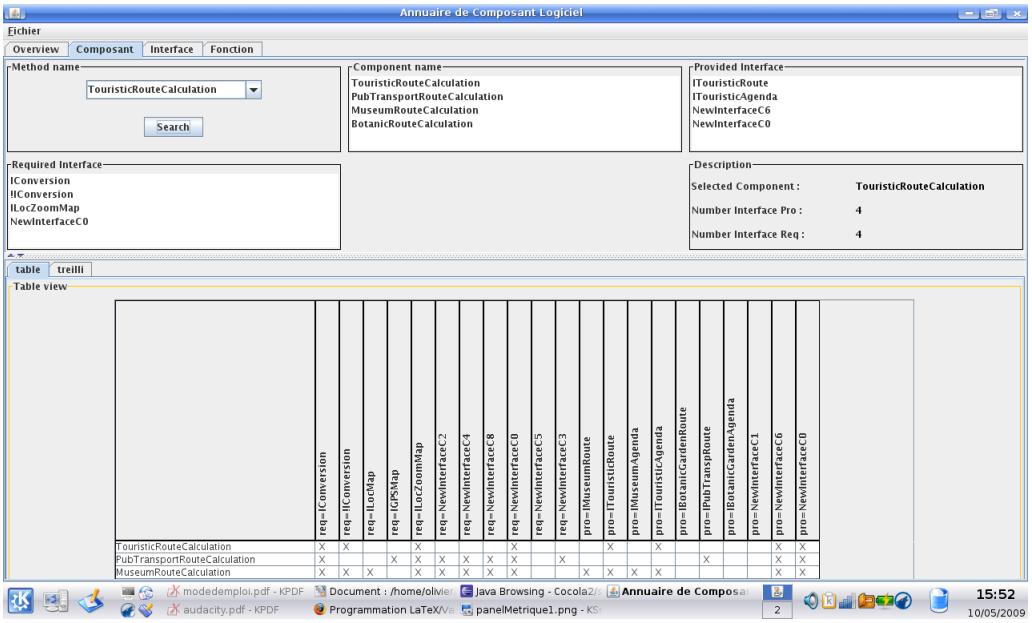


Affichage vue d'ensemble

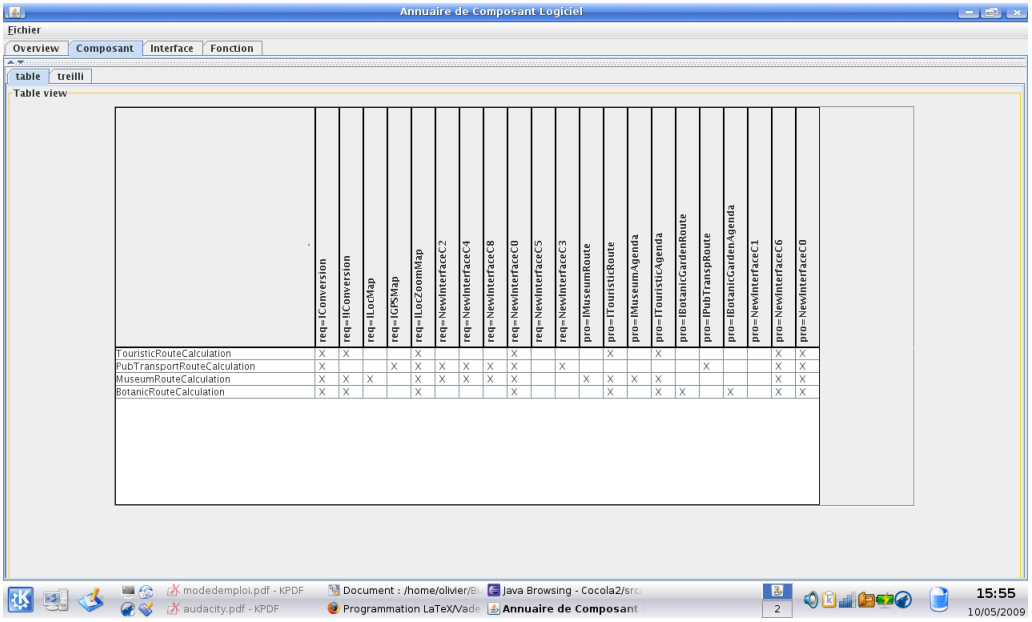
Cette vue permet d'obtenir rapidement des informations sur un composant, une interface ou une fonction. Par exemple, si l'on veut obtenir des informations sur TouristicRouteCalculation, il faut sélectionner son nom dans le menu défilant puis cliquer sur le bouton "search". Pour accéder directement à la vue des composants il suffit de cliquer sur le bouton "GO".



