Programmation par composants avec EJB

UV IDL

Correction

Objectifs

L'objectif de ce TP est de découvrir les bases du développement avec des composants EJB. Pour cela, vous utiliserez une plate-forme Java EE et, en particulier, le serveur d'applications Glassfish.

À la fin de l'activité, l'étudiant doit être capable de :

- Développer des composants EJB simples en utilisant Eclipse
- Déployer une application constituée de plusieurs composants EJB
- Documenter une application simple à base de composants en utilisant UML
- Expliquer l'intérêt des intercepteurs et en développer un dans le cadre des EJBs

Préliminaires

Les outils

Pour réaliser le TP, vous utiliserez une machine virtuelle VMWare pré-configurée avec les outils nécessaires au TP. Il s'agit de la machine virtuelle *Ubuntu14_BD_PROG*. Elle contient, entre autres, l'IDE Eclipse dans sa version Java EE et un serveur Glassfish sur lequel vous déploierez les composants et services développés. Le document « Description de l'environnement de travail », disponible sur Moodle, introduit brièvement la virtualisation de manière générale puis présente les caractéristiques de la machine virtuelle que nous avons mis à votre disposition ainsi que son utilisation.

Les technologies

Pendant ce TP vous allez utiliser et programmer de manière très basique les composants EJB (*« Entreprise Java Beans »*) de type session. Il s'agit de la technologie Java pour le développement de composants à utiliser dans un contexte réparti (ce sont des composants qui peuvent être

accessibles à distance) et transactionnel (toutes les opérations réalisées par un composant le sont au sein d'une transaction).

Le modèle de composants EJB est un modèle « plat », c.-à-d. qu'un composant EJB ne peut pas être constitué d'autres composants. La plate-forme d'exécution des composants EJB est appelée *conteneur EJB*. Ce conteneur offre aux composants des services destinés à faciliter notamment la programmation des transactions, de la sécurité et de l'accès à distance. Il gère également le cycle de vie des composants. C'est pour ces raisons qu'ils sont utilisés pour contenir la logique métier dans une application 3-tier écrite en Java.

Depuis la version 3.1 de la spécification, dans ce modèle de composants, chaque composant **peut** déclarer son/ses interface/s. Un composant qui ne déclare pas d'interface peut être utilisé *via* son implantation, ce qui va à l'encontre de la programmation par composants... mais après tout, c'est le programmeur qui décide !!

L'unité de déploiement dans le cas du modèle EJB est appelée « module EJB » et correspond à une archive .jar. Un module EJB peut contenir un ou plusieurs composants EJB et toute autre classe Java (ou autres resources) qui soit nécessaire.

Exercice 1 (*Le `HelloWorld` multi-implantation*)

