Rapport préliminaire

Composants et Architecture

FAGNIEZ Florian et RULLIER Noémie
1er décembre 2013

Table des matières

1 Introduction

Ce rapport contient l'ensemble du travail fourni lors des séances de TD du module **Composants et Architecture** sur le système Client/Server.

Ce document contient les diagrammes de classes suivants :
- Celui du métamodèle M2
- Celui du modèle M1 (Non-Agrandi)
- Celui du modèle M1 (Agrandi)

Le lecteur de ce rapport pourra trouver les explications (textuelles, sous forme OCL, etc...) permettant de justifier nos choix lors de la conception de ces diagrammes.

Enfin, nous écrirons dans notre langage les interactions entre les composants et configurations de notre système.

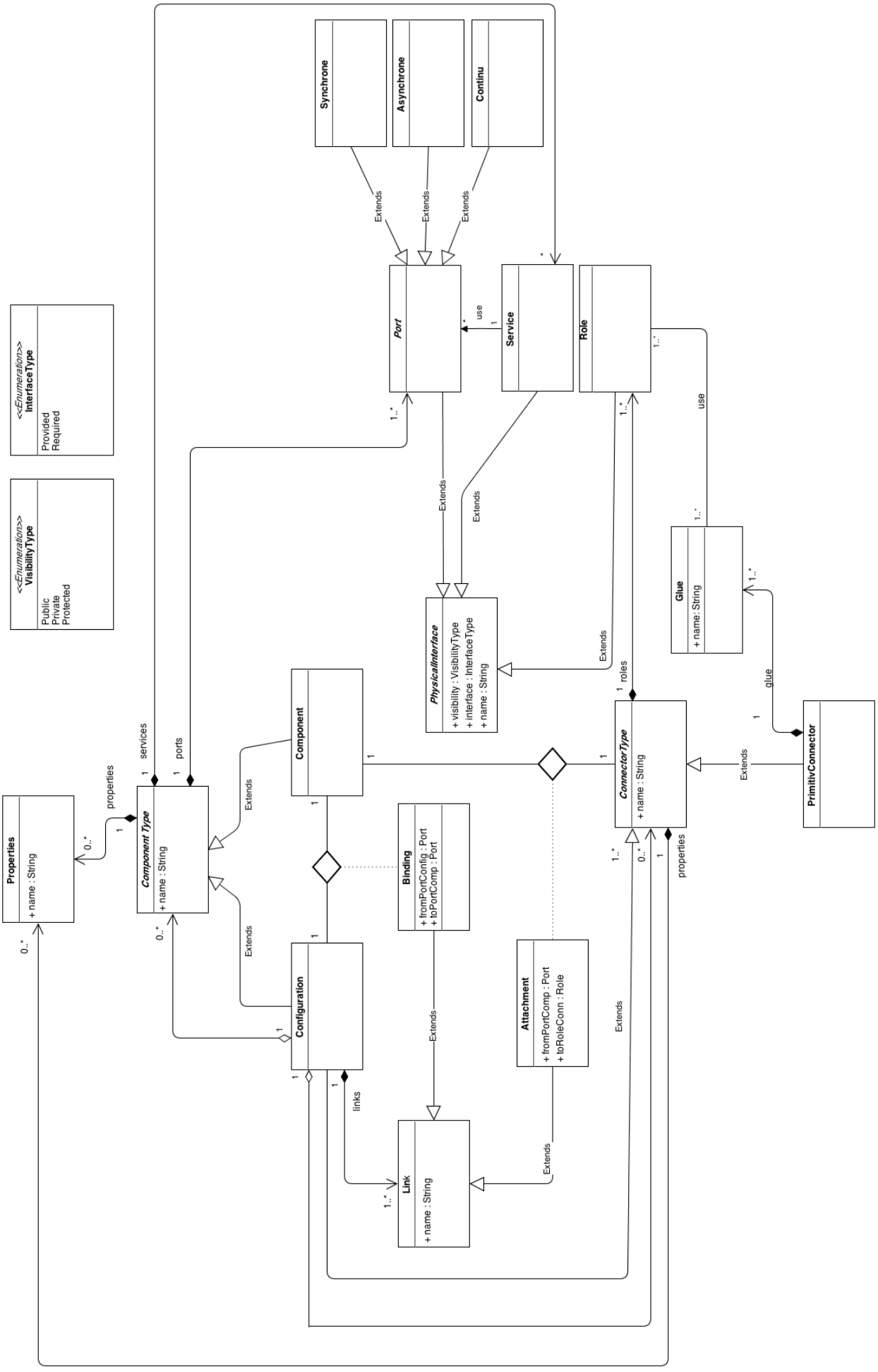
2 Métamodele niveau M2

Métamodèle signifie littéralement modèle du modèle. Il peut être défini comme la représentation d'un point de vue particulier sur des modèles.

