Conception Logicielle - TD n°3

Ce TD consiste en la création d'un *pattern* serveur concurrent pour permettre de réutiliser le code des TD précédents dans d'autres projets.

Etape 1 : Inversion de dépendance entre le serveur et la banque

Pour casser la dépendance entre le serveur et la banque, le serveur ne doit doit plus référencer une banque. C'est à dire que le serveur ne doit plus être utilisé avec un objet "banque" (qui était jusque là implémenté par la classe Banque à partir de l'interface IBanque) mais d'un objet quelconque.

Création d'un objet quelconque

On crée une interface IObjet, qui correspondra à un objet utilisé avec le serveur. Cet objet ne doit contenir aucune méthode particulière donc l'interface est presque vide.

Modification de l'objet Banque

Puisque la banque doit toujours pouvoir être utilisée avec le serveur, il faut qu'elle implément l'interface IObjet.

```
public class Banque implements IBanque, IObjet{...}
```

Modification du serveur TCP

Puisqu'on cherche à casser la dépendance entre la banque et le serveur, le gros du travail sera la modification de la classe ServeurTCP.
Le serveur ne doit plus dépendre de la banque, donc on remplace la variable:

```
public IBanque banqueCentrale;
```

par:

```
public IObjet objet;
```

Cette modification génère des erreurs dans le fichier puisque le serveur est écrit pour être utilisé avec une banque.

On modifie le constructeur ServeurTCP pour ne plus utiliser la banque:

```
public ServeurTCP(IBanque b, int port) {
        this(port);
        banqueCentrale = b;
}
```

```
public ServeurTCP(IObjet b, int port) {
        this(port);
        objet = b;
}
```

On modifie ensuite le *getter* et le *setter* de la variable associée à la banque:

```
public void setBanqueCentrale(IBanque uneBanque) {
   banqueCentrale = uneBanque;
}

public IBanque getBanqueCentrale() {
   return banqueCentrale;
}
```

```
public void setObjet(IObjet unObjet) {
   objet = unObjet;
}

public IObjet getObjet() {
   return objet;
}
```

Modification temporaire de ServeurSpecifique

Pour pouvoir executer le code à cette étape, il faut modifier également la classe ServeurSpécifique.

```
int valeurRetrait = monServeur.getBanqueCentrale().demandeRetrait(valeur);
```

devient:

```
int valeurRetrait = ((IBanque)
monServeur.getObjet()).demandeRetrait(valeur);
```

On fait un *cast* de IObjet vers IBanque (càd qu'on converti l'objet en IBanque) pour avoir accès à la méthode demandeRetrait.

```
int valeurDepot = monServeur.getBanqueCentrale().demandeDepot(valeur);
```

devient:

```
int valeurDepot = ((IBanque) monServeur.getObjet()).demandeDepot(valeur);
```

et, pour finir:

```
for (String h : monServeur.getBanqueCentrale().getHistoriqueOperations())
{
   outputString = outputString + h + "\n";
}
```

devient:

```
for (String h : ((IBanque)
monServeur.getObjet()).getHistoriqueOperations()) {
   outputString = outputString + h + "\n";
}
```

Etape 2 : Séparer les responsabilités

Dans la classe ServeurSpecifique, tous les échanges sont gérés par la même méthode (run). Pour appliquer le principe de séparation des responsabilités, on crée des classes différentes pour chaque protocole (retrait et dépôt).

On crée donc une classe **ProtocoleRetrait** et une classe **ProtocoleDepot** pour remplir cette fonction.

Ensuite, on reprend les instructions de la méthode *run* pour remplir les deux nouvelles classes.

Protocole de retrait

La partie de la méthode *run* qui gère le retrait est:

```
if (chaines[0].contentEquals("retrait")) {
   int valeur = (new Integer(chaines[1])).intValue();
```

```
 System.out.println(" valeur demandee  " + valeur);

 int valeurRetrait = ((IBanque)
monServeur.getObjet()).demandeRetrait(valeur);

 String valeurExpediee = "" + valeurRetrait;
 System.out.println(" Retrait dans serveur " + valeurExpediee);

 os.println(valeurExpediee);

 System.out.println(monServeur);
}
```