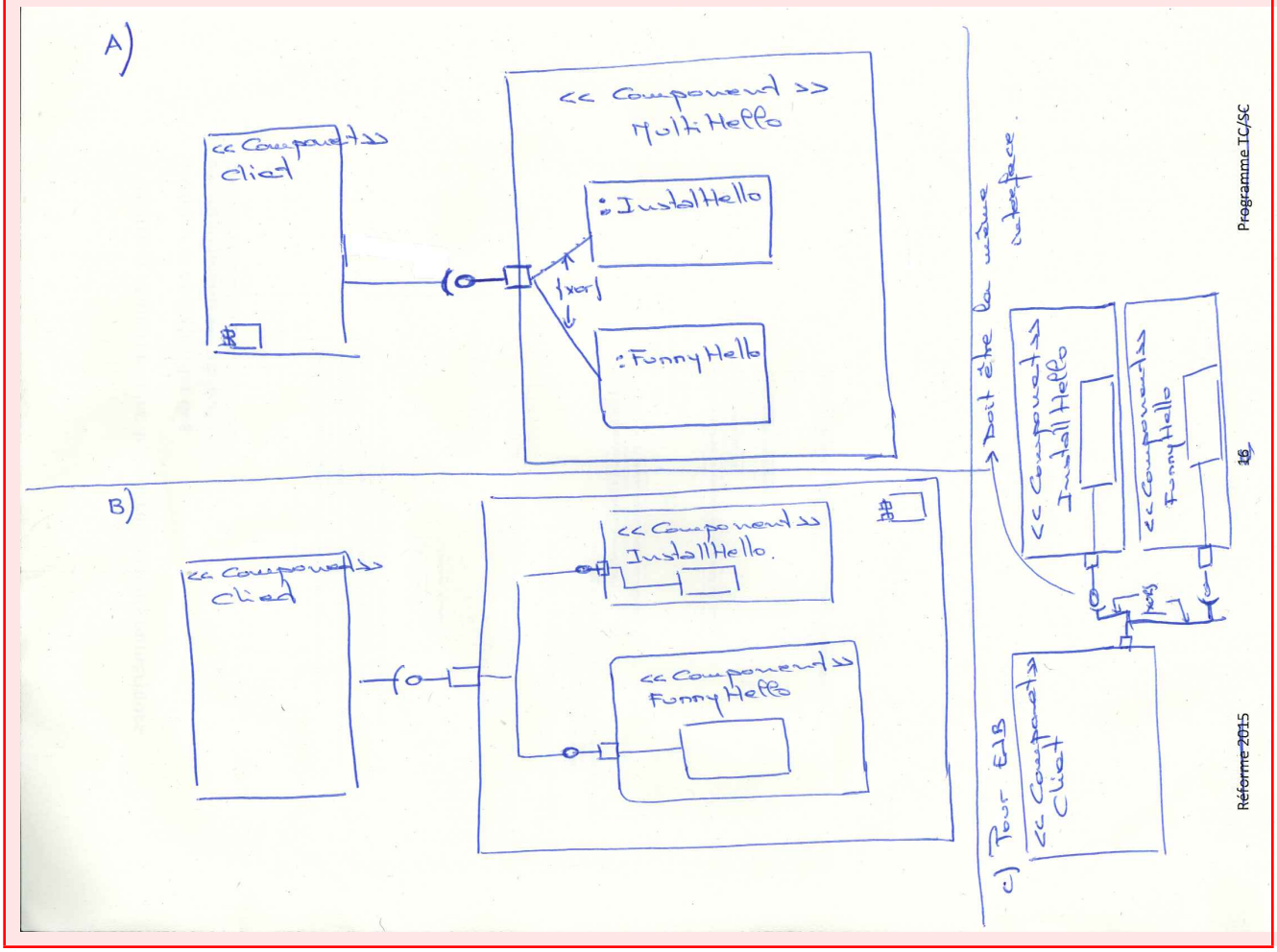
Nous souhaitons développer une application *Hello World !!* multi-implantation. L'application doit offrir deux implantations de ce classique : une implantation qui renvoie le texte *Hello guys !!* et une autre qui renvoie le texte *Hello* suivi d'un nom *aléatoire*[1].

Regardez les 7 premières minutes de la vidéo `https://www.youtube.com/watch?v=KQUGFFN4M90` pour avoir une introduction aux diagrammes de composants UML. Un autre site qui vous renseigne de manière moins ludique sur les bases des diagrammes de composants est ici `www.uml-diagrams.org/component-diagrams.html`

▷ **Question 1.1 :**
Avec un papier/crayon, créez le diagramme de composants correspondant à votre application.

Dans la correction il y a trois options. Elles sont toutes valables. La troisième (C) est la seule qui peut être implantée avec des EJB.

1. La manière de générer ce nom n'est pas importante ici. D'ailleurs plusieurs stratégies pourraient être considérées...



Exercice 2 (*Votre premier EJB*)

Lancez la machine virtuelle si ce n'est pas encore fait. En utilisant Eclipse créez un projet de type EJB.

▷ **Question 2.1 :**
Demandez la création d'un premier composant *Session bean* sans état (*Stateless*) dont l'implantation sera le renvoi du texte *Hello guys ! !*. Attention, lors de la création demandez la création d'une interface *Bussiness Interface* pour qu'un client puisse l'utiliser.

Attention, par défaut Eclipse ne génère pas d'interfaces pour un composant EJB ! ! Il faut penser à décocher la case ...