Le M2 permet de fournir le langage de notre application : il fournit des concepts de classes, d'instance et de relations d'héritages entre autres.

2.1 Diagramme de classe du M2

Ci-dessous, le diagramme de classe de notre M2


Synchrone
Asynchrone
Continu
Extends
Port
Service
use
Role
<<Enumeration>>
InterfaceType
Provided
Required
<<Enumeration>>
VisibilityType
Public
Private
Protected
PhysicalInterface
+ visibility : VisibilityType
+ interface : InterfaceType
+ name : String
Glue
+ name: String
roles
glue
Component
ConnectorType
+ name : String
PrimitivConnector
services
ports
properties
Properties
+ name : String
Component Type
+ name : String
Binding
+ fromPortConfig : Port
+ toPortComp : Port
Configuration
Attachment
+ fromPortComp : Port
+ toRoleConn : Role
links
Link
+ name : String

2.2 Explications et Contraintes OCL

Notre M2 permet de définir les différents concepts que nous allons utiliser par la suite :

Pattern Composite pour les composants : On remarque la présence d'un premier pattern composite permettant de définir un composant et une configuration. Un composant possède différents types de propriétés (regroupé dans une classe **Properties**), il est de plus composé de ports et services tous les deux héritant d'une classe **PhysicalInterface**. Chaque composant doit au moins avoir un port *Provided*, cette contrainte a été définie à l'aide d'OCL.

Pattern Composite pour les connecteurs : On peut voir un deuxième pattern composite permettant cette fois de définir un connecteur. Le composite est aussi la configuration définie ci-dessus. Le **PrimitivConnector** est composé de glues définies par la classe **Glue**. Celle-ci utilise des rôles héritant aussi de la classe **PhysicalInterface**. Un **ConnecteurType** est composé de ces mêmes rôles et de différents type de propriétés définies elles aussi dans la classe **Properties**.

Glue : La glue permet de définir comment les rôles intéragissent entre eux. Elle doit relier au minimum deux ports dont un *Provided* et un *Required*.

PhysicalInterface : Cette classe permet de définir une interface. Cela peut être un port, un service ou un rôle. Chaque interface est donc définie par un type qui peut être *Provided* pour fournis et *Required* pour requis. Une interface fournie va fournir des informations. Contrairement à une interface requise qui est une interface par laquelle on va pouvoir faire passer des informations (elle requiert des informations).

Liens : Nous avons de plus défini différents liens **Binding** et **Attachment**. Le premier permet de relier un composant à une configuration via des ports. Afin d'expliciter le fait que le port de la configuration doit être *Required* et que le port du composant doit être *Provided*, nous avons utilisé une contrainte OCL. Le deuxième type de lien (Attachment) permet de relier un composant à un connecteur via un port du composant et un rôle du connecteur. Nous avons aussi ici besoin d'une contrainte OCL afin de spécifier que l'un des deux attributs doit être *Provided* et l'autre *Required*.

Voici nos différentes contraintes OCL :

```
1  context Binding
2  inv: self.fromPortConfig.interface.oclIsTypeOf(InterfaceType:Required)
3  and self.toPortComp.interface.oclIsTypeOf(InterfaceType:Provided)
4
5  context Attachment
6  inv: (self.fromPortComp.interface.oclIsTypeOf(InterfaceType:Required) implies self.toRoleConn.
       interface.oclIsTypeOf(InterfaceType:Provided))
7  and (self.fromPortComp.interface.oclIsTypeOf(InterfaceType:Provided) implies self.toRoleConn.
       interface.oclIsTypeOf(InterfaceType:Required))
8
9  context Component
10 inv: self.ports -> exists ( p | p.interface.oclIsTypeOf(InterfaceType:Provided))
11
12 context Glue
13 inv: self.roles -> exists ( r1, r2 | r1.interface.oclIsTypeOf(InterfaceType:Provided) > 1
14 and r2.interface.oclIsTypeOf(InterfaceType:Required) > 1)
```

