Compléments POO : Programmation orientée composants

Stéphane Frénot

INSA Lyon

Programmation Objet

```
package contacts;
public class Contact {
  protected String nom;
  public Contact(String nom){this.nom=nom;}
  public getNom(){return this.nom;}
  ...
}

public class client {
  public static void main(String [] arg){
    Contact unClient=new Client("Pierre", "Durand");
    unClient.getNom();
  }
}
```

==> Si la classe n'est pas bonne, il faut changer la classe et le client

Programmation par Interface

```
public interface Contact {
  public String getNom();
  ...
}

public interface ContactFinder {
  Contact [] findByNom(String ln);
  ...
}
```

==> L'interface permet d'extraire l'information significative et rend le client indépendant du service

Code client

```
public class ListByNom{
  public static void main(String [] args){
    ContactFinder cf=new ContactFinderImpl();
    Contact[] ct=cf.findByLastName(args[0]);
    for (int n=0; n<ct.length; n++){
      System.out.println(cts[n]);
    }
  }
}
```

==> Si l'implantation n'est bonne seule une seule ligne du code client est changée

Extension du service

```
package contacts.entreprise;
public interface ContactEntreprise extends Contact {
  public int getEntrepriseSize();
}
```

==> On peut facilement étendre le service par extension de l'interface, et définition d'une classe d'implantation. Le client existant reste inchangé.

!!! Mais ce n'est pas de la programmation par composants

Approche par composants

Composant : C'est une unité indépendante de production et de déploiement qui est combinée à d'autres composants pour former une application. L'objet reste au centre du composant mais l'objectif diffère

- Programmation Objet
  - Conception et développement
  - Relations à l'exécution entre les entités
- Programmation par Composants : pas uniquement exécution
  - Deploiement
  - Production

Les questions sont différentes

Approche Orientée Objets :

- Est ce que la conception a bien capturé la problématique du domaine ?
- Est ce que les classes et interfaces concues sont facilement adaptables ?

Approche Orientée Composants

- Comment un client trouve sa classe d'implantation à l'exécution ?
- Que faire s'il existe plusieurs versions d'implantation ?
- Comment un composant trouve et charge sa configuration ?
- Que se passe t'il si un processus ou un conteneur doit être arrété ?
- Comment les composants sont ils reliés en terme de mise à jour ?
- Est ce que les composants ne contiennent pas du code inutile au moment du développement ?
- Comment perfectionner un composant éprouvé sans toucher à son code ?
- Que se passe t'il si une partie du système est implanté sur une plateforme différente ?

Quelques réponses des machines virtuelles

- Le classloader
  permet de piloter finement le chargement d'une classe
- La reflexion de package
  permet de connaitre des meta-attributs d'un composant
- Attributs du bytecode de classe
  pour l'association entre client et composant
- La reflexion de classe
  permet de connaitre dynamiquement le contenu d'une classe

INRIA

Quelques réponses des machines virtuelles

- Génération automatique de code
- Proxys statiques
- Java Native Interface

INRIA

Programmation Composant : Usine

```
public class ContactFinderFactory{
  public ContactFinder getDefaultFinder(){
    try{
      String classname =
       System.getProperty("contacts.FinderClass",
                  "contacts.impl.SimpleContactFinder");
      Class clazz=
       Class.forName(className, true,
                Thread.currentThread().getContextClassLoader());
      return (ContactFinder)clazz.newInstance();
    }catch(Exception e){e.printStackTrace();}
  }
}
```

Programmation Composant

```
public class SimpleContactFinder implements ContactFinder {
  static {
    try {
      InputStream is=
SimpleContactFinder.class.getClassLoader().\\
       getResourceAsStream("contacts/impl/config.properties");
      Properties props=(new Properties()).load(is);
    }catch(Exception e){
     throw new Error ("Impossible de charger les propriétés \\
      : contacts.impl.config.properties");
    }
  }
}
```