++++++++++++++++++ **Interface** ++++++++++++++++++

```
package helloServices;
import javax.ejb.Local;
@Local
public interface HelloInterface {
  String sayHello();
```

++++++++++++++++++ **Implantation** ++++++++++++++++++

```java
package helloServices;
import javax.ejb.Stateless;
/**
 * Session Bean implementation class InstantHello
 */
@Stateless
public class InstantHello implements HelloInterface {
  public InstantHello() {
  }
  @Override
  public String sayHello() {
    return "Hello guys!!";
  }
```

Pour exécuter votre application il faut d'abord la déployer sur le conteneur EJB. Pour ce faire, utilisez l'onglet (la vue) *Servers* d'Eclipse, sélectionnez le serveur Glassfish et, à l'aide du menu contextuel [2], demandez à ajouter votre projet. Pensez également à démarrer l'exécution du conteneur ... Notez qu'à chaque modification du composant, le conteneur EJB se désynchronise et doit être resynchronisé pour prendre en compte les changements.

▷ **Question 2.2 :**
Regardez les messages affichés par le conteneur EJB sur la console d'Eclipse. Quels sont les composants EJB qui ont été déployés ?

Il y a un seul composant qui a deux noms. Dans les deux cas, le nom de la classe qui implante le composant est inclue dans le nom ... Ce n'est donc pas de la programmation de services (contrat) mais de composants ...

Créez maintenant, dans le même projet un composant client de type *Stateless*. Ce composant sera utilisé par une application Java classique qui fera le rôle de *bootstrap* de votre application. À nouveau, pensez à demander la création d'une interface *Bussiness Interface* pour votre composant client. L'interface contiendra une seule méthode, par ex. `start()`, dans laquelle le composant serveur sera appelé.

▷ **Question 2.3 :**
L'interface à créer pour le composant client, doit être de type distante ou local ?

L'interface devra être de type *Remote* pour que l'application de *bootstrap*, s'exécutant dans une autre JVM que celle du serveur GlassFish, puisse l'utiliser.

Les conteneurs EJB utilisent, depuis la version 3.0, l'injection de dépendances pour la création d'instances de composants EJB. Pour injecter une instance d'un composant EJB il faut utiliser l'annotation `@EJB` lors de la déclaration de la référence au composant.

▷ **Question 2.4 :**
Ajoutez l'implantation de la méthode `start()` de votre composant client.

Attention, il faut utiliser l'injection de dépendances pour initialiser la référence au composant serveur. En EJB ce type d'injection se fait avec l'annotation `@EJB`.

2. Il s'agit du menu qui apparaît lorsqu'on clique le bouton droit.

Si l'injection n'est pas utilisée, à l'exécution `NullPointerException` sera levée. La correction ici correspond à la réponse de la première question de l'exercice 3.
Dans la correction de cet exercice pas besoin de donner le `beanName` dans l'annotation `@EJB`.
++++++++++++++++++ Interface Client ++++++++++++++++++

```
package helloClient;
import javax.ejb.Remote;
@Remote
public interface IHelloClient {
  void start();
}
```

++++++++++++++++++ Implantation Client ++++++++++++++++++

```
package helloClient;
import helloServices.*;
import javax.ejb.*;
/**
 * Session Bean implementation class HelloClient
 */
@Stateless
public class HelloClient implements IHelloClient {
  //@EJB(beanName="InstantHello")
  @EJB(beanName="FunnyHello")
  private HelloInterface bean;
  @Override
  public void start() {
    // Call the server
```

Déployez le nouveau projet sur le conteneur EJB si ce n'est pas encore fait. Vous devriez maintenant voir s'afficher dans la console d'Eclipse, le nom des deux composants EJB que vous venez de créer.

Pour tester votre application vous allez utiliser une application Java classique qui utilise l'interface de votre composant client.

Créez un nouveau projet Eclipse qui contiendra les sources de cette application Java.

▷ **Question 2.5 :**
Est-il possible d'utiliser l'injection de dépendances dans le code de votre application *bootstrap* ? Pourquoi ?

Non, l'application ne s'exécute pas dans la JVM du serveur d'applications GlassFish, donc, il ne peut pas initialiser la référence au composant client.

▷ **Question 2.6 :**
En utilisant la documentation disponible ici `https://glassfish.java.net/javaee5/ejb/EJB_FAQ.html#StandaloneRemoteEJB` et ici `https://glassfish.java.net/javaee5/ejb/EJB_FAQ.html#What_is_the_syntax_for_portable_global_`, développez le code de votre application. Attention, pour qu'elle puisse utiliser un composant EJB disponible sur Glassfish, il faut ajouter dans son *classpath* l'archive `gf-client.jar` qui est disponible dans le répertoire d'installation de GlassFish.