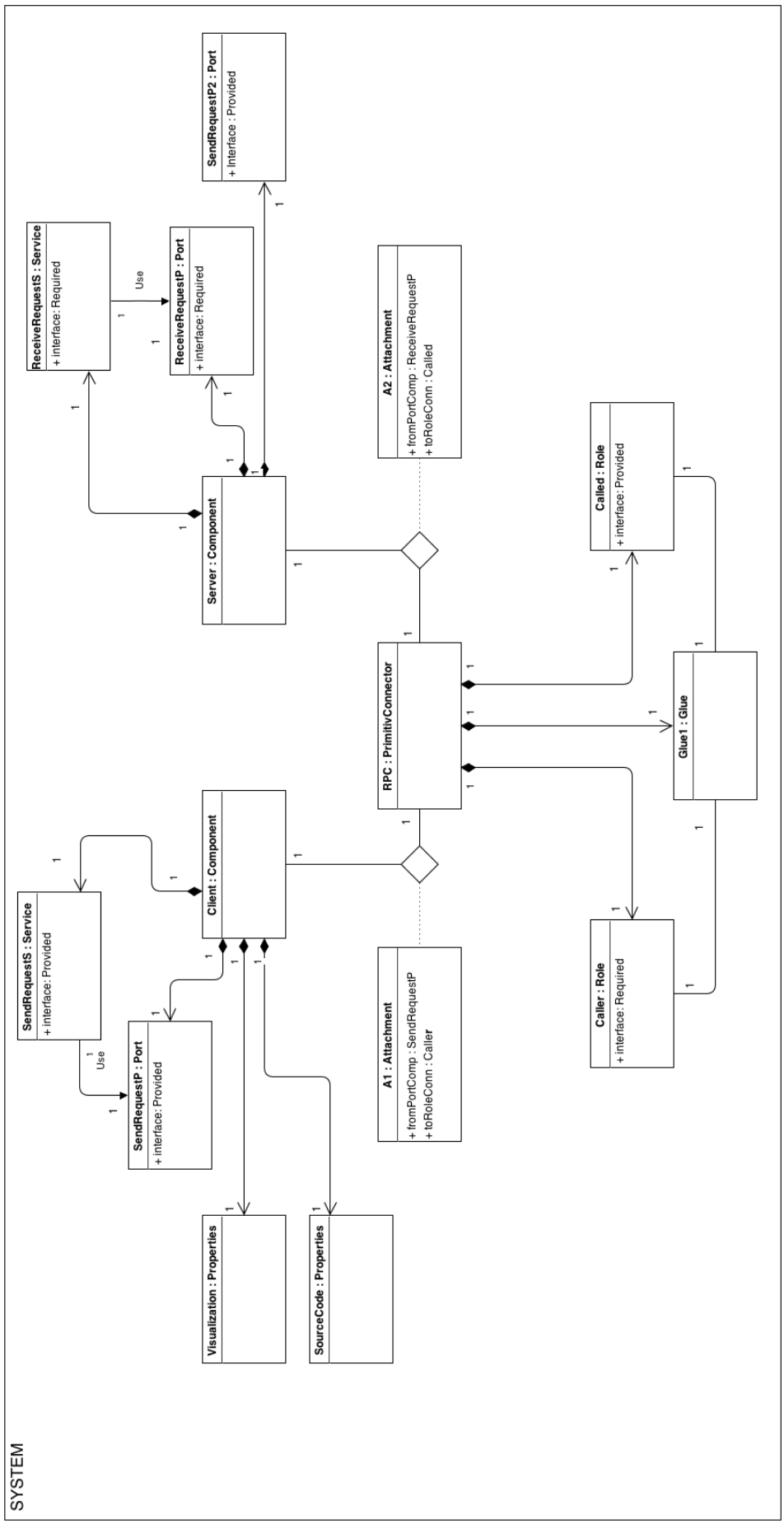
3 Modèle niveau M1

Le modèle niveau M1 concerne l'instanciation des concepts définis auparavant. Ici, l'objectif est de réaliser une modélisation de notre système client/serveur, avec notre serveur en tant que composant d'une part et en tant que composition d'autre part.

Remarque : le lecteur pourra remarquer que nous n'avons en aucun cas spécifié le type de port utilisé ici. En effet, on pourra remarquer que lorsque nous avons établit notre M2, nous avons spécifié la classe **Port** comme étant une classe abstraite pouvant être "généralisée" en tant que **Port Synchrone/Asynchrone/Continu**. N'ayant pas d'informations précises sur le type de port à utiliser ici, nous avons décidé de tout simplement garder la classe **Port** pour le moment. Bien évidemment, cette distinction sera à faire lors de la programmation de notre système client/serveur.

3.1 Diagramme de classe du M1 "Non-Agrandi"

Ci-dessous notre premier diagramme de classe avec notre système client/serveur, le serveur étant ici représenté comme étant un composant.