Programmation Composant : service

```
public class SimpleContact implements Contact, Serializable {
  public readObject(ObjectInputStream ois)
   throws IOException, ClassNotFoundException, ContactException{
    ois.defaultReadObject();
    this.validateNewInstance();
  }
  private void validateNewInstance() throws ContactsException {
  }
}

;contacts.jar ==> Composant
Sealed=true
Implementation-Title=SimpleContacts
Implementation-Version=1.0.0
Implementation-Vendor=SFR
Specification-Title=Contacts
Specification-Version=1.0.0
Specification-Vendor=SFR
```

Le ClassLoader 1

Les applications changent et ne sont plus statiquement disponibles en un seul execuable. La tendance est de décomposer les applications en composants qui :

- Seront partagés par différentes applications
- Pour certains seront mis à jour, sans que l'ensemble de l'application soit modifiée
- Sont dynamiques et sont récupérés en fonction de l'environnement d'exécution
- Reposent sur des resources (BD, images...) qui ne sont pas à la charge du programmeur
- Sont entièrement et uniquement définis à la phase de déploiement

Le ClassLoader 2

Pouvoir assembler une application à l'exécution soulève quelques problèmes. Une platforme de services doit permettre de :

- Trouver les composants de l'application
- Vérifier que le composant est compatible avec l'application
- Choisir parmis plusieurs version possibles
- Gérer les pannes de mise à disposition

L'architecture du class loader doit être :

- Transparent
- Extensible
- Competent
- Configurable
- Gérer les conflits de nom et de version
- Sécurisé

Chargement de classe Implicite

```
Toto t;
t=new Toto(); /* Le classLoader est lancé */
```

Chargement de classe Explicite

```
public class ChargesMoi{
  static {
    System.out.println("Je suis chargé");
  }
}

public class Chargeur {
  public static void main(String [] arg){
    URL url = new URL ("file:sousrepertoire/");
    URL [] urls=new URL[]{url};
    URLClassLoader loader=new URLClassLoader(urls);
    Class cls=Class.forName("LoadMe", true, loader);
    cls.newInstance();
  }
}
```

Chargement de ressources

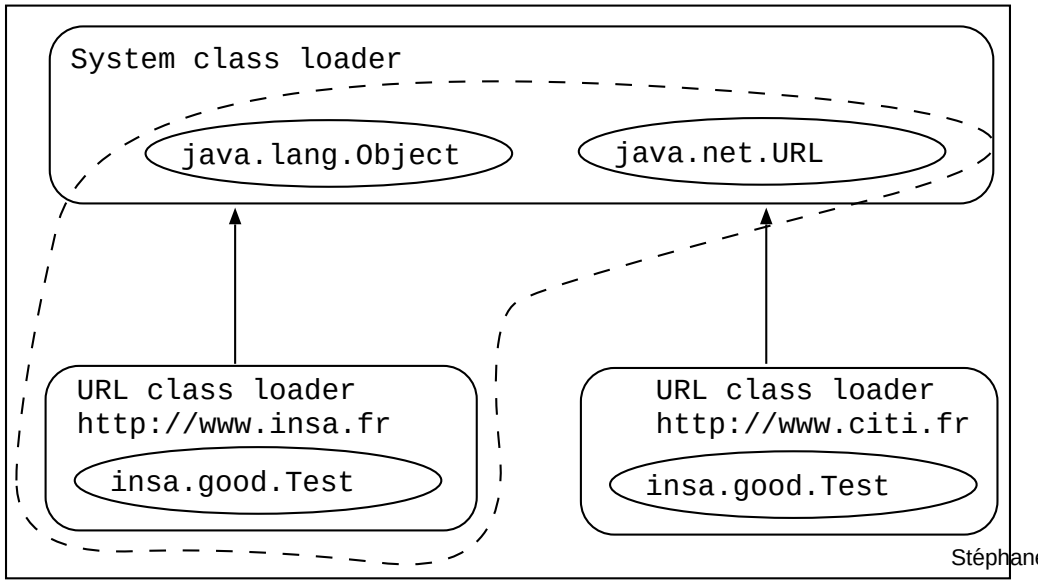
Le classloader permet également de charger des ressources

```
public class ChargesResources{
  public static void main(String [] arg){
    ClassLoader cl= LoadResources.class.getClassLoader();
    InputStream is=cl.getResourceAsStream("config.props");
    Properties props=new Properties();
    props.load(is);
    props.list(System.out);
  }
}
```