6.3 Onglet Composant



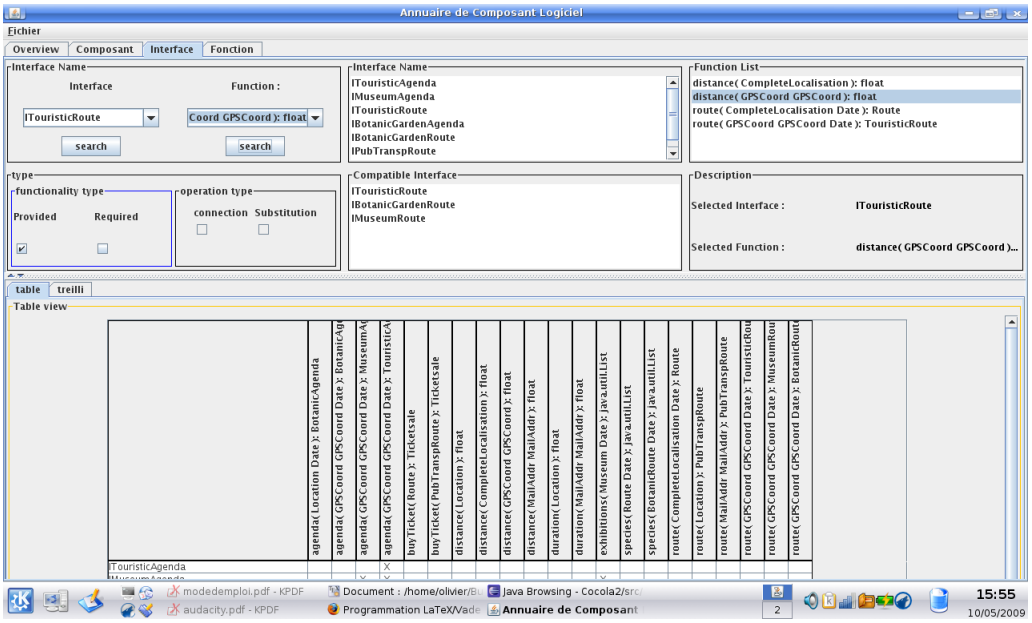
Informations relatives aux composants



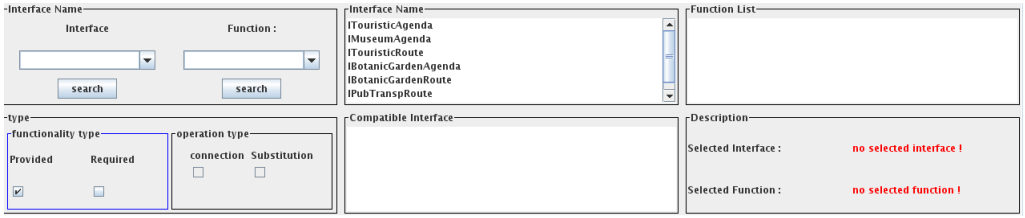
Affichage de la table des relations entre composants et interfaces.