Deux ajouts dans le *Build Path* du projet :

— Comme dit dans la question, ajouter *chemin_install_glassfish*/lib
— Pour résoudre les problèmes de compilation qui apparaissent, ajouter le chemin aux binaires du projet contenant le composant utilisé. Une manière de faire : sélectionner le projet à utiliser et, à l'aide du menu contextuel, sélectionner *Exporter*. Ajouter ensuite le .jar généré dans le *classpath* du projet *bootstrap*.

```
package eu.telecom_bretagne.front;
import helloClient.IHelloClient;
import javax.naming.*;
public class Main {
  public static void main(String[] args) {
    InitialContext ctx;
    try {
      System.out.println("Démarrage ... ");
      ctx = new InitialContext();
      IHelloClient bean = (IHelloClient) ctx
          .lookup(args[0]);
      System.out.println("Composant trouvé ... ");
      bean.start();
      System.out.println("Fin de l'appel ... ");
    } catch (NamingException e) {
      e.printStackTrace();
    }
  }
}
```

▷ **Question 2.7 :**
On souhaite maintenant ajouter la possibilité d'avoir un autre message de bienvenue : *Hello* suivi du nom de la personne. Quelles modifications devez vous apporter à votre application ? Effectuez les et testez.

Il faut ajouter une méthode `String sayHello(String name)` à l'interface du composant serveur et bien sur l'implémenter dans la classe correspondante. Le composant client doit être modifié pour qu'il utilise la nouvelle méthode. Par contre, pas de changements d'architecture.
++++++++++++++++++ Interface ++++++++++++++++++

```
package helloServices;
import javax.ejb.Local;
@Local
public interface HelloInterface {
  String sayHello();
  String sayHello(String name);
}
```

++++++++++++++++++ Implantation Serveur ++++++++++++++++++

```
package helloServices;
import javax.ejb.Stateless;
/**
 * Session Bean implementation class InstantHello
 */
@Stateless
```

```java
public class InstantHello implements HelloInterface {
  public InstantHello() {
  }
  @Override
  public String sayHello() {
    return "Hello guys!!";
  }
  @Override
  public String sayHello(String name) {
    return "Hello " + name + "!!";
  }
}
```

++++++++++++++++++ Interface Client ++++++++++++++++++

```java
package helloClient;
import javax.ejb.Remote;
@Remote
public interface IHelloClient {
  void start();
}
```

++++++++++++++++++ Implantation Client ++++++++++++++++++

```java
package helloClient;
import helloServices.*;
import javax.ejb.*;
/**
 * Session Bean implementation class HelloClient
 */
@Stateless
public class HelloClient implements IHelloClient {
  //@EJB(beanName="InstantHello")
  @EJB(beanName="FunnyHello")
  private HelloInterface bean;
  @Override
  public void start() {
    // Call the server
    System.out.println(bean.sayHello());
    // Call the server
    System.out.println(bean.sayHello("Anna"));
  }
}
```

Exercice 3 (*Une deuxième implantation*)

Dans cet exercice vous allez ajouter la deuxième implantation du classique *Hello World* présentée dans l'exercice 1.

▷ **Question 3.1 :**
Faites les modifications nécessaires à votre projet Eclipse pour qu'il respecte l'architecture que vous avez identifiée dans l'exercice 1. Testez votre application. Que

constatez vous ?