==> Si le classloader sait charger la classe il sait charger la resource

Les règles du classLoader

- Règle de consistance : pas deux fois la même classe
- Règle de délégation : le classLoader demande au CL qui l'a chargé
- Règle de visibilité : toutes les classes ne sont pas connues



Serveur

```
public interface Point {
  public void move (int dx, int dy);
}

public class PointImpl {
  public void move (int dx, int dy){
    this.x+=dx; //erreur
  }
}
```

==> Le service est classique

Loader

```
public class ServeurDePoint{
  static ClassLoader c1;
  static Class ptClass;
  public static Point createPoint{Point tmp){
    Point newPt=(Point)ptClass.newInstance();
    if (tmp!=null){
      newPt.move(tmp.getX(), tmp.getY();
    }
    return newPt;
 }

 public synchronized void reloadImpl() throws Exception{
   URL[] serverURLS=new URL[]{new URL("file:subdir/");
   cl=new URLClassLoader(serverURLS);
   ptClass=cl.classLoader("PointImpl");
 }
}
```

==> Le loader permet de charger dynamiquement une nouvelle version

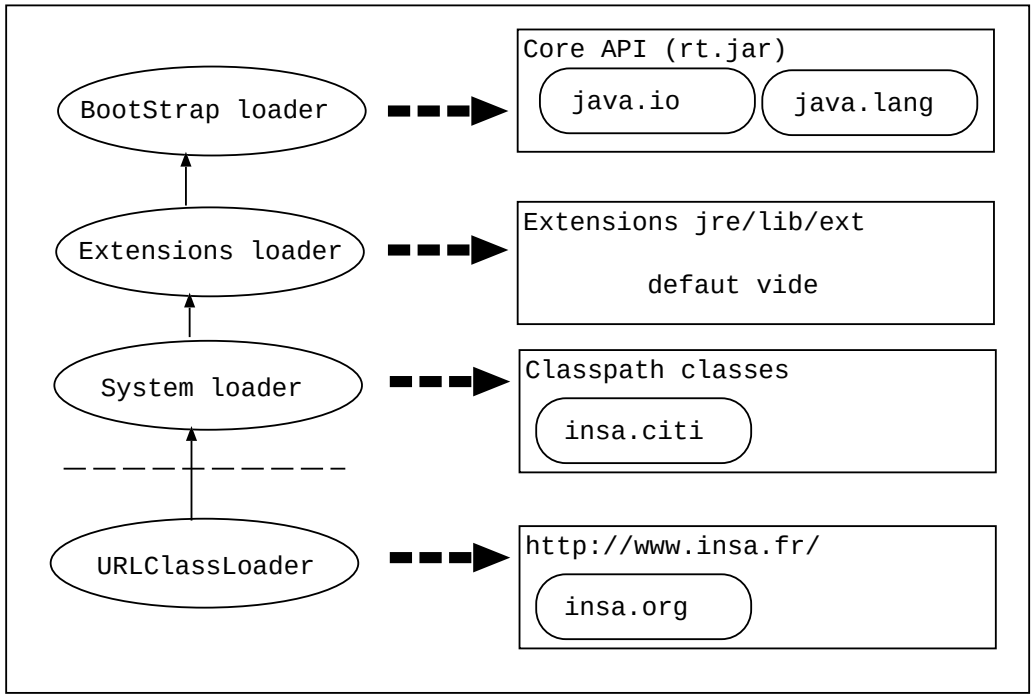
Client

```
public class Client{
  static Point pt;
  public static void main(String [] arg){
    pt=PointServer.createPoint(null);
    while (true){
      ...
      pt.move(1,1); // pt=1,0
      ...
      PointServer.reloadImpl();
      pt=PointServer.createPoint(pt);
      pt.move(1,1); // pt=1,1
    }
  }
}
```

Contraintes d'écriture du Client

- Ne pas référence l'implantation
- Le client peut référencer l'impl ou une parente
- Impl et Interf doivent être sur la même version
- Il faut pouvoire transmettre l'ancien état (copie constructeur)
- Le client doit fournir la nouvelle et détruire l'ancienne référence

Les classes loaders standards



==> Securité !

