IUT de Paris - Rives de Seine
Université Paris Cité

RAPPORT

Projet plateforme de services dynamiques Bri

10 NOVEMBRE 2023

GROUPE 302
LIN Xingtong & ZHANG Anxian

Contents

Présentation du projet

Ce projet a été réalisé dans le cadre de notre formation en BUT troisième année. Le projet a pour but de manipuler la réflexivité en java au niveau de l'introspection, en réalisant une application qui permet de lui-même de lancer d'autre application à la demande du client et modifiable par le développeur.

Ce qui a été réaliser

Services proposer pour les utilisateurs (programmeur, amateur)

Nous avons développé trois services qui permettent aux programmeurs de réaliser des actions sur l'application, il y a : fournir un nouveau service qui respecte la norme bri, mettre à jour un service ainsi que déclarer un changement d'adresse de son serveur ftp.

Du côté de l'amateur, lorsque celui-ci arrive sur serveur, tous les services ajoutés par les programmeurs lui seront proposés.

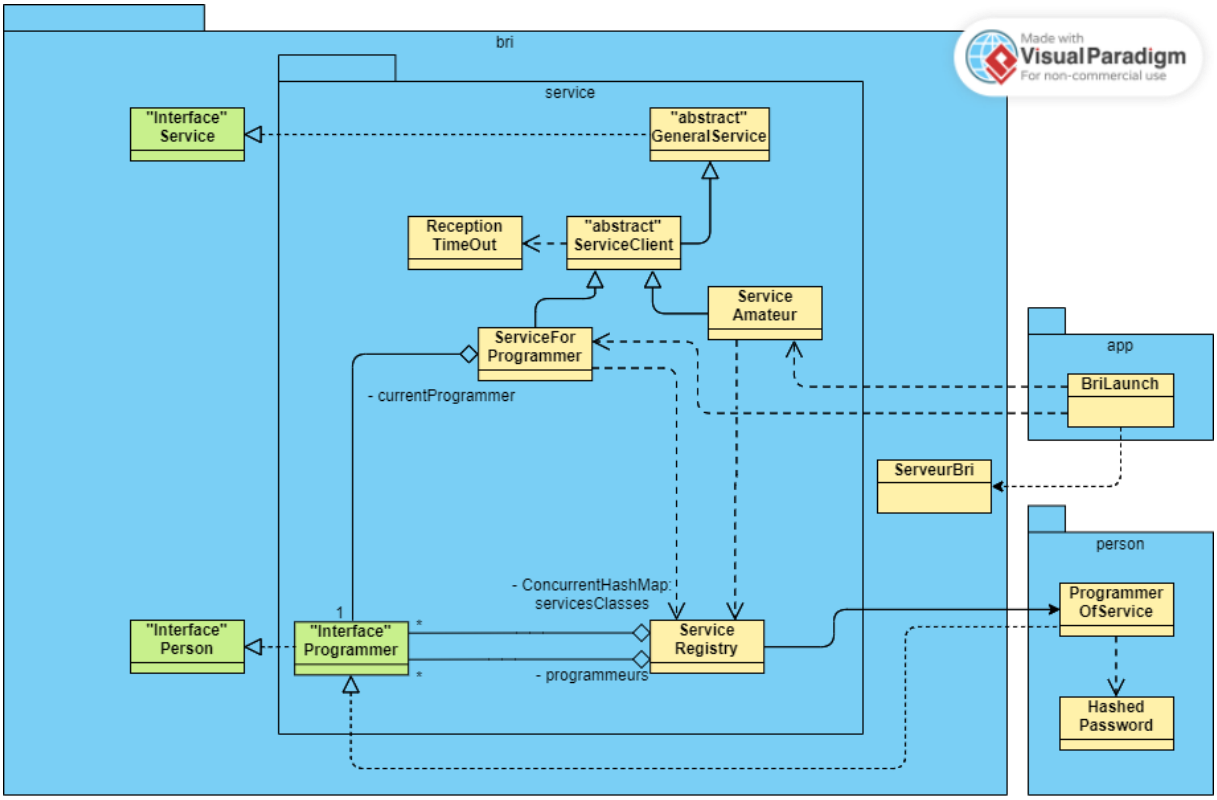
Services fournis par les programmeurs (Anxian et Xingtong)

Afin de tester notre application, nous avons mis en place quatre services, dont trois réalisent des opérations sur une chaine de caractère, inversion de texte, longueur du texte et transformation en majuscule un texte donnée. Le dernier consiste à faire une lecture d'un fichier XML et d'envoyer le résultat dans la boite mail de l'utilisateur, ce service a été réaliser avec javax.mail et javax.activation, il est donc nécessaire d'importer cette bibliothèque au sein du projet (coté serveur) pour pouvoir rendre ce service utilisable à la demande de l'amateur.

Structuration du code

Après avoir mis en place l'ensemble du programme, nous avons fait de notre mieux pour structurer au mieux le code. Nous avons notamment factorisé une partie du code des deux services (Amateur [les non-programmeurs] et Programmeur). Ces deux classes comportent chacune une méthode *numActivityToLaunch()* avec des variations situées au même en endroit, mais avec des valeurs différentes. Nous avons donc décidé d'adopter le design pattern Template Méthode au travers de la classe « ServiceClient ». Ce qui nous amène alors, à ajouter de méthodes abstraites pour les zones variantes du code qui seront spécifiées dans les classes concrètes qui l'étendent.