La classe ProtocoleRetrait doit donc au moins contenir ces instructions pour maintenir le fonctionnement de l'application:

```
package banqueServeur;

import java.io.PrintStream;


/**
 * Protocole de retrait utilisé dans {@link ServeurSpecifique}.
 */
public class ProtocoleRetrait {
    public int valeurDemandee;
        public ServeurTCP monServeur;
        public PrintStream os;

        public ProtocoleRetrait(int valeurDemandee, ServeurTCP monServeur,
PrintStream os) {
        this.valeurDemandee = valeurDemandee;
                this.monServeur = monServeur;
                this.os = os;
        }

    public void run(){
        System.out.println(" valeur demandee  " + valeurDemandee);
                int valeurRetrait = ((IBanque)
monServeur.getObjet()).demandeRetrait(valeurDemandee);
                String valeurExpediee = "" + valeurRetrait;
                System.out.println(" Retrait dans serveur " +
valeurExpediee);
                os.println(valeurExpediee);
                System.out.println(monServeur);
        }

}
```

On remplace alors la partie de la méthode *run* dont le code figure ci-dessus par:

```
if (chaines[0].contentEquals("retrait")) {
   int valeur = parseInt(chaines[1]);
        ProtocoleRetrait protocoleRetrait = new ProtocoleRetrait(valeur,
monServeur, os);
        protocoleRetrait.run();
}
```

(On se débarasse au passage de la classe Integer qui est *deprecated*)

Protocole de dépôt

On procède de la même manière que pour le retrait.
Le code de la méthode *run* qui gère le dépôt est:

```
if (chaines[0].contentEquals("depot")) {
   int valeur = (new Integer(chaines[1])).intValue();

 System.out.println(" valeur demandee  " + valeur);

 int valeurDepot = ((IBanque) monServeur.getObjet()).demandeDepot(valeur);

 String valeurExpediee = "" + valeurDepot;
 System.out.println(" Depot dans serveur " + valeurExpediee);

 os.println(valeurExpediee);

 System.out.println(monServeur);
}
```

On écrit donc la classe ProtocoleDepot avec les mêmes instructions:

```
package banqueServeur;

import java.io.PrintStream;


/**
 * Protocole de dépôt utilisé par {@link ServeurSpecifique}.
 */
public class ProtocoleDepot {
    public int valeurDemandee;
        public ServeurTCP monServeur;
        public PrintStream os;
```

```
        public ProtocoleDepot(int valeurDemandee, ServeurTCP monServeur,
PrintStream os) {
        this.valeurDemandee = valeurDemandee;
                this.monServeur = monServeur;
                this.os = os;
        }

    public void run(){
        System.out.println(" valeur demandee  " + valeurDemandee);
                int valeurDepot = ((IBanque)
monServeur.getObjet()).demandeDepot(valeurDemandee);
                String valeurExpediee = "" + valeurDepot;
                System.out.println(" Depot dans serveur " +
valeurExpediee);
                os.println(valeurExpediee);
                System.out.println(monServeur);
        }
}
```

Et la méthode *run* devient:

```
if (chaines[0].contentEquals("depot")) {
   int valeur = parseInt(chaines[1]);
        ProtocoleDepot protocoleDepot = new ProtocoleDepot(valeur,
monServeur, os);
        protocoleDepot.run();
}
```

Etape 3 : Inversion de dépendance entre les protocoles et le serveur

Maintenant que les protocoles sont séparés de la classe ServeurSpecifique, il faut casser la dépendance entre ces classes et le serveur spécifique.
Pour cela, on va appliquer un pattern stratégie.

Création d'une interface protocole

Puisqu'on cherche à casser la dépendance entre les classes de protocoles et la classe de serveur, on crée une interface IProtocole, qui contient la méthode *run*.

```
package banqueServeur;

/**
 * Interface correspondant aux protocoles à réaliser par le serveur
spécifique. * {@link ServeurSpecifique}
 */
```

```
public interface IProtocole {
    public void run();
}
```

On ajoute dans les classes ProtocoleRetrait et ProtocoleDepot l'implémentation de l'interface IProtocole:

```
public class ProtocoleRetrait implements IProtocole{...}
public class ProtocoleDepot implements IProtocole{...}
```

Modification du serveur spécifique

Maintenant que l'interface IProtocole est écrite, on va s'inspirer de l'implémentation du pattern stratégie dans la banque pour les retrait et les dépôts pour mettre en place cette version.

On ajoute à ServeurSpecifique une variable d'instance qui représentera le protocole:

```
private IProtocole protocoleEchange;
```

On ajoute un *getter* et un *setter* pour cette variable (utiliser alt+inser dans IntelliJ pour l'écriture automatique de getter/setter/constructeur...).

```
public IProtocole getProtocoleEchange() {
   return protocoleEchange;
}

public void setProtocoleEchange(IProtocole protocoleEchange) {
   this.protocoleEchange = protocoleEchange;
}
```

Ensuite, pour généraliser le code dela méthode *run* de *ServeurSpecifique*, on va également créer un protocole pour gérer la requête de l'historique, puisque c'est le dernier protocole qui reste lié au serveur spécifique.