- java -Djava.ext.dirs=myext MaClasse
- grant codeBase "file:${java.home}/lib/ext/*"{permission java.security.AllPermission;};

Instrumentation du chargement

- Instrumentation de l'application
- Lancement en mode verbose (java -verbose:class)
- Instrumentation de la core API (rt.jar)

```
public class Gurp{
  public void useFoo{ Foo f=new Foo();} // Ca plante
  static {
    ClassLoader cl= Gurp.class.getClassLoader();
    while(cl!=null){
      System.out.println("==>"+cl);
      cl=cl.getParent();
    }
  }
}
==> sun.misc.Laucher$AppClassLoader@242484
==> sun.misc.Laucher$AppClassLoader@66d877
{bootStrap loader}
```

La reflexion

La reflexion Java repose sur plusieurs mécanismes

- La description du bytecode
- L'API de reflexion
- Les proxy dynamiques

INRIA

Le Byte Code Java

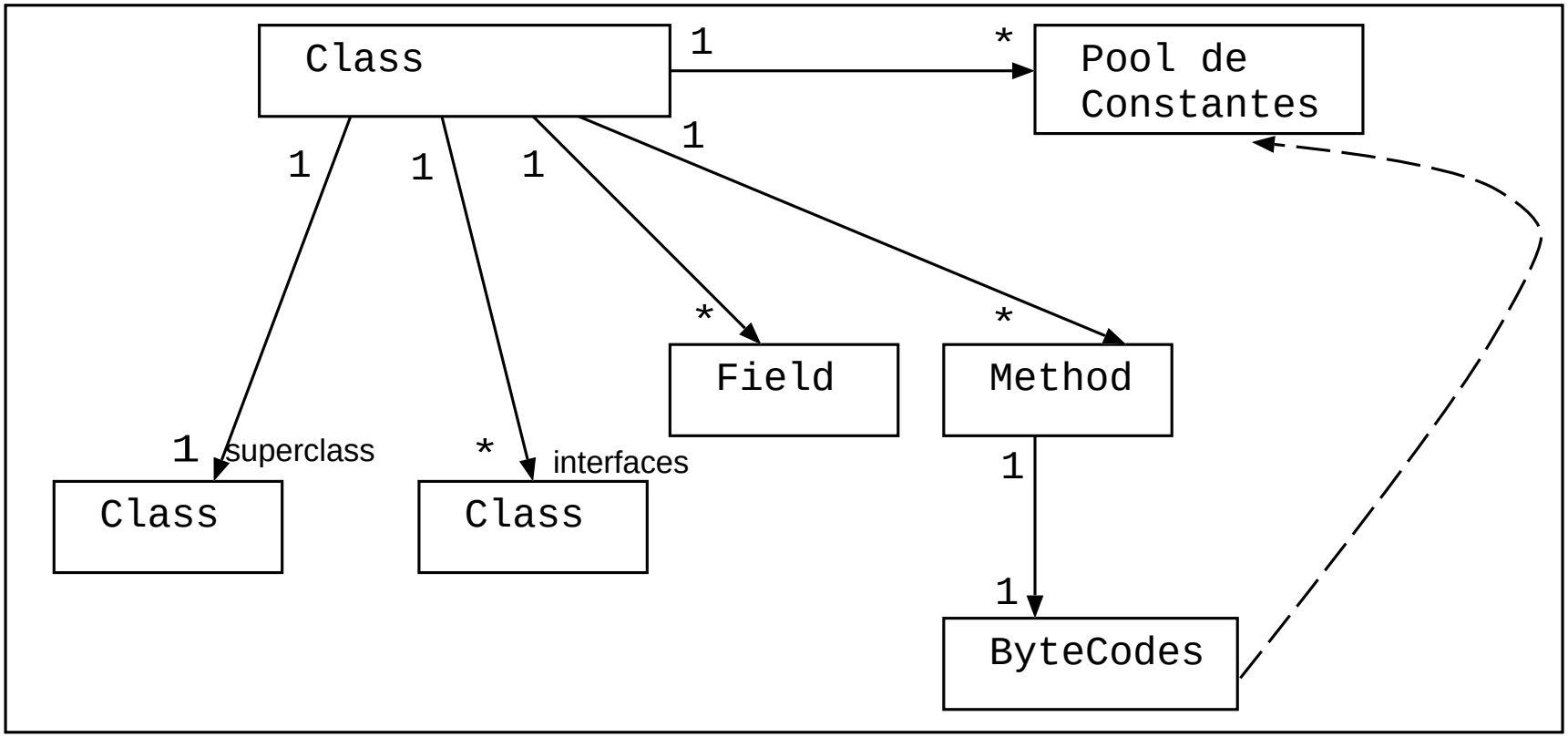
```
public class Test{
  public void test1 () { System.out.println("Coucou 1");}
  public void test2 () { System.out.println("Coucou 2");}
}

public class ClientTest {
  public static void main(String [] arg){
    Test t=new Test();
    t.test1();
}
```