SYSTEM
SendRequestP2 : Port
+ Interface : Provided
ReceiveRequestS : Service
+ interface: Required
Use
ReceiveRequestP : Port
+ interface: Required
A2 : Attachment
+ fromPortComp : ReceiveRequestP
+ toRoleConn : Called
Server : Component
Called : Role
+ interface: Provided
RPC : PrimitivConnector
Glue1 : Glue
Client : Component
SendRequestS : Service
+ interface: Provided
Use
SendRequestP : Port
+ interface: Provided
A1 : Attachment
+ fromPortComp : SendRequestP
+ toRoleConn : Caller
Caller : Role
+ interface: Required
Visualization : Properties
SourceCode : Properties

Nous remarquons ici 3 "**blocs**" :
- Le client et les éléments dont il est composé
- Le serveur et les éléments dont il est composé
- Le connecteur primitif et les éléments dont il est composé

Commençons par le client : il est composé d'un **Port** (*SendRequestP*) ainsi que d'un **Service** (*SendRequestS*) (qui sont tous les deux *Provided*) ainsi que de deux éléments qui sont des *Properties* : **Visualization** et **SourceCode**

Le serveur quant à lui est composé de deux **Ports** (l'un étant *Provided* (*SendRequestP2*), l'autre *Required* (*ReceiveRequestP*)) et d'un service (qui lui est *Required* (*ReceiveRequestS*)).

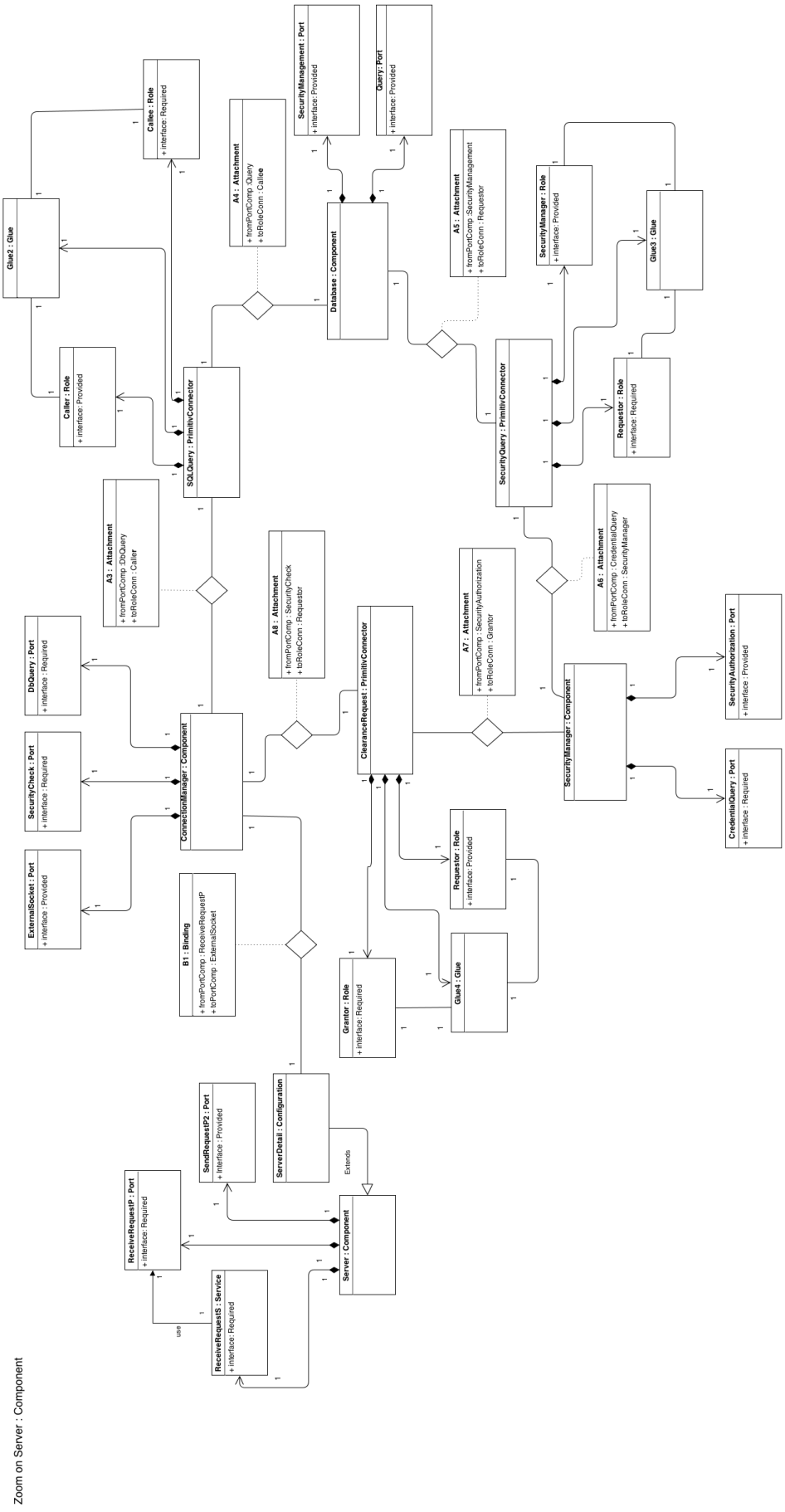
Enfin, notre RPC (le *PrimitivConnector*) est composé de deux **Roles** (un *Required* (*Caller*) et l'autre *Provided* (*Callee*)) ainsi que de **Glue**.