Protocole de requête d'historique

On crée une classe ProtocoleRequeteHistorique qui implémente l'interface IProtocole. On transfère dans cette classe le contenu de la méthode *run* de *ServeurSpecifique* qui gère les requêtes d'historique:

```
package banqueServeur;

import java.io.PrintStream;

/**
* Protocole de requête de l'historique des transactions utilisé dans
{@link ServeurSpecifique}
*/
public class ProtocoleRequeteHistorique implements IProtocole{
    private ServeurTCP monServeur;
        private PrintStream os;

        public ProtocoleRequeteHistorique(ServeurTCP monServeur,
PrintStream os){
        this.monServeur = monServeur;
                this.os = os;
        }

    @Override
        public void run() {
        String outputString = "Liste des operations :\n";
                for (String h : ((IBanque)
monServeur.getObjet()).getHistoriqueOperations()) {
            outputString = outputString + h + "\n";
                }
        System.out.println(" Liste a envoyer : \n " + outputString);
                os.println(outputString);
                os.flush();
        }
}
```

L'ajout de ce protocole dans ServeurSpécifique transforme la partie de *run* qui gère les requêtes d'historique en:

```
if (chaines[0].contentEquals(automateClient.Historique.requeteHisto)) {
        ProtocoleRequeteHistorique p = new
ProtocoleRequeteHistorique(monServeur, os);
        p.run();
```

Généralisation du choix de protocole

On va maintenant reprendre la méthode *run* pour casser sa dépendance avec les différents protocoles. Pour déterminer quel protocole utiliser, on va encore une fois séparer les responsabilités en créant une classe Contexte (merci à Madeleine pour l'idée de cette partie).

```java
package banqueServeur;

import java.io.IOException;
import java.io.PrintStream;

import static java.lang.Integer.parseInt;

/**
 * Classe permettant de gérer le choix entre les différents protocoles
{@link IProtocole} dans {@link ServeurSpecifique}
 */
public class Contexte {
    public String[] chaine;
        public ServeurTCP monServeur;
        public PrintStream os;

        public Contexte(String[] chaine, ServeurTCP monServeur,
PrintStream os) {
        this.chaine = chaine;
                this.monServeur = monServeur;
                this.os = os;
        }

        public IProtocole getProtocole() throws IOException {
        String comType = chaine[0];
                String value = "";
                if(chaine.length > 1){
                value = chaine[1];
                }
        switch (comType){
            case "retrait":
                return new ProtocoleRetrait(parseInt(value), monServeur,
os);
                        case "depot":
                return new ProtocoleDepot(parseInt(value), monServeur,
os);
                        case automateClient.Historique.requeteHisto:
                return new ProtocoleRequeteHistorique(monServeur, os);
                        default:
                os.println("Erreur de protocole..... \n");
                                throw new IOException("Protocole
Inexistant");
                }
    }
}
```

(pas besoin de *break* dans le *switch* puisque chaque cas utilise un *return*)
La méthode *run* de *ServeurSpecifique* devient donc:

```
public void run() {
        String inputReq;

        try {
        BufferedReader is = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));
                PrintStream os = new
PrintStream(clientSocket.getOutputStream());
                System.out.println("Serveur avec  Client ");

                if ((inputReq = is.readLine()) != null) {
                System.out.println(" Msg 2 Recu " + inputReq);
                        String chaines[] = inputReq.split(" ");
                        System.out.println(" Ordre Recu " + chaines[0]);
                        try {
                contexte = new Contexte(chaines, monServeur, os);

setProtocoleEchange(contexte.getProtocole());
                                protocoleEchange.run();
                        }catch (IOException e){
                os.println("Erreur de protocole..... \n");
                        }
      }
    clientSocket.close();
        os.close();
        is.close();

        } catch (IOException e) {
      e.printStackTrace();
        }
}
```