Si Test est modifiée que ce passe t'il pour ClientTest ?

- en C++ L'invocation d'une méthode est résolu par un offset mémoire
- en Java Le byte code maintient des meta data, qui indiquent que la classe serveur n'est plus compatible

Schéma de classes



- javap Test ==> Renvoie les structures de la classe
- javap -c ==> Présente le bytecode de la classe

Décompilation exemple

```
Compiled from Test.java
public class Test extends java.lang.Object {
    int i;
    Test t;
    public Test(){}
    public static void main(java.lang.String[]){}
}
```

INRIA

Décompilation exemple

```
Method Test()
   0 aload_0
   1 invokespecial #1 <Method java.lang.Object()>
   4 aload_0
   5 iconst_0
   6 putfield #2 <Field int i>
   9 aload_0
  10 new #3 <Class Test>
  13 dup
  14 invokespecial #4 <Method Test()>
  17 putfield #5 <Field Test t>
  20 getstatic #6 <Field java.io.PrintStream out>
  23 ldc #7 <String "Je suis créé">
  25 invokevirtual #8 <Method void println(java.lang.String)>
  28 return
```

Décompilation exemple

```
Method void main(java.lang.String[])
   0 new #3 <Class Test>
   3 dup
   4 invokespecial #4 <Method Test()>
   7 astore_1
   8 return
```

Reflexion

L'API reflexion permet de manipuler dynamiquement un objet (sans connaitre a priori son implantation)

Exemple 1: Exécution dynamique de classe

```
public class RunFromURL{
 public static void run(String url, String className, String [] args
  URLClassLoader ucl=new URLClassLoader(new URL[]{new URL(url)});
  Class cls=ucl.loadClass(clsName);
  Class argClass=String[].class;
  Method mainMeth = cls.getMethod("main", new Class[]{argClass});
  mainMeth.invoke(null, new Object[]{args}); }
 public static void main(String [] args){
  int argCount=args.length-2;
  String[] newArgs=new String[args.length-2];
  for (int i=0;i<argCount;i++){newArgs[i]=args[i+2];}
  run(args[0], args[1], newArgs); } }

RunFromUrl insa.org.Test
```

Reflexion : Exemple

Exemple 2 : Chargement dynamique de drivers
Dans le code jdbc (Connexion à une base de données), le driver spécifique à telle ou telle base n'est connu qu'à l'exécution.