Diagramme de dépendance



Indication :

L'interface Service ne se trouve pas dans le paquetage service, car il y a une norme à respecter dans le projet, qui est la norme Bri (créer par le professeur), et celui-ci indique qu'une classe fournie doit implémenter l'interface « bri.Service ».

Découplage

Dans la classe « ServiceForPorgrammeur », il existe une méthode *(performServiceAction(String action))* qui consiste d'effectuer les actions d'ajouter ou de mise à jour. Cette méthode comporte trois lignes qui varient en fonction de l'action spécifiée en paramètre. Avant d'opter pour cette solution, nous nous sommes d'abord orientés vers Template Méthode. Cependant, après la mise en place de ce modèle, il y a davantage de dépendance qui s'est rajouté. C'est pour cette raison que nous nous sommes tournés vers la solution d'une méthode avec conditions.

Gestion de la concurrence

Dans le contexte du projet, les deux services (amateur et programmeur) qui sont développés ont besoin d'accéder à une ressource partagée, à savoir la classe « ServiceRegistry ». Cela implique donc un accès concurrentiel de cette classe par les programmeurs et amateurs (les non-programmeurs). Nous devons alors nous assurer que les ressources se trouvant dans « ServiceRegistry » soient thead-safe en intégrant des verrous. Nous avons vu, lors de nos recherches, qu'il y avait des types qui le sont, comme Vector ou bien ConcurrentHashMap, qui sont conçus pour la gestion de concurrence. Nous les avons donc utilisés. Cependant, bien que ces deux classes soient thread-safe pour les opérations individuelles comme *put(), set()* ou encore *get()*, l'utilisation d'un itérateur n'est pas atomique. Cela peut en conséquence entrainer des problèmes de concurrence si d'autre thread les modifient pendant que nous itérons dessus. Il est donc essentiel de verrouiller la ressource en question lors des itérations, et c'est ce que nous avons fait.

Les échanges client / serveur

Les clients se connectent au serveur en utilisant le port dédié. Afin de garantir la stabilité du code client, nous avons décidé de récupérer ce numéro de port à partir des arguments de l'application client, plutôt que de modifier le code client directement et de devoir le recompiler.

Les échanges entre le client et le serveur sont gérés de la manière suivante : le client attend une réponse du serveur, l'affiche, et lorsque cela est nécessaire, l'application serveur demande à l'utilisateur d'entrer une donnée dans la console. Cette donnée est ensuite renvoyée au serveur. Cet échange se poursuit jusqu'à ce que le client, via sa réponse, indique qu'il souhaite arrêter les échanges (notamment grâce à la saisie du mot-clé *quit*) ou soit par une déconnexion automatique au bout de 10min d'inactivité.

Comme les BufferReader et PrinterWriter ne prennent pas en compte les retours chariot, nous avons mis en place une solution de cryptage avec la méthode *public String replace (old, new)* de la classe String. Avant d'envoyer la chaine au client, nous remplaçons les caractères « \n » par des « ## ». Lorsque le client reçoit le message, il décrypte avec la même méthode les « ## » par des « \n ». Ainsi, nous utilisons une application BTTP 2.0.

Les tests

Nous avons dans un premier temps fait des tests sur nos appareils respectifs en locale, tout marche sans encombre. Puis, nous nous sommes orientés vers des tests en passant par le wifi. Il suffit, de lancer le serveur sur un post, et de se connecter au serveur avec l'adresse IP sur lequel il se trouve. Nous avons fait de même pour le serveur FTP. Et tout fonctionne correctement.

Maintenances évolutives

Ajout d'un nouveau service

Si nous voulons ajouter un nouveau service, par exemple un service pour les commentateurs, qui pourront ajouter des commentaires, ou alors donnée des notes au port 7777, il faut :

1. Étendre la classe « ServiceClient »
2. Implémenter les méthodes abstraites de la classe parente
3. Ajouter ce nouveau service dans l'application du serveur, qui sera lancé au nouveau port

Et si c'était une application web ?

Pour faciliter la transition de notre application basée sur serverSocket/socket, nous pouvons l'implémenter avec l'aide d'un Framework, en l'occurrence JEE. Chaque service se trouvant dans l'application deviendra alors une servlet (une URL). Effectivement, puisque nous passons à une application web, nous n'avons plus besoin de manipuler des sockets. Nous allons plutôt récupérer les données entrées par les utilisateurs à travers des champs de formulaire HTML/JSP, et les envoyer au serveur via des requêtes HTTP.

Pour afficher les résultats ou les messages d'erreurs, nous allons remplacer les PrintWriter par une réponse HTTP envoyée au navigateur client, qui pourra être affichée dans une fenêtre pop-up ou dans une zone de texte dédiée sur la page web.

Les améliorations possibles

Qualité du code

En regardant notre diagramme de classe, nous nous sommes dit qu'on pouvait mieux faire, surtout pour le paquetage service. De notre point de vue, toutes les classes se trouvant à l'intérieur ont leurs places, mais d'un autre côté, nous trouvons qu'il en contient trop.

Expérience utilisateur

Nous pensons qu'il sera mieux d'utiliser le format HTML pour envoyer des e-mails dans de service « ServiceAnalyseXML », car cela permet une présentation plus esthétique et une amélioration de l'expérience utilisateur.

Nous pouvons aussi ajouter un compteur pour indiquer aux utilisateurs le temps avant le time-out de leur session. Par exemple, afficher un message "Réponse dans 5 minutes" avant chaque réponse du service, avec le message et le temps restant changeant au fur et à mesure du temps.