Université Nice SOPHIA ANTIPOLIS

Université Nice SOPHIA ANTIPOLIS

Chargement dynamique de classes

Philippe Collet

A partir des éléments de cours de Michel Buffa

Master 1 IFI

2012-2013

http://deptinfo.unice.fr/twiki/bin/view/Minfo/GenieLog1213

Plan

- Principe du ClassLoader
- Ecrire un ClassLoader *custom*
- Architecture pour charger des plugins
- Comment recharger des plugins
- Application au simulateur de bébêtes

Définition d 'un ClassLoader

- La JVM contient un ClassLoader
- Les Class Loaders permettent de charger des classes depuis le filesystem, mais aussi depuis de multiples endroits (BD, réseau, etc…)
- Rôle : convertir un nom de classe en tableau d 'octets représentant la classe

```
Class c = loadClass(String nomClasse, boolean
                    resolveIt);
```

Généralités sur les ClassLoaders

- Toutes les JVM ont un ClassLoader (il peut charger certaines classes sans vérifications, JDK…)
- Le CL par défaut implémente une méthode loadClass() qui cherche dans le CLASSPATH, les fichiers .jar et/ou .zip
- On peut créer de nouveaux CL en dérivant de la classe ClassLoader et en redéfinissant loadClass() et les méthodes qu'elle utilise (findClassFromClass, ResolveClass…)

Moment de chargement des classes

- Ca dépend !

- En général quand :
  - Toto t = new Toto();
  - Référence statique comme :
    - System.out,
    - Toto.class,
    - forName("Toto");

Ecrire son propre ClassLoader ?

- Intérêt ?
  - Charger des classes depuis le WWW,
  - Charger des classes depuis une BD,
  - Charger des classes *« différemment »*
- Ne Marche pas avec les applets !
- Depuis JDK1.2, un URLClassLoader est disponible

Ecrire un ClassLoader : les étapes

- Sous-classer ClassLoader et implémenter la méthode abstraite loadClass
  - 1) Vérifier le nom de la classe, voir si on ne l'a pas déjà chargée
  - 2) Vérifier s'il s'agit d'une classe « Système »
  - 3) Essayer de la charger
  - 4) Définir la classe pour la VM
  - 5) La résoudre (charger les dépendances)
  - 6) Renvoyer la classe à l'appelant
- JDK1.3 et plus: sous-classer SecureClassLoader si on veut respecter les recommandations relatives à la nouvelle politique de sécurité de Java

Exemple de ClassLoader *custom*

```
public synchronized Class loadClass(String className, boolean resolveIt) throws
    ClassNotFoundException {
    Class result;
    Byte []classData;

    // 1) Cherche dans le cache si la classe n'a pas déjà été chargée
    result = (Class) classes.get(className);
    if(result != null)
        return result;

    // 2) Utilisation du primordial CL pour voir s'il s'agit d'une classe
    // Systeme. Très important de faire ce test ! Sinon on pourrait remplacer
    // Le security manager !
    try {
        result = super.findSystemClass(className);
        return result;
    } catch(ClassNotFoundException e) {
        System.out.println("Pas une classe système !");
    }

    // 3) On charge la classe depuis NOTRE REPOSITORY (le www ou une BD par
    exemple). Méthode à nous ! Voir exemple page suivante
    classData = getImplFromDataBase(className);
    if(classData == null) { throw new ClassNotFOundException() }
```

Exemple de ClassLoader *custom* (suite)

```
...
    // 4) On "définit" la classe (on  la parse). En fait, la méthode suivante
    va // vérifier la validité du ByteCode + voir si tout est ok.
    // Stocke le tout dans la JVM dans une certaine structure de données.
    result = defineClass(classData, 0, classData.length);

    // 5) "Résoudre" la classe. C'est-à-dire faire le même traitement que celui
    qui est en train d'être fait, mais pour les superclasses et les classes
    dépendantes.
    if(resolveIt)
        resolveClass(result);

    // Avant de renvoyer le résultat final, on met la classe dans le cache
    classes.put(className, result);

    // 6) On renvoie le résultat !
    Return result;

}
```

Exemple de code qui lit une classe

```
…
byte [] result;
try {
  FIS fis = new FIS("store\\ " + className + ".impl");
   result = new byte[fis.available()];
   fis.read(result);
} catch(Exception e) {
   return null;
}
```

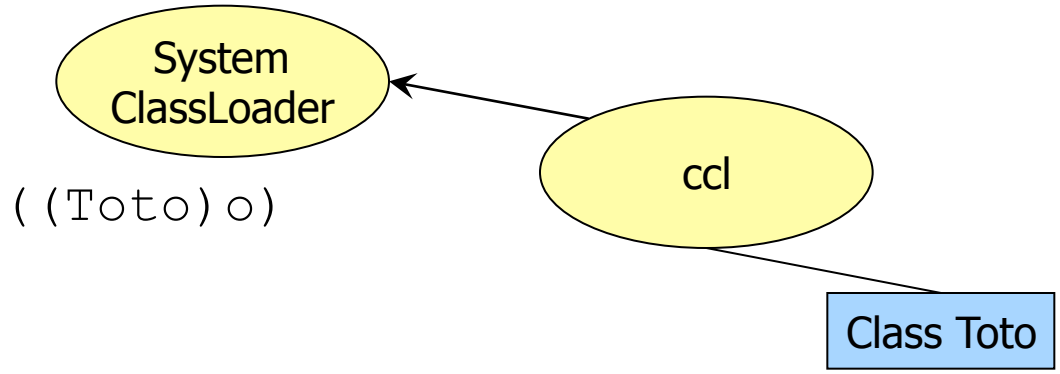
Exemple de code qui lit une classe depuis le WWW

```
…
URL url = new URL(urlClassName);
URLConnection uc = new URLConnection(url);
int length = uc.getContentLength();
IS is = uc.getInputStream();
byte []data = new byte[length];
is.read(data);
is.close();
return data;
...
```