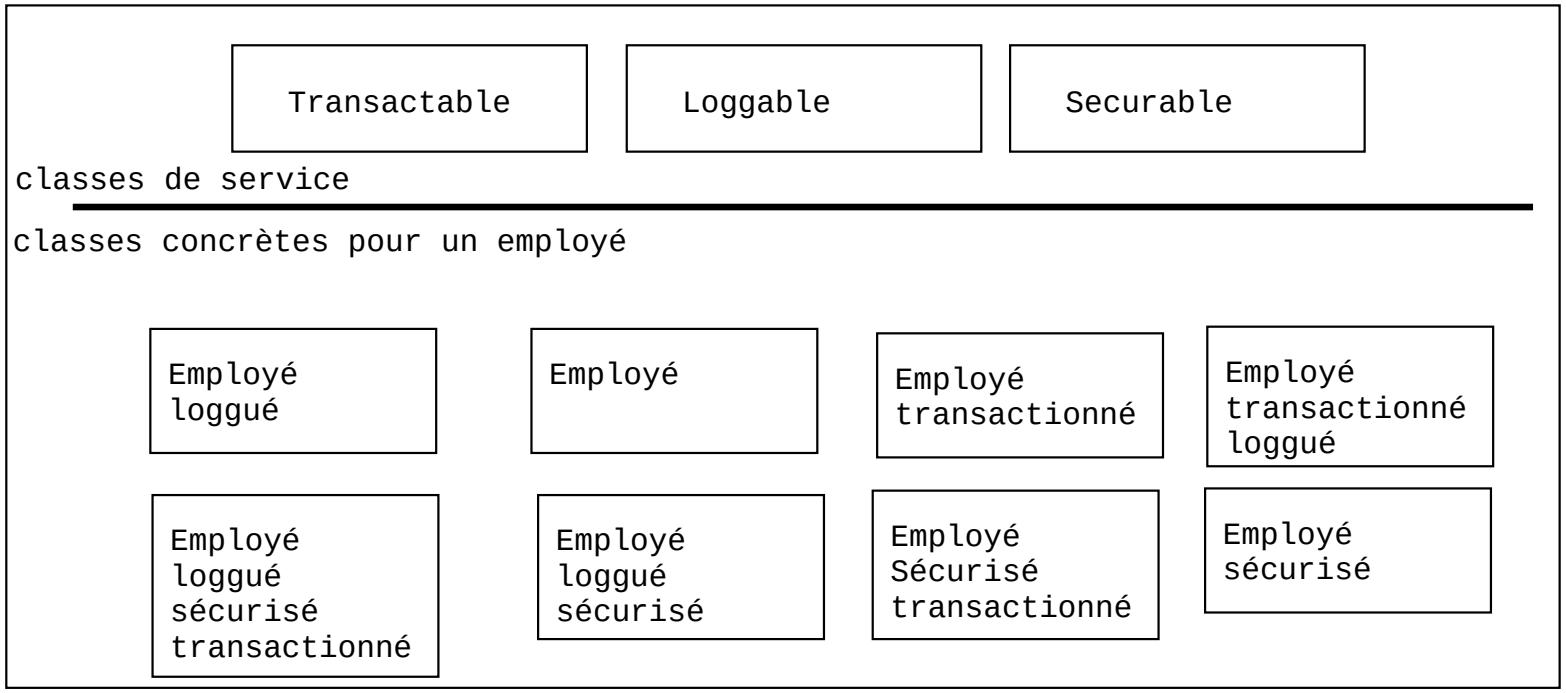
```
public class Connect {
 public Connect(String driverName, URL url){
  Class.forName(drivername).newInstance();
  Connection c=DriverManager.getConnection(url);
 }
}
```

==> Mécanisme général de chargement d'un service conforme à une API (Nommage, JDBC, JMS...)