Il faut ajouter un autre composant qui implante la même interface que pour le composant précédent. Ce qu'on constate c'est que le projet Eclipse avec l'application ne peut plus être déployé sur le conteneur EJB. C'est un problème de résolution de référence. Le problème vient du fait que lors de l'injection de dépendances `@EJB` nous ne donnons pas le nom de l'implantation à utiliser. Comme maintenant au sein du conteneur EJB il y a deux implantations de la même interface, il ne sait pas laquelle utiliser. Il faut donc le lui préciser ...
Pour ce faire il faut ajouter un paramètre à l'annotation : `@EJB(beanName="")` et entre les guillemets il faut mettre le nom de la classe d'implantation.
Attention, préalablement un autre problème doit être résolu qui lui est propre à Glassfish. Il faut ajouter dans le fichier de description de déploiement `glassfish-ejb-jar.xml` la ligne `<disable-nonportable-jndi-names>true</disable-nonportable-jndi-names>` juste avant la balise de fin du fichier. Ceci évite que Glassfish ne génère des noms JNDI non portables pour les EJB déployés.
++++++++++++++++++ Implantation Serveur ++++++++++++++++++

```
package helloServices;
import javax.ejb.*;
/**
 * Session Bean implementation class FunnyHello
 */
@Stateless
public class FunnyHello implements HelloInterface {
  @EJB
  RandomNameInterface randomName;
  @Override
  public String sayHello() {
    String toWhom = randomName.generateName();
    return "Hello " + toWhom + "!!";
  }
  @Override
  public String sayHello(String name) {
    String toWhom = randomName.generateName();
    return "Hello " + toWhom + "!!";
  }
}

package helloServices;
import java.util.Random;
import javax.ejb.Stateless;
@Stateless
public class RandomName implements RandomNameInterface {
  @Override
  public String generateName() {
    String[] array = { "Julie", "Marie", "Lisa", "Cécile", "Virginie",
        "Dominique", "Ambre", "Anna", "Elena" };
    Random ran = new Random();
    return array[ran.nextInt(array.length)] + " "
        + array[ran.nextInt(array.length)];
  }
```

```
}

package helloServices;
import javax.ejb.Local;
@Local
public interface RandomNameInterface {
  String generateName();
}
```

++++++++++++++++++ Implantation Client ++++++++++++++++++

```
package helloClient;
import helloServices.*;
import javax.ejb.*;
/**
 * Session Bean implementation class HelloClient
 */
@Stateless
public class HelloClient implements IHelloClient {
  //@EJB(beanName="InstantHello")
  @EJB(beanName="FunnyHello")
  private HelloInterface bean;
  @Override
  public void start() {
    // Call the server
    System.out.println(bean.sayHello());
    // Call the server
    System.out.println(bean.sayHello("Anna"));
  }
}
```

▷ **Question 3.2 :**
Que faut-il faire pour modifier l'implantation utilisée par le client ?

Il faut modifier le code du client, vu qu'on utilise des annotations... du coup ce qu'on gagne avec l'injection de dépendances reste limité ... le client n'est pas indépendant de l'implantation de l'interface. On aurait pu mettre les choses non pas dans l'annotation mais dans un fichier XML. Dans ce cas, on ne modifie pas le code du client mais le fichier ...

Exercice 4 (*Les EJB* Stateful)

Dans cet exercice nous allons ajouter la gestion d'un état correspondant à l'historique des appels au message de bienvenue de type *Hello* suivi du nom de la personne. L'historique sera conservé en mémoire et, à chaque requête, il sera affiché.

▷ **Question 4.1 :**
Modifiez votre composant serveur pour ajouter la gestion de cet état. Exécutez votre application une fois, puis une deuxième fois. Que constatez vous ?

À chaque exécution, l'historique de noms est incrémenté d'un nom : l'instance du composant serveur est donc la même indépendamment du client qui l'utilise. Ceci est vrai aussi si on crée une autre classe cliente qui appelle le serveur un nom différent en paramètre.
Dans cette correction l'annotation `@Stateless` a des attributs qui sont nécessaires pour distinguer l'implantation `InstantHello` avec la précédente.
++++++++++++++++++ Implantation Serveur ++++++++++++++++++

```
package helloServices.state;
import java.util.*;
import helloServices.HelloInterface;
import javax.ejb.Stateless;
/**
 * Session Bean implementation class InstantHello
 */
@Stateless(name="InstantHelloState")
public class InstantHello implements HelloInterface {
  private List<String> nameHistory;
  public InstantHello() {
      nameHistory = new ArrayList<String>();
  }
  @Override
  public String sayHello() {
    return "Hello guys!!";
  }
  @Override
  public String sayHello(String name) {
    nameHistory.add(name);
    System.out.println("Call history " + nameHistory);
    return "Hello " + name + "!!";
  }
}
```

