Université Paris Est Marne-la-Vallée
Master 2 Logiciel

Jeremy BIDET
Enzo RAMOS
Jérôme RAMSAMY

Projet C++/Corba : Le Petit Prince

Jeremy BIDET
Enzo RAMOS
Jérôme RAMSAMY

Table des matières

Jeremy BIDET
Enzo RAMOS
Jérôme RAMSAMY

Projet C++/Corba : Le Petit Prince

Description

L'objectif est de créer un programme fonctionnel en C++11 utilisant les concepts de la P.O.O.

L'application devra simuler la réalisation d'une grande fresque murale réalisée à partir de plusieurs dessins d'enfants. Les enfants (les clients) envoient à leur maîtresse (le serveur) leurs dessins afin de recueillir son avis et ses suggestions d'amélioration.
La maîtresse retourne à l'élève son dessin accompagné de ses annotations. La maîtresse peut à tout moment retrouver le dessin d'un enfant.

Le projet a été découpé en trois parties :

- Un projet maître
- Un projet client
- Un projet serveur

Le **projet maitre** fournit les outils nécessaires à la compilation des sous-projets. Il possède le fichier IDL qui définit les interfaces, les structures et les services qui seront utilisés par le client et le serveur.
Il détient également un fichier README permettant d'expliquer comment compiler et exécuter l'application, ainsi que les accès à la documentation.
Le **projet serveur** contient toutes les sources et librairies de l'application, ainsi qu'un main exécutable.
Le **projet client** ne contient que les sources générées par la compilation de l'IDL, ainsi qu'un main exécutable.

Limite du projet

Dû à une incompréhension d'une partie de l'énoncé du projet, nous n'avons pas réussi à déterminer si les maîtresses étaient un type de client ou les serveurs eux-mêmes.
Nous avons donc tranché, si bien que nous avons implémenté les maîtresses comme des serveurs.

Actuellement, il n'y a qu'un seul serveur qui tourne à la fois. Cela dit, nous avons réfléchi à des méthodes afin de palier à ce problème :
La première méthode consiste à utiliser plusieurs threads simulant chacun une maîtresse agissant en local sur le serveur.
La deuxième méthode consiste à lancer plusieurs serveurs utilisant le même service de nommage afin de partager les ressources avec un thread unique simulant la maîtresse.

Répartition du travail

La répartition du travail est détaillée dans le header de chaque fichiers sources.

Jeremy BIDET
Enzo RAMOS
Jérôme RAMSAMY

Fichiers réalisés par BIDET Jeremy :
main_server.cpp
PetitPrinceServiceImpl.hpp
PetitPrinceServiceImpl.cpp
README.md

Fichiers réalisés par RAMOS Enzo:
main_client.cpp
DrawServiceImpl.hpp
DrawServiceImpl.cpp

Fichiers réalisés par RAMSAMY Jérôme :
OBV_Draw.hpp
OBV_Line.hpp
OBV_Circle.hpp
OBV_Ellipse.hpp
OBV_Polygone.hpp

Nous avons tous les trois travaillé sur la conception de l'IDL et sur la génération de la documentation C++.

Prérequis

Nomenclature

Symboles	Description
$	L'invite de commande
~	Le répertoire utilisateur
%ROOT	Le répertoire racine du projet (master)
%HOST	L'adresse du serveur (ip ou localhost)
%PORT	Le port sur lequel le serveur écoute

Pre-Compilation

Il y'a un fichier unique concernant l'IDL nommé *PetitPrince.idl* dans le dossier idl à la racine du projet maître.

La compilation d'un fichier IDL génère trois fichiers : .hpp, _Stub.cpp, _DynStub.cpp.

Pour compiler l'IDL, il est nécessaire d'installer OmniORB et de taper les commandes suivantes :

Jeremy BIDET
Enzo RAMOS
Jérôme RAMSAMY

```
$ cd %ROOT/idl/
$ omniidl -bcxx -Wba -Wbh=.hpp -Wbs=_Stub.cpp -Wbd=_DynStub.cpp -Wbuse_quotes
PetitPrince.idl
```

Cette commande crée les fichiers PetitPrince.hpp, PetitPrince_Stub.cpp et PetitPrince_DynStub.cpp.
Ces trois fichiers se trouvent dans le dossier où a été tapé la commande, c'est à dire %ROOT/idl/.