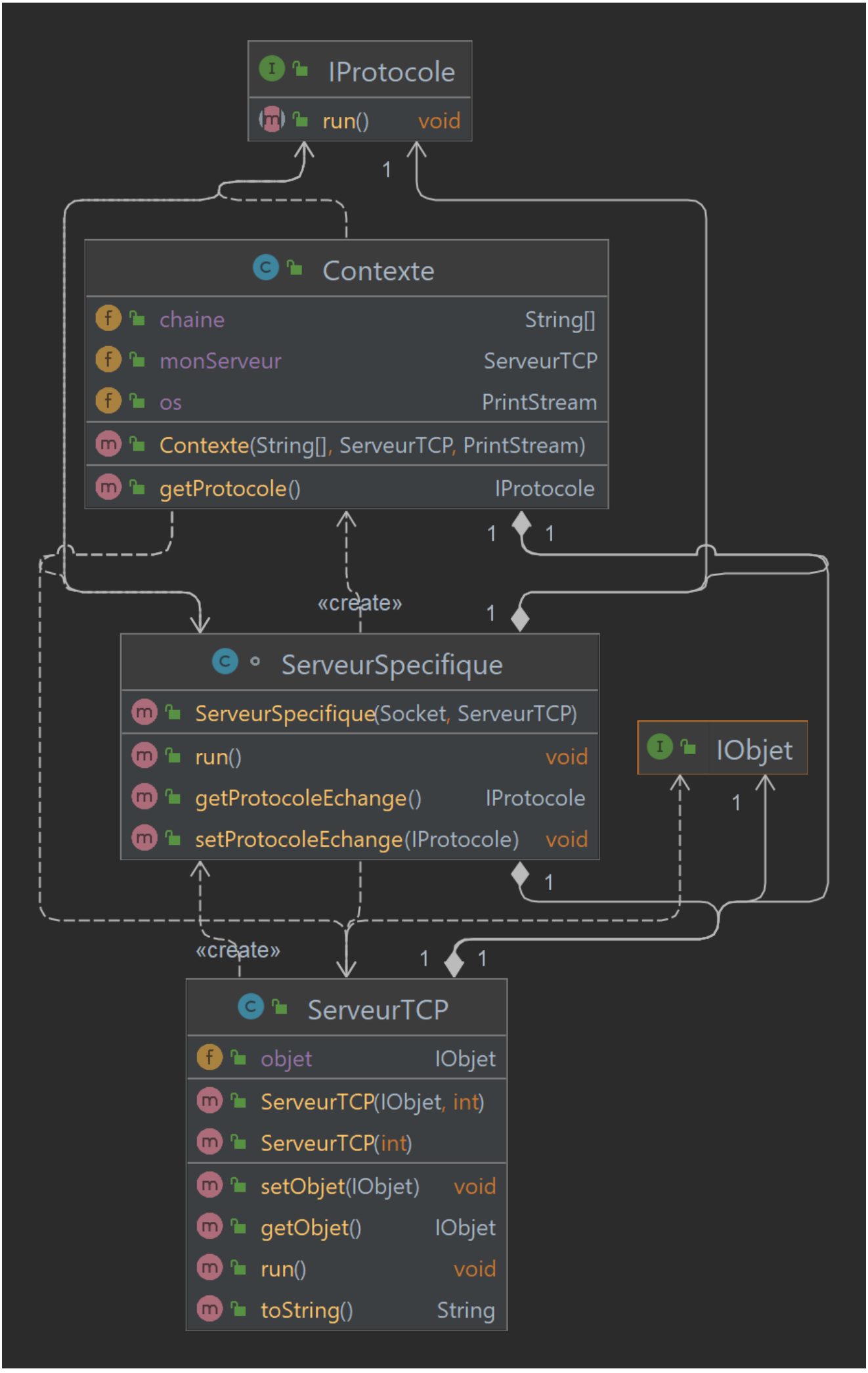
Pour encore plus d'inversion de dépendances, on auait pu créer une interface IContexte pour casser la dépendance de ServeurSpecifique avec Contexte.

Conception du pattern

Maintenant qu'on a réussi à tout modifier, à casser les dépendance et à séparer les responsabilités, on va détailler le pattern qu'on vient de créer pour le rendre réutilisable.

Le nouveau pattern est *Serveur Concurrent* (vous pouvez choisir un autre nom).
Il peut être utilisé dans un projet qui necessite un serveur TCP ayant plusieurs protocoles d'échanges.

Son diagramme de classe est le suivant:


IProtocole
run() void
1
Contexte
chaine String[]
monServeur ServeurTCP
os PrintStream
Contexte(String[], ServeurTCP, PrintStream)
getProtocole() IProtocole
1
1
«create»
1
ServeurSpecifique
ServeurSpecifique(Socket, ServeurTCP)
run() void
getProtocoleEchange() IProtocole
setProtocoleEchange(IProtocole) void
IObjet
1
1
«create»
1
1
ServeurTCP
objet IObjet
ServeurTCP(IObjet, int)
ServeurTCP(int)
setObjet(IObjet) void
getObjet() IObjet
run() void
toString() String

Le code source est disponible sur GitHub à l'adresse:
https://github.com/CorentinGoet/ConceptionLogicielle-TD3