++++++++++++++++++ Implantation Client 1 ++++++++++++++++++

```
package helloClient.state;
import helloClient.IHelloClient;
import helloServices.*;
import javax.ejb.*;
/**
 * Session Bean implementation class HelloClient
 */
@Stateless(name="HelloClientState", mappedName="HelloClientState")
public class HelloClient implements IHelloClient {
  @EJB(beanName="InstantHelloState")
  private HelloInterface bean;
  @Override
  public void start() {
    // Call the server
    System.out.println(bean.sayHello("Lucie"));
  }
}
```

++++++++++++++++++ Implantation Client 2 ++++++++++++++++++

```
package helloClient.state;
import helloClient.IHelloClient;
import helloServices.HelloInterface;
import javax.ejb.*;
/**
 * Session Bean implementation class AnotherHelloClient
 */
@Stateless
public class AnotherHelloClient implements IHelloClient {
  @EJB(beanName="InstantHelloState")
  private HelloInterface bean;
    public AnotherHelloClient() {
    }
  @Override
  public void start() {
    // Call the server
    System.out.println(bean.sayHello("Yann"));
  }
}
```

On souhaite maintenant avoir un historique différent en fonction du client qui appelle. Dans le cadre des EJBs, lorsque l'on veut « isoler » les traitements faits par différentes instances d'un composant, il faut utiliser des composants de type *Session Stateful*. Nous allons utiliser maintenant ce type de composants pour réaliser la gestion de l'état.

▷ **Question 4.2 :**
Modifiez votre architecture pour inclure la nouvelle implantation.

Il suffit d'ajouter un nouveau composant qui réalise la même interface que jusqu'à présent.

▷ **Question 4.3 :**
Modifiez votre projet Eclipse pour réaliser la nouvelle architecture.

Attention pour bien se rendre compte de la différence entre le *Stateless* et le *Stateful* il faut créer un nouveau composant client qui appelle le serveur en utilisant un autre nom, sinon on ne fera la pas la différence.
Avec le type *Stateful* pour un client on a toujours la même instance de composant qui est utilisée, les noms s'ajoutent à l'historique. Pour l'autre client ce n'est pas la même instance, donc les mêmes noms qui sont dans l'historique.
++++++++++++++++++ Implantation Serveur ++++++++++++++++++

```
package helloServices.stateful;
import helloServices.HelloInterface;
import helloServices.interceptors.LoggingInstanceCreation;
import helloServices.interceptors.RecordingCalls;
import java.util.*;
import javax.ejb.Remove;
import javax.ejb.Stateful;
import javax.interceptor.Interceptors;
```

```java
/**
 * Session Bean implementation class InstantHelloStateful
 */
@Stateful
@Interceptors({RecordingCalls.class,LoggingInstanceCreation.class})
public class InstantHelloStateful implements HelloInterface {
  private List<String> nameHistory;
    public InstantHelloStateful() {
      nameHistory = new ArrayList<String>();
    }
  @Override
  @Remove
  public String sayHello() {
    return "Hello guys!!";
  }
  @Override
  public String sayHello(String name) {
    nameHistory.add(name);
    System.out.println("Call history " + nameHistory);
    return "Hello " + name + "!!";
  }
}
```

++++++++++++++++++ Implantation Client ++++++++++++++++++

```java
package helloClient.stateful;
import helloClient.IHelloClient;
import helloServices.*;
import javax.ejb.*;
/**
 * Session Bean implementation class HelloClient
 */
@Stateless(name="HelloClientStateful", mappedName="HelloClientStateful")
public class HelloClient implements IHelloClient {
  @EJB(beanName="InstantHelloStateful")
  private HelloInterface bean;
  @Override
  public void start() {
    // Call the server
    System.out.println(bean.sayHello("Lucie"));
  }
}
```

++++++++++++++++++ Implantation Client ++++++++++++++++++

```java
package helloClient.stateful;
import helloClient.IHelloClient;
import helloServices.HelloInterface;
import javax.ejb.*;
/**
 * Session Bean implementation class AnotherHelloClient
 */
@Stateless(name="AnotherHelloClientStateful", mappedName="AnotherHelloClientStateful")
```

```
public class AnotherHelloClient implements IHelloClient {
  @EJB(beanName="InstantHelloStateful")
  private HelloInterface bean;
    public AnotherHelloClient() {
    }
  @Override
  public void start() {
    // Call the server
    System.out.println(bean.sayHello("Yann"));
  }
}
```

Exercice 5 (*Des services d'un conteneur EJB*)

Pour réaliser cet exercice vous serez amenés à consulter des extraits du chapitre 18 du livre *« Entreprise JavaBeans 3.1 »* disponible sur *Google Books*.