Conformément au M2 que nous avons représenté auparavant, ces deux composants sont reliés à un **connecteur primitif** (qui est ici notre **RPC**) à l'aide de liens d'attachements que nous avons ici nommés *A1* et *A2*

Nous remarquons que les "**règles**" établies lors de notre M2 sont respectées :
- Tout composant doit au moins posséder un port fourni
- Tout composant est relié à un connecteur à l'aide d'un lien d'attachement
- Un connecteur possède deux rôles (l'un étant fourni, l'autre requis) et communiquent grâce à la glue
- Les liens d'attachement sont constitués du port composant d'une part et du rôle connecteur d'autre part (Sachant que si l'un est *Provided* l'autre doit obligatoirement être *Required*)

3.2 Diagramme de classe du M1 "Agrandi"

Le diagramme de classe M1 "Agrandi" représente le diagramme précédemment décrit mais en mettant le serveur en tant que configuration :


Zoom on Server : Component
Callee : Role
+ interface: Required
SecurityManagement : Port
+ interface: Provided
Query: Port
+ interface: Provided
A4 : Attachment
+ fromPortComp: Query
+ toRoleConn : Callee
Glue2 : Glue
Database : Component
A5 : Attachment
+ fromPortComp: SecurityManagement
+ toRoleConn : Requestor
SecurityManager : Role
+ interface: Provided
Glue3 : Glue
Caller : Role
+ interface: Provided
SQLQuery : PrimitivConnector
SecurityQuery : PrimitivConnector
Requestor : Role
+ interface: Required
A3 : Attachment
+ fromPortComp: DbQuery
+ toRoleConn : Caller
A8 : Attachment
+ fromPortComp : SecurityCheck
+ toRoleConn : Requestor
A7 : Attachment
+ fromPortComp : SecurityAuthorization
+ toRoleConn : Grantor
A6 : Attachment
+ fromPortComp : CredentialQuery
+ toRoleConn : SecurityManager
DbQuery : Port
+ interface : Required
SecurityCheck : Port
+ interface : Required
ExternalSocket : Port
+ interface : Provided
ConnectionManager : Component
ClearanceRequest : PrimitivConnector
SecurityManager : Component
SecurityAuthorization : Port
+ interface : Provided
CredentialQuery : Port
+ interface : Required
B1 : Binding
+ fromPortComp : ReceiveRequestIP
+ toPortComp : ExternalSocket
Requestor : Role
+ interface: Provided
Grantor : Role
+ interface: Required
Glue4 : Glue
SendRequestP2 : Port
+ Interface : Provided
ServerDetail : Configuration
Extends
ReceiveRequestIP : Port
+ interface: Required
Server : Component
ReceiveRequestS : Service
+ interface: Required
use

Nous ne re-expliquerons pas les interactions entre le RPC, le client et le serveur (composant) ainsi que les éléments qui les composent comme ceci à été expliqué dans le paraghraphe précédent.

Afin de bien montrer ce lien qui permet de passer de notre **serveur** composant à notre **serveur composition**, nous avons décidé de modéliser cette relation par une relation d'héritage : concept qui respecte notre Métamodèle M2.

Avant d'expliquer l'ensemble des connections qui s'établit entre les éléments de notre serveur, il nous semblait évident de détailler ces éléments et leur composition :
Du côté des **Components** :

1. Un **ConnectionManager** composé de trois ports distincts :
   - **ExternalSocket** qui est un *Port Provided*
   - **SecurityCheck** qui est un *Port Required*
   - **DbQuery** qui est un *Port Required*
2. La **Database** qui est composé de deux ports distincts :
   - **SecurityManagement** qui est un *Port Provided*
   - **Query** qui est un *Port Provided*
3. La **SecurityManager** qui est également composée de deux ports distincts :
   - **CredentialQuery** qui est un *Port Required*
   - **SecurityAuthorization** qui est un *Port Provided*

Du côté des **PrimitivConnector** :

1. **SQLQuery** qui est composé de deux **Rôles** et de sa **Glue** :
   - **Caller** qui est un *Rôle Provided*
   - **Callee** qui est un *Rôle Required*
   - Les deux rôles pouvant s'échanger des informations grâce à la glue
2. **SecurityQuery** qui est également composé de deux **Rôles** et de sa **Glue** :
   - **SecurityManager** qui est un *Rôle Provided*
   - **Requestor** qui est un *Rôle Required*
   - Les deux rôles pouvant s'échanger des informations grâce à la glue
3. **ClearanceRequest** qui est également composé de deux **Rôles** et de sa **Glue** :
   - **Grantor** qui est un *Rôle Required*
   - **Requestor** qui est un *Rôle Provided*
   - Les deux rôles pouvant s'échanger des informations grâce à la glue