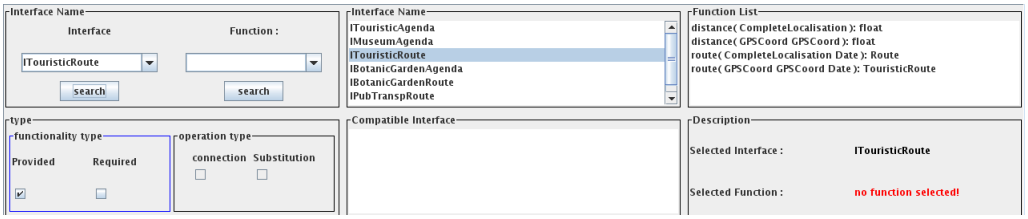
Il suffit de choisir dans le menu déroulant ou de taper le nom d'un composant puis d'appuyer sur le bouton "search" pour afficher la table des relations entre composants et interfaces.

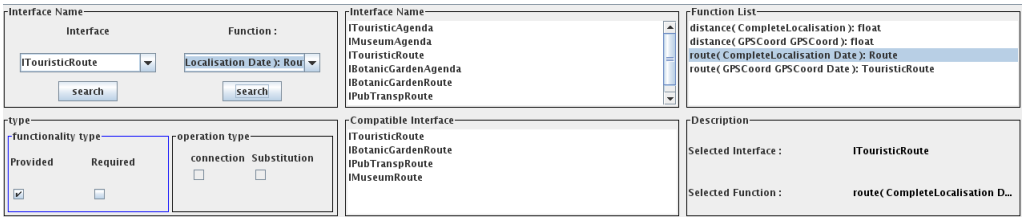
6.4 Onglet Interface



Informations relatives aux interfaces

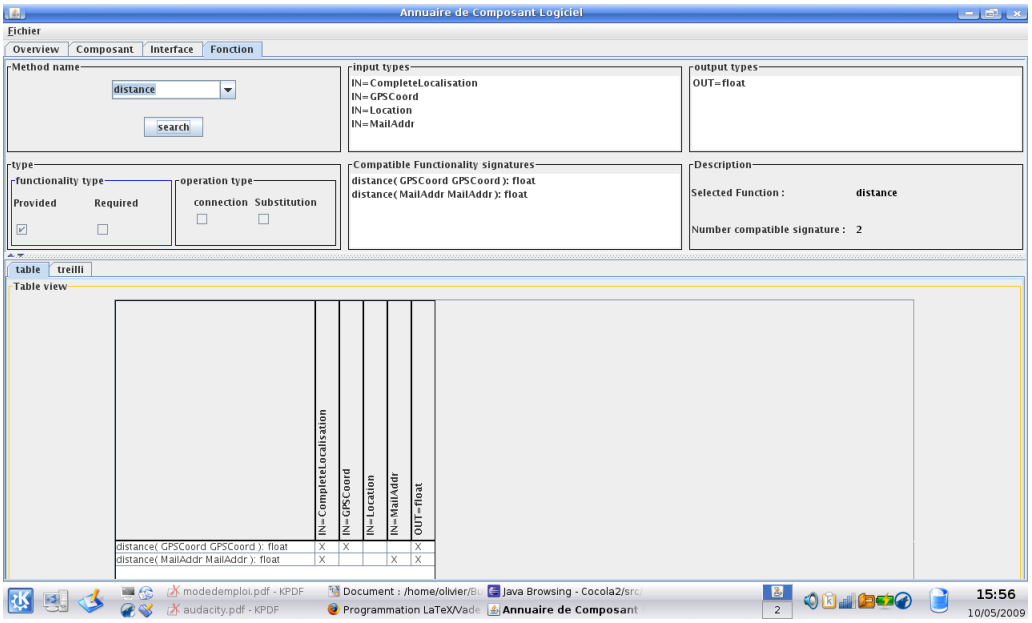






Pour obtenir les informations relatives à une interface, il faut dans un premier temps sélectionner le nom de l'interface à rechercher et de cliquer sur le bouton "search" en dessous du menu déroulant correspondant aux interfaces, on obtient alors la liste de ses fonctions. Ensuite, il faut sélectionner le nom d'une fonction et cliquer sur le bouton "search" en dessous du menu déroulant correspondant aux fonctions, afin d'obtenir la liste des interfaces possédant aussi cette fonction. L'encadré "functionality type" indique si l'interface est de type fourni ou requis. L'encadré "operation type" indique si l'interface peut se connecter ou se substituer à d'autres interfaces.