Il est ensuite nécessaire de copier ces fichiers dans les bons dossiers grâce aux commandes suivantes :

```
cp *.hpp %ROOT/PetitPrince_Client/hdr
cp *.cpp %ROOT/PetitPrince_Client/src
cp *.hpp %ROOT/PetitPrince_Server/hdr
cp *.cpp %ROOT/PetitPrince_Server/src
```

Compilation

Il existe deux façons de compiler le projet.
La première consiste en l'utilisation du Makefile global.

```
$ cd %ROOT/
$ make
```

La seconde consiste à compiler les deux projets indépendamment.

```
$ cd %ROOT/PetitPrince_Server/
$ make

$ cd %ROOT/PetitPrince_Client/
$ make
```

Ces commandes créent deux exécutables :

```
%ROOT/PetitPrince_Client/dist/Debug/GNU-Linux/petitprince_client
%ROOT/PetitPrince_Server/dist/Debug/GNU-Linux/petitprince_server
```

Exécution

Dans un premier temps, il faut démarrer le service de nommage. Dans un nouveau terminal, on entre la commande suivante :

```
$ omniNames &
```

Il est à noter que cette commande peut requérir des privilèges administrateurs.

Ensuite, il faut invoquer le serveur :

```
$ cd %ROOT/PetitPrince_Server/dist/Debug/GNU-Linux
$ ./petitprince_server -ORBInitRef
NameService=corbaloc:iiop:%HOST:%PORT/NameService
```

Enfin, les clients peuvent être lancés dans un nouveau terminal.

```
$ cd %ROOT/PetitPrince_Server/dist/Debug/GNU-Linux
$ ./petitprince_server -ORBInitRef
NameService=corbaloc:iiop:%HOST:%PORT/NameService
```

Jeremy BIDET
Enzo RAMOS
Jérôme RAMSAMY

Architecture

CORBA IDL

Ce module définit les services et les formes.

Pour la conception de l'IDL nous avons pensé à développer deux interfaces au lieu d'une, afin de bien séparer les services de l'application, ainsi que les services liés aux formes.

Dans CORBA, les interfaces définissent un service, composé d'une liste de méthodes destinées à être appelées (en remote) par le(s) client(s).
L'utilisation de CORBA pour ces services, permet ici d'obscurcir l'implémentation côté serveur et l'utilisation côté client des services.
Ainsi, peu importe le client et la technologie utilisée, le client peut appeler de façon autonome les méthodes du service sans se préoccuper de leur implémentation et surtout du langage utilisé.
Le seul prérequis est la compilation de l'IDL côté client vers le langage souhaité.

Nous avons défini deux interfaces et des valuetype pour décrire les formes.
Soient les deux interfaces choisies :

- PetitPrinceService
- DrawService

Soient les valuetype choisis :

- Draw
- Line
- Circle
- Ellipse
- Polygon

1. PetitPrinceService

Cette interface possède 1 exception et 6 méthodes.

L'exception permet d'informer l'utilisateur de l'échec d'ajout d'une forme.
Quatre méthodes permettent la création d'une forme :

- createLine()
- createEllipse()
- createCircle()
- createPolygon()

Tandis qu'une méthode assure la recherche par auteur des formes :

- getDraws()

Jeremy BIDET
Enzo RAMOS
Jérôme RAMSAMY

Et une autre assure la notation d'une forme :

- markDraw()

Axes d'amélioration

Pour la création des formes nous avons pensé à utiliser le design pattern Factory.

Ce dernier aurait permis de n'avoir qu'une méthode de création de formes dans laquelle nous aurions passé en paramètre le type de la forme et ses paramètres.
Nous n'avons pas implémenté ce design par manque de temps lié à la complexité de son développement.
Il aurait fallu en effet, générer une une classe enum dynamique dans laquelle nous aurions pu ajouter/modifier/supprimer des formes dynamiquement (sans retoucher l'IDL).
Mais ceci aurait impliqué l'utilisation du type Any de CORBA sachant que chaque forme possède un nombre de paramètres différent et des types différents pour chaque paramètre.

Cette réalisation n'est pas impossible, car déjà réalisée, mais coûte trop cher en ressources pour ce projet.

Pour la récupération des formes nous aurions pu également ajouter d'autres filtres de sélection, comme par ID, par type, ou par note.
Nous aurions pu également développer une méthode générique (à la manière de la création des formes, cf. premier axe d'amélioration) qui aurait permis de passer un ou plusieurs filtres dans une seule et même méthode.