Pour respecter les concepts de notre M2, il est nécessaire d'établir des liens **d'Attachment** entre nos **Components** et nos PrimitivConnector, nous en avons plusieurs ici :

- Un lien A3 entre le **ConnectionManager** et le **SQLQuery**, établissant une connexion du port **DbQuery** au rôle **Caller**
- Un lien A4 entre la **Database** et le **SQLQuery**, établissant une connexion du port **Query** au rôle **Callee**
- Un lien A5 entre la **Database** et le **SecurityQuery**, établissant une connexion du port **SecurityManagement** au rôle **Requestor**
- Un lien A6 entre le **SecurityQuery** et le **SecurityManager**, établissant une connexion du port **CredentialQuery** au rôle **SecurityManager**
- Un lien A7 entre le **SecurityManager** et le **ClearanceRequest**, établissant une connexion du port **SecurityAuthorization** au rôle **Grantor**
- Un lien A8 entre le **ConnectionManager** et le **ClearanceRequest**, établissant une connexion du port **SecurityCheck** au rôle **Requestor**

Enfin nous pouvons relier le **ConnectionManager** au **ServerDetail** (étant notre serveur en tant que configuration) grâce à un lien **Binding** se faisant entre les ports **ReceiveRequestP** et **ExternalSocket**

4 Langage HADL

Nous avons pu définir notre propre langage (HADL) permettant de décrire des architectures de composants. Voici la représentation de notre M1 CLient/Server dans ce langage :

```
System Client_Server = {
    Component Client = {
        Port = { Provided: SendRequestP; }
        Service = { Provided: SendRequestS; }
        Properties = { Vizualisation = {}, SourceCode = {};}
    }
    Component Server = {
        Port = { Required: ReceiveRequestP; Provided: SendRequestP2; }
        Service = { Required: ReceiveRequestS; }
    }
    PrimitivConnector RPC = {
        Role = { Required: Caller; Provided: Called; }
        Glue = { Glue1 = {Caller, Called}; }
    }
    Attachment = {
        A1 = {fromPortComp Client.sendRequestP toRoleConn RPC.Caller},
        A2 = {fromPortComp Server.receiveRequestP toRoleConn RPC.Called};
    }

    System ServerDetail = {
        Component ConnectionManager = {
            Port = { Provided: ExternalSocket; Required: SecurityCheck, DbQuery; }
        }
        Component Database = {
            Port = { Provided: SecurityMangaement, Query ;}
        }
        Component SecurityManager = {
            Port = { Required: CredentialQuery; Provided: SecurityAuthorization; }
        }
        PrimitivConnector SQLQuery = {
            Role = { Provided: Caller; Required: Callee; }
            Glue = { Glue 2 = {Caller, Callee}; }
        }
        PrimitivConnector SecurityQuery = {
            Role = { Provided: SecurityManager; Required: Requestor; }
            Glue = { Glue 3 = {SecurityManager, Requestor}; }
        }
        PrimitivConnector Clearancerequest = {
            Role = { Provided: Caller; Requestor: Grantor; }
            Glue = { Glue 4 = {Requestor, Grantor}; }
        }
        Attachment = {
            A3 = {fromPortComp ConnectionManager.DbQuery toRoleConn SQLQuery.Caller},
            A4 = {fromPortComp Database.Query toRoleConn SQLQuery.Callee},
            A5 = {fromPortComp Database.SecurityManagement toRoleConn SecurityQuery.Requestor
                },
            A6 = {fromPortComp SecurityManager.CredentialQuery toRoleConn SecurityQuery.
                SecurityManager},
            A7 = {fromPortComp SecurityManager.SecurityAuthorization toRoleConn
                ClearanceRequest.Grantor},
            A8 = {fromPortComp ConnectionManager.SecurityCheck toRoleConn ClearanceRequest.
                Requestor};
        }
        Binding = {
            B1 = {fromPortComp Server.ReceiveRequestP toPortComp ConnectionManager.
                ExternalSocket};
```

```
52          }
53      }
54  }
```

5 Implémentation du problème

5.1 Hiérarchie du projet

Afin de posséder une implémentation propre et la plus lisible possible, nous avons décidé de séparer nos concepts dans 3 packages : Le package **M2** qui possède les concepts principaux de notre architecture, le package **M1** qui contient l'instanciation des concepts définis dans le **M2**. Enfin le **M0** contiendra le main de notre application.