6.5 Onglet Fonction



Informations relative aux fonctions

Pour obtenir les informations relatives aux fonctions, il suffit de taper ou de sélectionner le nom d'une fontion puis de cliquer sur le bouton "search".

7 Perspectives

Nous souhaiterions ajouter :

- visualisation des treillis dans l'interface graphique.
- pouvoir ajouter un nouveau composant au réferentiel de manière incremental*(sans avoir à recalculer tous les treillis).*
- Actuellement, certaines librairies ne peuvent être calculées. En effet nous nous trouvons confrontés à des tables de relations trop importantes. Ce qui provoque une explosion combinatoire lors de la génération des treillis. Nous avons tenté d'apporter une première solution qui consiste à calculer les composantes connexes entre les composants. 5.2.5. Cependant cette approche n'est pas suffisante (Ex : La librairie Dream possède une composante de 97 composants).

8 Bilan

Ce projet s'est avérée être très enrichissant. Nous avons travaillé dans un environnement professionnel, pouvant ainsi nous familiariser avec le monde du travail. Pour effectuer un petit bilan de ce que nous a apporté ce projet, voici un bref listing des connaissances assimilées.

8.1 Notions appropriées durant le projet

8.1.1 Acquis liés au cadre

- Réalisation d'un projet dans un cadre de collaboration (et les contraintes que cela peut apporter).
- Respect de conventions et normes (notamment sur les commentaires détaillés du code).
- Suivi strict et méthodique d'un cahier des charges et des instructions données par nos encadrants.

8.1.2 Expériences assimilées

- Analyse détaillée d'un projet existant pour en dégager ses fonctionnalités, faiblesses et atouts.
- Développement d'un système de "reverse engineering".
- étude et réalisation d'un métamodèle permettant la description de modèles.
- Utilisation de nombreux plugins et packages (EMF, Erca, Reflect etc.).
- Application d'une méthode formelle pour réaliser un algorithme.
- Génération de treillis de Galois.