Les autres filtres indépendants n'ont pas été implémenté, non pas par complexité mais par manque de temps. La méthode générique n'a pas été implémenté pour les mêmes raisons que le premier axe d'amélioration.

2. DrawService

Cette interface possède 2 exceptions et 9 méthodes.

Les exceptions permettent d'informer l'utilisateur que l'opération ne peut être exécutée sur cette forme (i.e. une ligne n'a pas d'air) et également que la forme souhaitée n'existe pas.

Sept méthodes concernent les opérations sur les formes :

- area()
- perimeter()
- homothetie()
- translation()
- rotation()
- symCenter()
- symAxial()

Jeremy BIDET
Enzo RAMOS
Jérôme RAMSAMY

Une méthode assure l'ajout d'une forme dans une autre :

- addDraw()

Une autre méthode assure la méthode d'affichage standard d'une forme :

- toString()

3. Valuetype Draw, Line, Circle, Ellipse, Polygon

Le valuetype Draw définie la classe abstraite avec les champs et méthodes communes.

Les autres valuetype définissent les différentes formes héritant de Draw. Elles implémentent les méthodes avec les formules géométriques.
Certaines méthodes de certains valuetype ne sont pas implémentées car elles n'auraient pas de sens (ex: Line et area), elles renvoient alors l'exception NonApplicable().

La méthode toString() de Draw affiche les champs communs et est appelée par tous les valuetype héritant et affiche les champs spécifiques.

Sources

Cette partie décrit l'implémentation des classes et des choix faits dans l'IDL.

Implémentation des formes

1. Draw

Cette classe est une classe virtuelle qui définit le comportement initial de toutes les implémentations possibles de formes.

Cette classe contient plusieurs méthodes virtuelles pures qui ne sont pas implémentées dans cette classe car l'implémentation est à la discrétion des classes qui l'héritent.
Cependant, elle redéfinit les getters et setters des champs privés de la classe.

Las champs, qui sont communs à toutes les classes qui en héritent sont :

- Id : type long
- Auteur : type string
- Dessins internes : type liste de dessins
- Note : type double

Les classes ci-dessous héritent toutes de la classe virtuelle Draw en redéfinissant les méthodes virtuelles pures et en ajoutant les paramètres nécessaires à la création de la forme désirée.

2. Line

Les méthodes :

- Area() : type double. Elle crée une exception car une ligne n'a pas d'aire
- Perimeter() : type double. Elle crée une exception car une ligne n'a pas de périmètre

Jeremy BIDET
Enzo RAMOS
Jérôme RAMSAMY

- Homothetie() : type void
  On a choisi de translater la forme vers l'origine du plan, puis d'effectuer une homothétie et enfin de re-translater la forme à ses coordonnées initiales.
- Translation() : type void
  On effectue une translation suivant les coordonnées x,y.
- Rotation() : type void
  On calcule la distance entre la ligne et l'origine du plan afin de simuler une rotation trigonométrique de centre O. Ainsi, nous multiplions le rayon (la distance) par le cosinus de l'angle pour la coordonnée x et le sinus de l'angle pour la coordonnée y.
- symCenter() : type void
  On effectue une symétrie centrale par rapport à l'origine du plan (rotation de 180°)

Les champs :
Il existe deux points : start et end. Ils définissent le début et la fin d'une ligne.

3. Circle

Les méthodes :

- Area() : type double. Elle retourne l'aire d'un cercle.
- Perimeter() : type double. Elle retourne le périmètre d'un cercle.
- Homothetie() : type void
  On augmente juste la taille du rayon par rapport au scalaire de l'homothétie.
- Translation() : type void
  On effectue la translation suivant les coordonnées x,y du centre.
- Rotation() : type void
  Il n'y a pas de rotations.
- symCenter() : type void
  Il n'y a pas de symétrie centrale.
- symAxial() : type void
  Il n'y a pas de symétrie axiale.

Les champs :
Il existe un point définissant le centre et un rayon définissant la taille du cercle.

4. Ellipse

Les méthodes :

- Area() : type double.
  Elle retourne l'aire d'une ellipse calculée par la fonction suivante PI*R*r
- Perimeter() : type double.
  Elle retourne le périmètre d'une ellipse selon la formule suivante $PI*\sqrt{(2*R^2*r^2)}$
- Homothetie() : type void
  On augmente la taille des rayons R et r par rapport au scalaire de l'homothétie.
- Translation() : type void
  On effectue la translation suivant les coordonnées x,y du centre.
- Rotation() : type void
  Il n'y a pas de rotations.