5.1.1 M2

Le M2 contient l'ensemble des concepts de notre application :
- Configurations,Connecteurs et Composants
- Liens (Attachement et Binding)
- PhysicalInterface (Roles, Ports et Services)
- Glue
- Les diverses énumérations(InterfaceType et VisibilityType)

Le M2 contient également les interfaces et classes abstraites qui dépendent de nos concepts. Nous ne redétaillerons pas ces concepts car ils ont été expliqués auparavant.Pour davantage de renseignements sur l'implémentation ou sur les différentes méthodes de ce composant, le lecteur pourra directement consulter la Javadoc ou le code source des classes concernées.

5.1.2 M1

Le M1 contient l'instanciation des concepts définis dans le M2
- Client
- Server et ServerDetail
- ConnectionManager
- SQLQuery
- Database
- SecurityQuery
- ClearanceRequest
- Les liens d'Attachment et de Binding

Une fois de plus, nous ne redétaillerons pas l'instanciation de ces concepts ni les liens qui les unissent au M2, ces liens ayant été définis auparavant. Notons cependant que notre serveur défini en tant que **Configuration** est désormais **composé** de notre **ServerDetail** il ne s'agit ainsi plus d'un héritage comme le précisait notre diagramme de classe.

5.1.3 M0

Le package **M0** contient « l'exécutable » de notre application. Elle fait appel à un service du client qui permet l'envoi d'une requête au serveur. Cette requête transite à travers les différents composants de notre application puis retourne au client afin de lui fournir une réponse.

5.2 Points sensibles de l'implémentation

Au cours de l'implémentation de notre projet, nous avons détecté des points sensibles que nous allons expliquer dans cette partie.

5.2.1 Pré-requis : la base de données

Afin de faire fonctionner correctement le programme, l'utilisateur doit disposer d'une base de données nommé "hadl" avec pour **login** : "**root**" et pour **password** : "**root**"

Nous avons en effet créé une véritable base de données pour ce projet. Dès le lancement du **main**, nous nous assurons automatiquement de créer une table **Person** si elle n'existe pas et de remplir celle-ci de personnes contenant chacune un **ID** et un **nom**. Le service du client permet à l'aide d'une requête de récupérer des informations de la base de données.

5.2.2 Pattern composite

Nous allons expliquer l'implémentation des patterns composites de notre modèle. En effet, afin d'implémenter un pattern composite en Java nous devons définir le concept *Composant* comme une interface. Les deux concepts *Feuille* et *Composite* implémente l'interface du *Composant*. Notre concept *Composant* est composé d'autres classes. Nous devons donc définir des attributs comme des *List*. Cependant puisque notre concept *Composant* est défini comme une interface, nous ne pouvons pas ajouter ces attributs dans l'interface. Nous sommes donc obligés de les définir pour chacun des concepts *Composite* et *Feuille*.

5.2.3 Transmission de la requête

Les composants d'une configuration ne se connaissent pas les uns les autres, afin qu'ils puissent communiquer ils doivent avertir la configuration qui se charge de transmettre l'information via les liens.

Pour cela nous avons implémenté un pattern Observateur. Chaque rôle/port étend la classe *Observable*, ce qui permet de notifier l'observateur lorsqu'il reçoit la requête. Chaque composant lui implémente l'interface *Observer*, on ajoute les rôles/ports qu'il doit observer et ce qu'il doit faire quand il reçoit une notification de l'un d'eux. A chaque fois qu'il est notifié, il doit donc transmettre à la configuration le rôle/port appelant et rechercher dans quel lien celui-ci intervient afin de récupérer le rôle/port cible et appeler un service fourni par ce dernier. Il peut ensuite transmettre l'objet à celui-ci et ainsi de suite. Au sein d'un composant, lorsque qu'une requête est arrivée au niveau du port, celui-ci notifie le composant qui à son tour appel un de ses propres services afin de renvoyer la requête. Au niveau de notre lien de Binding, le fonctionnement reste le même ; à la différence près que le *Server* connait le *SeverDetail*. On a donc le serveur qui va récupérer le *ServerDetail* afin que celui-ci récupère le bon port via un lien de Binding.

Afin que notre modèle de transmission fonctionne, nous avons dû ajouter un lien de Binding entre le *ServerDetail* permettant au message de faire le chemin inverse et de revenir dans le *Server*. De même nous avons dû ajouter deux liens d'Attachement, un entre le *Server* et le *RPC*, et le deuxième entre le *RPC* et le *Client* permettant aussi à la requête de revenir chez le client.

Le client fourni des services qui peuvent être appelés de l'extérieur. Nous pouvons donc appeler à partir de notre main le service appelé *sendRequest* qui permet d'envoyer un message. Un message est une classe que nous avons définie ; elle est composée d'un attribut *requete* contenant la requête sous forme de String et d'un attribut appelé *response* permettant de stocker la réponse à la requête (nous avons ici une *List<Object>*).