8.2 Comparaison entre le planning initial et final

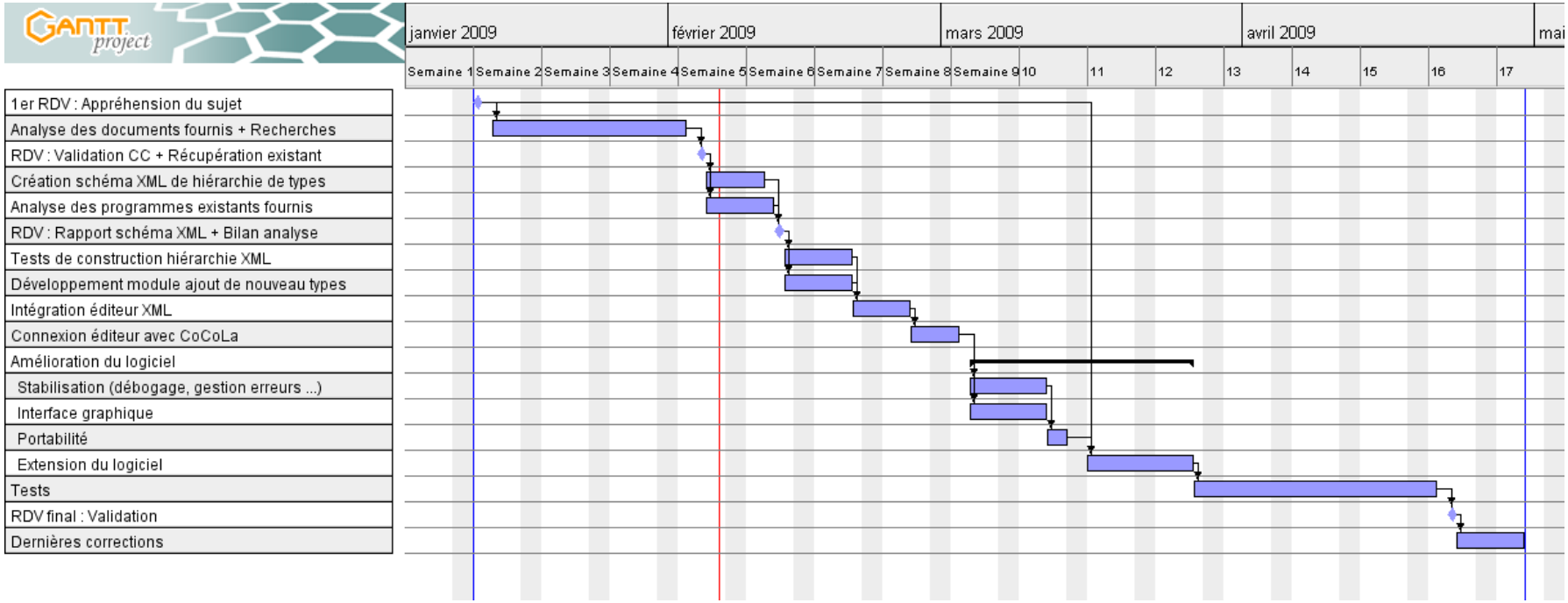
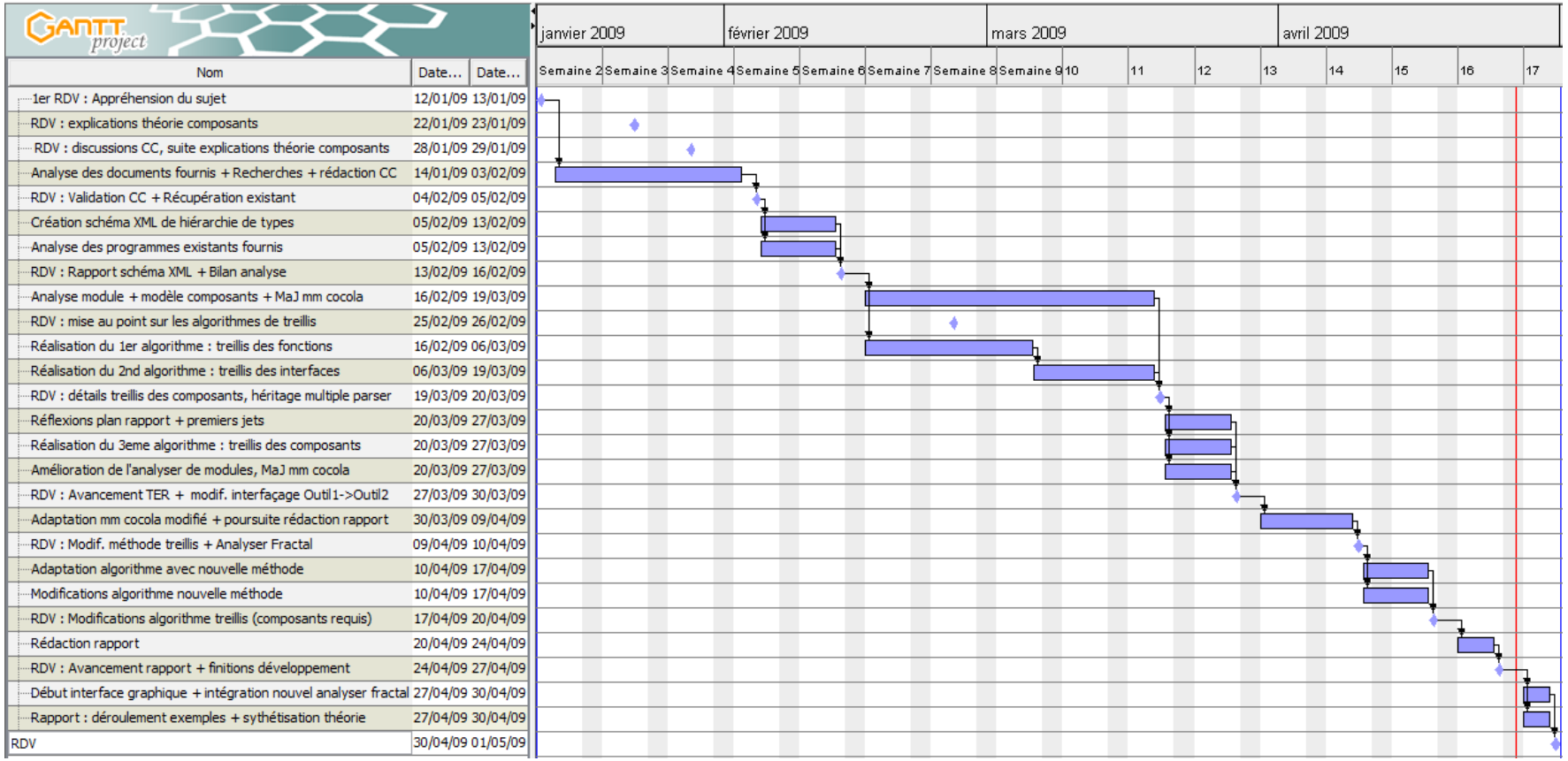