Les proxys dynamiques

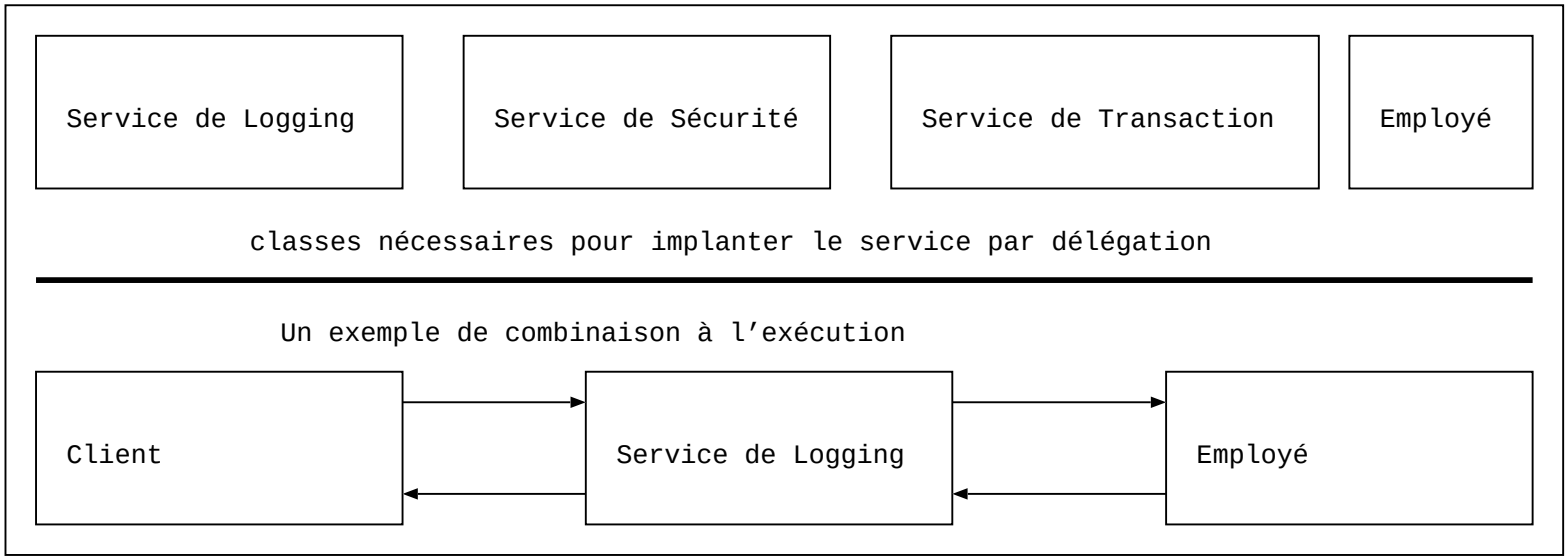
- **Reflexion** : Un client utilise une API générique pour appeler des méthodes sur une classe serveur inconnue à la compilation.
- **dynamic proxies** : Un serveur utilise une API générique pour implanter une méthode qui est fournie à l'exécution pour satisfaire le besoin du client.
  - Mécanisme de base pour les intercepteurs génériques. (conteneurs J2EE)
  - Modèle de délégation lorsque l'héritage explose

Implantation par héritage



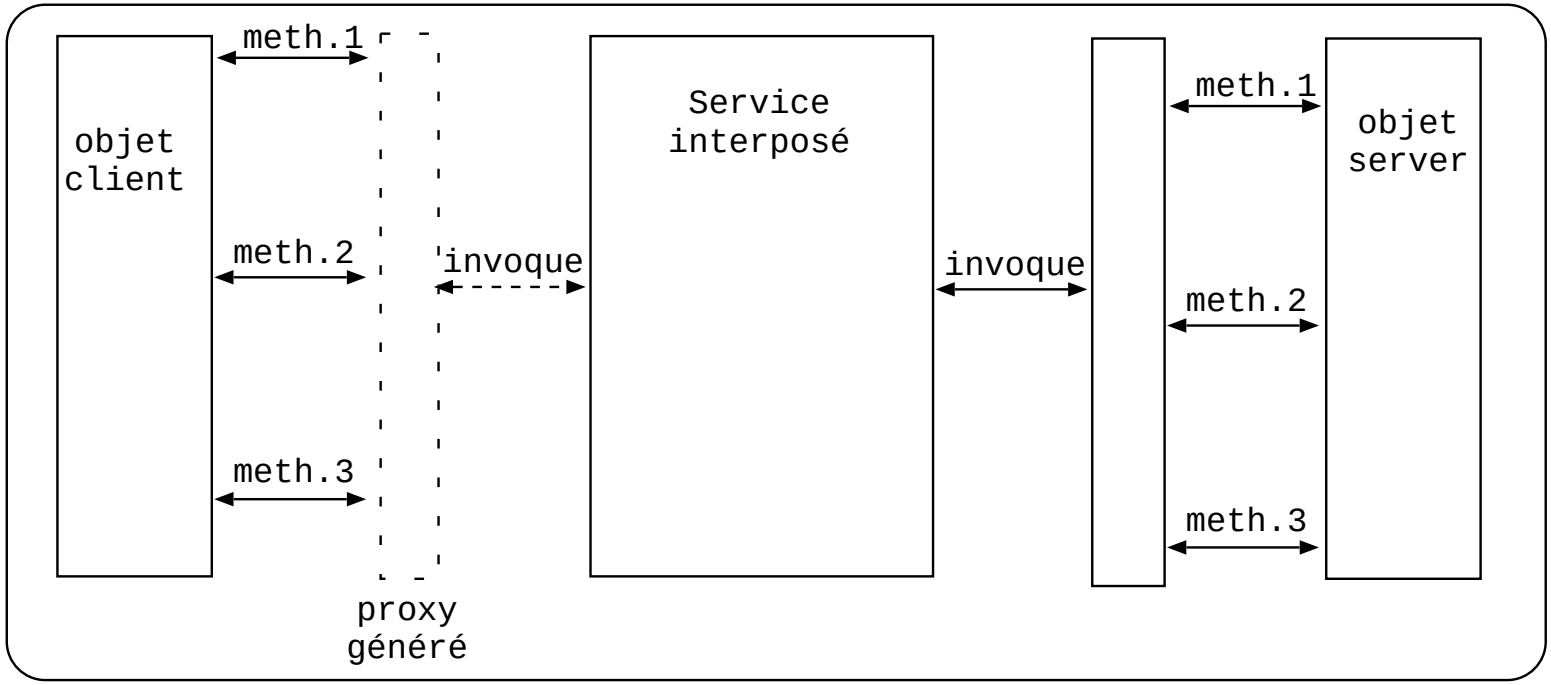
- Explosion de classes
- Evolutivité, nouvelle interface –> X nouvelles classes
- Souplesse, quelles sont les classes concrètes significatives