Jeremy BIDET
Enzo RAMOS
Jérôme RAMSAMY

- symCenter() : type void
  Il n'y a pas de symétrie centrale.
- symAxial() : type void
  Il n'y a pas de symétrie axiale.

Les champs :
Il existe un point définissant le centre et deux rayons R et r.

5. *Polygone*

Les méthodes :

- Area() : type double.
  On procède par une approximation. L'aire correspond à la somme des coordonnées alternées de chaque point avec son prédécesseur.
  Soit un polygone ABCD, on a

$$\sum_{I \in P} (I.x \times (I+1).y) - ((I+1).x \times I.y)$$

- Perimeter() : type double.
  On calcule le périmètre par la formule suivante :

$$\sum_{I \in P} \sqrt{\left((I+1).x - I.x\right)^2 + \left((I+1).y - I.y\right)^2}$$

- Homothetie() : type void
  On décompose le polygone en lignes successives et on applique l'homothétie à chaque ligne.
- Translation() : type void
  On effectue la translation sur chaque point du polygone.
- Rotation() : type void
  On décompose le polygone en lignes successives et on applique la rotation à chaque ligne.

Les champs :
Il s'agit d'une liste de points.

Implémentation des services

1. *PetitPrinceService*

Le but de cette classe est de fournir au client un service de création et de gestion des formes via l'utilisation unique des id des formes et non des formes elles-mêmes car le client n'a pas besoin de connaître l'implémentation des formes et donc de les manipuler directement.

Cette classe implémente l'interface PetitPrinceService générée par l'IDL après compilation.
La classe PetitPrinceServiceImpl hérite de la version POA générée par l'IDL qui elle-même hérite des classes _impl_ qui héritent elles même des interfaces.

Jeremy BIDET
Enzo RAMOS
Jérôme RAMSAMY

Elle définit l'implémentation de toutes les méthodes contenues dans l'interface de l'architecture.

Les méthodes de création de création de forme prennent toutes un auteur et leurs paramètres respectifs comme arguments.

- CreateLine() : type long
  Elle crée une ligne et l'ajoute au serveur
- CreateCircle() : type long
  Elle crée un cercle et l'ajoute au serveur
- CreateEllipse() : type long
  Elle crée une ellipse et l'ajoute au serveur
- CreatePolygone() : type long
  Elle crée un polygone et l'ajoute au serveur

Le type long correspond à l'Id de la forme créée.

- getDraws(): type liste de long
  Elle cherche toutes les formes qui ont été créées par un auteur donné et retourne la liste des id de ces formes.
- markDraw() : type void
  Elle prend en paramètre l'id et la note de la forme. Elle assigne enfin cette note à la forme.

2. *DrawService*

Le but de cette classe est de fournir au client un service permettant d'effectuer des opérations géométriques sur les formes. La manipulation des formes se fait uniquement par l'intermédiaires des id pour les mêmes raisons citées précédemment.

Cette classe implémente l'interface DrawService générée par l'IDL après compilation.
La classe DrawServiceImpl hérite de la version POA générée par l'IDL qui elle-même hérite des classes _impl_ qui héritent elles même des interfaces.
Elle définit l'implémentation de toutes les méthodes d'application géométrique sur les formes définies par l'architecture.

Les méthodes géométriques :

- Area() : type double
  Elle récupère la forme via son id et effectue un appel remote à la méthode éponyme.
  Elle capture les exceptions NonApplicable qui sont renvoyées lorsque l'opération n'est pas valide pour cette forme.
- Perimeter() : type double.
  Elle récupère la forme via son id et effectue un appel remote à la méthode éponyme.
  Elle capture les exceptions NonApplicable qui sont renvoyées lorsque l'opération n'est pas valide pour cette forme.
- Homothetie() : type void
  Elle récupère la forme via son id et effectue un appel local à la méthode éponyme.
- Translation() : type void

Jeremy BIDET
Enzo RAMOS
Jérôme RAMSAMY

Elle récupère la forme via son id et effectue un appel local à la méthode éponyme.
- Rotation() : type void
  Elle récupère la forme via son id et effectue un appel local à la méthode éponyme.

Les méthodes de gestion :
- addDraw() : type void
  Elle prend en paramètre l'id de la forme parent et l'id de la forme contenue dedans.
  Elle ajoute la forme dans le parent.
- toString() : type string
  Elle renvoie un affichage lisible par l'humain représentant les données de la forme.