FIG. 1 – Diagramme du planning initial



Nom	Date...	Date...
1er RDV : Appréhension du sujet	12/01/09	13/01/09
RDV : explications théorie composants	22/01/09	23/01/09
RDV : discussions CC, suite explications théorie composants	28/01/09	29/01/09
Analyse des documents fournis + Recherches + rédaction CC	14/01/09	03/02/09
RDV : Validation CC + Récupération existant	04/02/09	05/02/09
Création schéma XML de hiérarchie de types	05/02/09	13/02/09
Analyse des programmes existants fournis	05/02/09	13/02/09
RDV : Rapport schéma XML + Bilan analyse	13/02/09	16/02/09
Analyse module + modèle composants + MaJ mm cocola	16/02/09	19/03/09
RDV : mise au point sur les algorithmes de treillis	25/02/09	26/02/09
Réalisation du 1er algorithme : treillis des fonctions	16/02/09	06/03/09
Réalisation du 2nd algorithme : treillis des interfaces	06/03/09	19/03/09
RDV : détails treillis des composants, héritage multiple parser	19/03/09	20/03/09
Réflexions plan rapport + premiers jets	20/03/09	27/03/09
Réalisation du 3eme algorithme : treillis des composants	20/03/09	27/03/09
Amélioration de l'analyser de modules, MaJ mm cocola	20/03/09	27/03/09
RDV : Avancement TER + modif. interfaçage Outil1->Outil2	27/03/09	30/03/09
Adaptation mm cocola modifié + poursuite rédaction rapport	30/03/09	09/04/09
RDV : Modif. méthode treillis + Analyser Fractal	09/04/09	10/04/09
Adaptation algorithme avec nouvelle méthode	10/04/09	17/04/09
Modifications algorithme nouvelle méthode	10/04/09	17/04/09
RDV : Modifications algorithme treillis (composants requis)	17/04/09	20/04/09
Rédaction rapport	20/04/09	24/04/09
RDV : Avancement rapport + finitions développement	24/04/09	27/04/09
Début interface graphique + intégration nouvel analyser fractal	27/04/09	30/04/09
Rapport : déroulement exemples + sythétisation théorie	27/04/09	30/04/09
RDV	30/04/09	01/05/09