▷ **Question 5.1 :**
Nous souhaitons généraliser le stockage de l'historique des appels à toutes les implantations de notre serveur *Hello World*. Quelle démarche appliqueriez vous pour le faire ?

La plus simple est de copier le code qui savegarde les appels dans toutes les implantations... ce qui est « lourd ».

Un des services offerts par un conteneur EJB c'est la possibilité d'ajouter à vos EJBs de type session (et message) des traitements définis dans une classe Java. La classe qui contient ces traitements est appelée un *intercepteur*. Dans les spécifications EJB il est possible d'ajouter un ou plusieurs intercepteurs lors de l'invocation de méthodes de votre EJB mais aussi lors d'événements liés au cycle de vie de l'EJB.

Le modèle de programmation d'un intercepteur de base est assez simple : une classe Java classique qui contient une méthode avec l'annotation `@AroundInvoke`. La signature de la méthode est celle donnée page 325 du livre cité en début de l'exercice.

▷ **Question 5.2 :**
Programmez votre premier intercepteur EJB qui sera en charge d'ajouter aux implantations du *Hello World* la gestion de l'historique.

```
package helloServices.interceptors;
import java.util.*;
import javax.interceptor.*;
public class RecordingCalls {
  private final List<String> nameHistory = new ArrayList<String>();
  @AroundInvoke
  public Object recordInvocation(InvocationContext ic) throws Exception {
    String methodName = ic.getMethod().getName();
    Object[] params = ic.getParameters();
    if (methodName.equals("sayHello") && (params != null)) {
```

```
      nameHistory.add((String)params[0]);
    }
    System.out.println("Call history " + nameHistory);
    return ic.proceed();
  }
}
```

▷ **Question 5.3 :**
Modifiez vos implantations du composant serveur pour utiliser l'intercepteur que vous venez de programmer.

```
package helloServices.withInterceptors;
import helloServices.HelloInterface;
import helloServices.interceptors.*;
import javax.ejb.Stateless;
import javax.interceptor.Interceptors;
/**
 * Session Bean implementation class InstantHello
 */
@Stateless(name="InstantHelloIntercepted")
@Interceptors({RecordingCalls.class,LoggingInstanceCreation.class})
public class InstantHello implements HelloInterface {
  public InstantHello() {
  }
  @Override
  public String sayHello() {
    return "Hello guys!!";
  }
  @Override
  public String sayHello(String name) {
    return "Hello " + name + "!!";
  }
}
```

Comme déjà mentionné précédemment, un intercepteur peut être exécuté lorsque certains événements liés au cycle de vie de d'EJB surviennent. Lisez la documentation ici `docs.oracle.com/javaee/6/tutorial/doc/giplj.html` et ici `www.jmdoudoux.fr/java/dej/chap-ejb3.htm#ejb3-10` pour savoir quel est le cycle de vie d'un EJB et connaître les annotations qui y sont associées.

▷ **Question 5.4 :**
Utilisez les pour voir le nombre d'instances créées de votre composant serveur, aussi bien dans le cas *stateless* que *stateful*.

```
package helloServices.interceptors;
import javax.annotation.*;
import javax.ejb.PrePassivate;
import javax.interceptor.InvocationContext;
public class LoggingInstanceCreation {
  @PostConstruct
```

```
  public void logCreation(InvocationContext ic) {
    try {
      ic.proceed();
    } catch (Exception e) {
      e.printStackTrace();
    } finally {
      System.out.println("EJB post contruction" + ic.getTarget().getClass().getName());
    }
  }
  @PreDestroy
  public void logDestruction(InvocationContext ic) {
    System.out.println("EJB predestroy " + ic.getTarget().getClass().getName());
    try {
      ic.proceed();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
  @PrePassivate
  public void logPassivate(InvocationContext ic){
    System.out.println("EJB prepassivate " + ic.getTarget().getClass().getName());
    try {
      ic.proceed();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```