CORBA/C++

Serveur

On définit l'ORB et les implémentations des services. On initialise ensuite l'ORB en récupérant la référence sur le RootPOA.
On définit alors le service de nommage appelé NameService puis on initialise les objets d'implémentation.

La référence du servant pour chaque service est récupérée, puis bindée dans le service de nommage en définissant un nom approprié pour chaque service.

On crée un thread permettant de lancer les routines des maîtresses afin qu'elles puissent communiquer avec les clients (les élèves).

On récupère alors le POAManager depuis le RootPOA et on l'active avant de finalement lancer l'ORB (le serveur).

Le main gère les exceptions au cas où l'ORB plante.

Client

On définit et on initialise l'ORB.
On récupère la référence sur le service de nommage, ce qui va nous permettre de récupérer des managers sur les services remote du serveur (PetitPrinceService et DrawService).
Une fois, ces deux managers récupérés, on lance la routine client pour communiquer et interagir avec le serveur.

Dès lors, il est possible pour le client de créer des formes, de récupérer leur id et d'afficher les formes.

Le main client gère également les exceptions internes afin de poursuivre le programme même en cas d'erreur.

Jeremy BIDET
Enzo RAMOS
Jérôme RAMSAMY

Documentation

Une documentation doxygen contenant la documentation C++ des classes ainsi que les diagrammes de classes.

Cette documentation existe pour les deux projets client et serveur sous deux formats : html et LaTex.
La version html peut être observée depuis n'importe quel navigateur. Il suffit d'ouvrir le fichier index.html contenu dans le dossier doc/html/ des sous projets.

Bibliographie

Pour réaliser la partie CORBA et C++, nous avons sollicité l'aide de plusieurs tutoriels :
http://omniorb.sourceforge.net/omni41/omniORB/
http://docs.oracle.com/javase/7/docs/technotes/guides/idl/index.html
https://www.codeproject.com/Articles/24863/A-Simple-C-Client-Server-in-CORBA
http://igm.univ-mlv.fr/~midonnet/Polyamid/Corba/UPEMLV/CoursCORBAUPEMLV2013.pdf

Pour le développement C++ pur, nous avons utilisé les sources du site C++ référence :
http://www.cplusplus.com/reference/

Conclusion

Pour la réalisation de ce projet, la première difficulté a été de concevoir une architecture solide et évolutive avec les technologies CORBA IDL.
D'ailleurs, la solution que nous avons choisie est bonne, mais elle nous oriente vers de nouveaux axes d'amélioration (cf. cités précédemment).

Une autre difficulté a été l'installation d'un compilateur IDL permettant de récupérer les mêmes interfaces indépendamment de la plateforme de développement, sachant que nous utilisons Windows, Linux et Mac OS.
La solution retenue a donc été OmniORB qui est disponible pour toutes ces plateformes et qui compile le C++.

Nous avons eu du mal à implémenter le servant CORBA et la connexion avec le client. Toutefois grâce aux tutoriels nous sommes parvenus à implémenter ce qu'il nous fallait pour le projet.

Déterminer un type IDL pour les objets CORBA qui ne rendent pas un service, mais qui sont plus complexes que des structures n'a pas été simple.
La solution retenue a été l'utilisation des valuetype qui est une notion plus récente dans CORBA et qui permet de définir des objets complexes avec méthodes et attributs sans qu'ils ne soient interfacés comme des services du point de vue CORBA.

Au niveau du C++, nous avons eu du mal à gérer correctement l'héritage entre toutes les classes générées et l'implémentation que nous avons développé.

Jeremy BIDET
Enzo RAMOS
Jérôme RAMSAMY

Certaines classes possédaient des méthodes virtuelles pures qu'on a dû surcharger sans savoir quelle implémentation leur donner. Cela était dû au fait qu'elles étaient internes à CORBA.

Le mapping CORBA vers C++ nous a posé quelques soucis. En effet, dans le cas du type string, CORBA le mappe vers le type char* qui n'est pas objet et donc qui ne possède pas de méthode. De plus, le char* est utilisé en tant que CONST dans les paramètres de méthode, alors qu'il est utilisé sans CONST en tant que valeur de retour ; ce qui a généré des problèmes de const_cast.

Malgré tous ces problèmes, ce projet combiné nous a apporté bien des connaissances et d'affirmer certaines notions propres au C++ telles que l'héritage, la gestion des références et pointeurs, les casts d'un type non primitif vers un type primitif.