FIG. 2 – Diagramme du planning final

Le diagramme décrivant le planning initial a été réalisé durant la phase de production du cahier des charges. Cette dernière s'étant déroulée au tout début du projet, le calendrier à connu entre temps de nombreuses modifications. A l'origine, le travail à accomplir n'était que l'amélioration du programme de Nour Alhouda Aboud. Comme nous l'avons précédement explicité, notre analyse du travail existant y a révélé plusieurs imperfections. Nous avons donc en conséquence, préféré concevoir entièrement notre programme permettant la création, l'analyse et la substitution de composants pour la production de logiciels. Cependant, toutes les tâches évoquées (interface graphique, tests, portabilité etc.) dans le planning initial ont été prise en compte dans le nouveau programme. Un grand nombre de nouvelles tâches ont alors vu le jour et apparaissent dans le planning final.

8.3 Conclusion

Nous sommes très satisfaits du déroulement de notre TER. Nous avons eu l'occasion de beaucoup apprendre, tant sur les méthodes professionnelles que sur le plan humain. Nous avons su assimiler les notions relatives au projet et nous adapter à notre nouvel environnement, contrastant avec celui de l'université. Nous avons aussi eu le privilège d'appréhender diverses facettes du développement dans un cadre de recherches : la partie analyse, conception et programmation d'un projet, mais aussi au niveau organisationnel avec sa délicate gestion de projet ou encore d'un point de vue esthétique et ergonomique avec la réalisation et implémentation d'une interface graphique. Nous estimons que l'objectif de ce TER est accompli : réalisation d'un projet professionnel, utilisation et adaptation des connaissances universitaires et l'acquisition d'une expérience dans ce domaine. Notons par ailleurs que le travail obtenu satisfait pleinement nos encadrants. En effet, à la vue de ces resultats très prometteurs, les encadrants nous ont proposé de venir travailler en tant que stagiaire au LIRMM durant cet été.

Références

[1] Nour Alhouda Aboud. Construction d'annuaires de composants par classification. Master's thesis, Université Montpellier 2, 2007. Ce rapport expose les bases de la méthode permettant de mettre en évidence les relations de compatibilité entre composants.

[2] Aide Eclipse. L'aide contextuelle fournis par Eclipse (et leurs plugins) fût très probablement l'une des plus importantes sources de documentations pour la réalisation de notre TER.

[3] Documentation EMF. http://help.eclipse.org/ganymede/index.jsp?topic=/org.eclipse.emf.doc/tutorials/clibmod/clibmod.html. Documentation du plugin EMF.

[4] Luc Fabresse. *Du découplage à l'assemblage non-anticipé de composants*. PhD thesis, Université Montpellier 2, 2007. Cette thèse nous a servi de référence pour le vocabulaire spécifique aux composants logiciels.

[5] Tutoriels Fractal. http://fractal.ow2.org/tutorials/adl/index.html. Tutoriels sur l'utilisation du modèle composant Fractal ADL.

[6] Gabriela Arévalo, Nicolas Desnos, Marianne Huchard, Christelle Urtado, Sylvain Vauttier. Construction dynamique d'annuaires de composants par classification de services. Traduction française partielle de l'article "FCA-based service classification to dynamically build efficient software component directories"., 2008.

[7] Gabriela Arévalo, Nicolas Desnos, Marianne Huchard, Christelle Urtado, Sylvain Vauttier. Fca-based service classification to dynamically build efficient software component directories. *International Journal of General Systems*, 2008. Article très complet explicitant une méthode de classification de composants dans le but de créer des logiciels.

[8] Documentation Java. http://java.sun.com/j2se/1.5.0/docs/api/index.html. Javadoc détaillant l'ensemble des classes et méthodes des packages standards de Java.

[9] Forum JavaFr. http://www.javafr.com/forum/. Utilisation du forum de la communauté javafr pour des problèmes ciblés.

[10] Documentation Reflect. http://java.sun.com/docs/books/tutorial/reflect/index.html. Documentation du package reflect de java (utilisé pour le « reverse engineering »).