5.3 Le voyage de la requête

Dans notre exemple, la requête est constituée de la façon suivante :

```
1 Message message = new Message ("SELECT * FROM Person;", null);
```

On utilisera par la suite le service du client afin de transmettre ce message à la base de données. Celle-ci analysera le message et traitera la requête du client, inscrira la réponse dans le message puis renverra celle-ci au client.

On peut avoir un aperçu des différentes « couches » par laquelle le message transite :

```
[ Calling service from Client to send the request ]
[ENTRY] in port SendRequestP
[ ----- Client notify ----- ]
[ENTRY] in role Caller
[ENTRY] in glue Glue1
[ENTRY] in role Called
[ ----- RPC notify ----- ]
[ENTRY] in portReceiveRequestP
[ ----- Server notify ----- ]
[ Calling service from Server to send the request ]
[ENTRY] in port SendRequestP2
[ ----- Server notify ----- ]
[ENTRY] in port ExternalSocket
[ ----- ConnectionManager notify ----- ]
[ CALLING SERVICE ]
[ENTRY] in port DbQuery
[ ----- ConnectionManager notify ----- ]
[ENTRY] in role Caller
[ENTRY] in glue Glue2
[ENTRY] in role Callee
[ ----- SqlQuery notify ----- ]
[ENTRY] in port Query
[ ----- Database notify ----- ]
[ Calling service from Securitymanagement to send the request]
[ENTRY] in port SecurityManagement
[ ----- Database notify ----- ]
[ENTRY] in role Requestor
[ENTRY] in glue Glue3
[ENTRY] in role SecurityManagerR
[ ----- SecurityQuery notify ----- ]
[ENTRY] in port CredentialQuery
[ ----- SecurityManager notify ----- ]
[ Calling service from SecurityAuthorization to send the request ]
[ENTRY] in port SecurityAuthorization
[ ----- SecurityManager notify ----- ]
[ENTRY] in role Grantor
[ENTRY] in glue Glue4
[ENTRY] in role Requestor
[ ----- ClearanceRequest notify ----- ]
[ENTRY] in port SecurityCheck
[ ----- ConnectionManager notify ----- ]
[ CALLING SERVICE ]
[ENTRY] in port SocketResponse
[ ----- ConnectionManager notify ----- ]
[ENTRY] in portReceiveResponseP
[ ----- Server notify ----- ]
[ENTRY] in role CallerResponse
[ENTRY] in glue Glue5
[ENTRY] in role CalledRespons
[ ----- RPC notify ----- ]
[ENTRY] in port ReceiveResponseP2
[ ----- Client notify ----- ]
----------------------------------------
---------- Reponse client ----------
----------------------------------------
Personne : Florian d'ID : 1
Personne : Noemie d'ID : 2
Personne : Chaton d'ID : 3
```

On voit dès le début de la trace l'appel au service du client, on remarque l'entrée dans les différentes ports/services et glues ainsi que le passage d'informations au travers des connecteurs et composants.

5.4 Création des exceptions

Il nous a semblé pertinent de créer des exceptions pour ce projet. Ainsi nous avons créé des exceptions respectant nos contraintes OCL :

- Des exceptions assurant l'existance des rôles, ports et services
- Des exceptions assurant que les ports, rôles et services possèdent la bonne interface (provided ou required)
- Des exceptions assurant que nos liens de bindings et attachements sont construits selon nos normes OCL(si les liens ne sont pas bien construits, cela indique que soit le composant n'a pas été correctement instancié, soit les interfaces utilisées ne sont pas les bonnes)

Lors de l'exécution du programme, un message d'erreur peut ainsi apparaître si les liens ne sont pas correctement construits ou si un composant est manquant.

6 Conclusion et amélioration

Ce projet nous a permi de nous rendre compte des difficultés encourues pour passer de notre métamodèle papier à celui que nous avons développé. Il nous a fallu ainsi faire différents compromis afin de privilégier un code propre, extensible et modulaire qui continuerait d'être fidèle à notre métamodèle de base.

Nous pensons que le travail réalisé permet de tracer le message que le client envoie au serveur et son retour.

Si nous souhaitions être beaucoup plus rigoureux, nous aurions pu ajouter un champ dans notre classe message qui permettrait d'identifier l'utilisateur auprès de la base de données. Si l'utilisateur est référence auprès de la base, il peut interroger celle-ci, sinon l'accès lui serait impossible.

Avec un peu plus de temps, nous aurions pu compléter certaines classes afin de ne pas leur faire contenir uniquement la trace du passage de notre message.