Implantation par Délégation



- Génération statique de code client (stub) (ex : jonas)
- Proxy dynamique (ex : jboss)

La génération dynamique



Exemple d'interposition : le client

```
PersonneItf p=UsineDePersonne.create("Dupond");
p.getNom();
```

L'usine fabrique normalement un client, mais elle peut interposer un service.

Exemple d'interposition : l'usine

```
public static create(String name){
  PersonneItf personne=new PersonneImpl(name);
  PersonneItf proxy=(PersonneItf)Proxy.newProxyInstance(
    UsineDePersonne.class.getClassLoader(),
    new Class[]{Personne.class},
    new Interposition(personne));
  return (proxy);
}
```

L'usine fabrique un proxy dynamique, conforme à l'interface du client

Exemple d'interposition : le service

```
import java.lang.reflect.*;
public class Interposition implements InvocationHandler{
  private Object delegate;
  public Interposition (Object delegate){ this.delegate=delegate; }
  public Object invoke(Object source, Method method, Object[] args)
    throws Throwable {
    /* Interposition avant */
    Object result=null;
    try{
      result=method.invoke(delegate,args); /* getNom est invoqué */
    }catch(InvocationTargetException e){ throw
      e.getTargetException(); }
    /* Interposition après */
    if (result.equals("Dupond")){
      result="Frenot"; }
    return result;
  }
}
```

L'interposition agit !

Les avantages des Proxys dynamiques

N’apportent rien par rapport à coder la délégation à la main

1. Ils sont dynamiques et génériques
2. Validation de paramètres
3. Sécurité
4. Propagation de contexte
5. Audit, trace, débuggage
6. Dérouter un appel java vers un autre environnement

==> Coût supérieur à un codage à la main

Performances puissance de 10ns

1. Incrément d'un champ entier : 10e0 (1ns)
2. Invocation d'une méthode virtuelle : 10e1
3. Invocation par délégation manuelle : 10e1-10e2
4. Invocation reflexive : 10e4-10e5
5. Invocation par un proxy dynamique : 10e4-10e5
6. Incrémentation reflexive d'un entier : 10e4-10e5
7. Appel RMI sur une machine locale : 10e5
8. Ouverture de fichiers : 10e6
9. Vitesse de la lumière parcours 3000km : 10e7

Biblio

1. Stuart Dabbs Halloway – Component Development for the Java Platform. Addison-Wesley
2. Guy Bieber, Jeff Carpenter – Introduction to Service-Oriented Programming (Rev 2.1) – http://www.openwings.org