Utilisation d'un ClassLoader *custom*

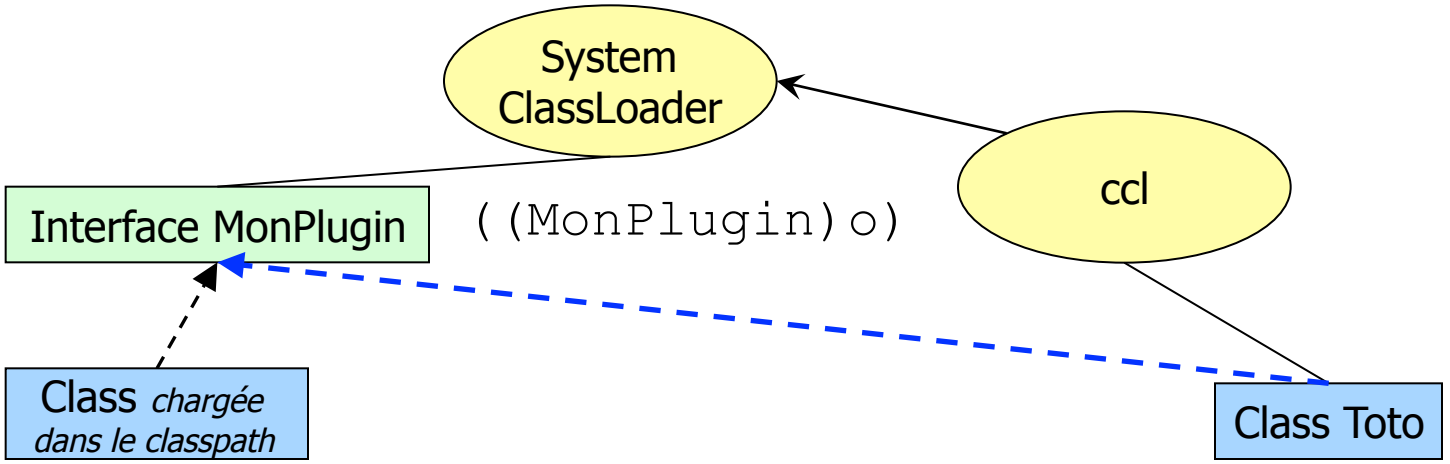
```
Class c = ccl.loadClass("Toto");
Object o = c.newInstance();
((Toto) o).f();
```

- ne marche pas car seul le nouveau cl connaît Toto !



Nécessité d'une Interface

- Seul le nouveau CL connaît Toto, si on veut effectuer un *cast*, il faut définir une Interface en commun, qui elle, est connue *aussi* du CL par défaut !
- Par exemple Plugin connue par le CL par défaut (SystemClassLoader par exemple), qui est implémentée par les classes chargées.



Ecrire des plugins

- Truc simple : lire des plugins dans un répertoire
- Définir une interface d'utilisation pour les plugins
- Chaque plugin devra implémenter cette interface
- Chaque classe *sera donc un Plugin*
- Il suffit ensuite de
  - 1) Lire le contenu du répertoire
  - 2) Pour chaque nom de classe présent dans le répertoire "plugins"
    - `Class c = Class.forName(nomClasse);`
    - `Plugin p = (PluginDessin) c.newInstance();`
    - `p.dessine(); // méthode présente dans Plugin.java`

Plugins suite...

- Difficultés lorsque
  - Pas de constructeur par défaut
  - Les plugins sont dans un .jar ou une BD
    - Chercher des ressources ? Images, sons, etc…
- Class.forName(nomCLasse) est l'équivalent du loadClass(nomCLasse) étudié dans le cas d'un ClassLoader custom !
- Les plugins, il y en a partout !!!
  - Photoshop, IE, Firefox, Chrome…

Plugins : modèle idéal

- Modèle idéal = un répertoire plugins,
- Des fichiers .jar dans le répertoire,
- L'application « découvre » les jars et « absorbe » les plugins qu'ils contiennent.

- Chaque jar contient
  - la classe qui implémente l'interface Plugin.java,
  - les autres classes dont elle a besoin,
  - Les images, icônes, sons, docs, etc… dont le plugin a besoin…

Plugin : modèle idéal

- Mais problème lorsqu'il y a de trop nombreuses classes dans le plugin… comment tester qu'une classe *est réellement* un plugin sans essayer de la charger, ce qui prend du temps…
  - Class.forName() est long…
  - Class.newInstance() aussi…

- Eclipse, par exemple, contient 800 plugins qui sont chargés au démarrage…
- Certains plugins étendent d'autres plugins et donc en dépendent,
- L'ordre de chargement est important…

Comment gérer ses plugins ?

- La solution passe souvent par un descripteur, une « carte » des plugins
  - Eclipse suit une norme pour décrire les plugins, leurs dépendances, etc… Il utilise un descripteur XML
- Le problème No 1 avec les plugins est le temps de chargement…
- Mais les avantages sont nombreux…

Développer un programme extensible par plugins

- Il faut fournir un SDK pour le développeur de plugins
  - Pour qu'il puisse compiler et tester ses plugins sans avoir le code de l'application principale,
  - Il faut fournir : Classes et Interfaces nécessaires,
  - Un ou plusieurs plugins d'exemples, avec le fichier ant correspondant,
  - De la doc, un tutorial, etc…

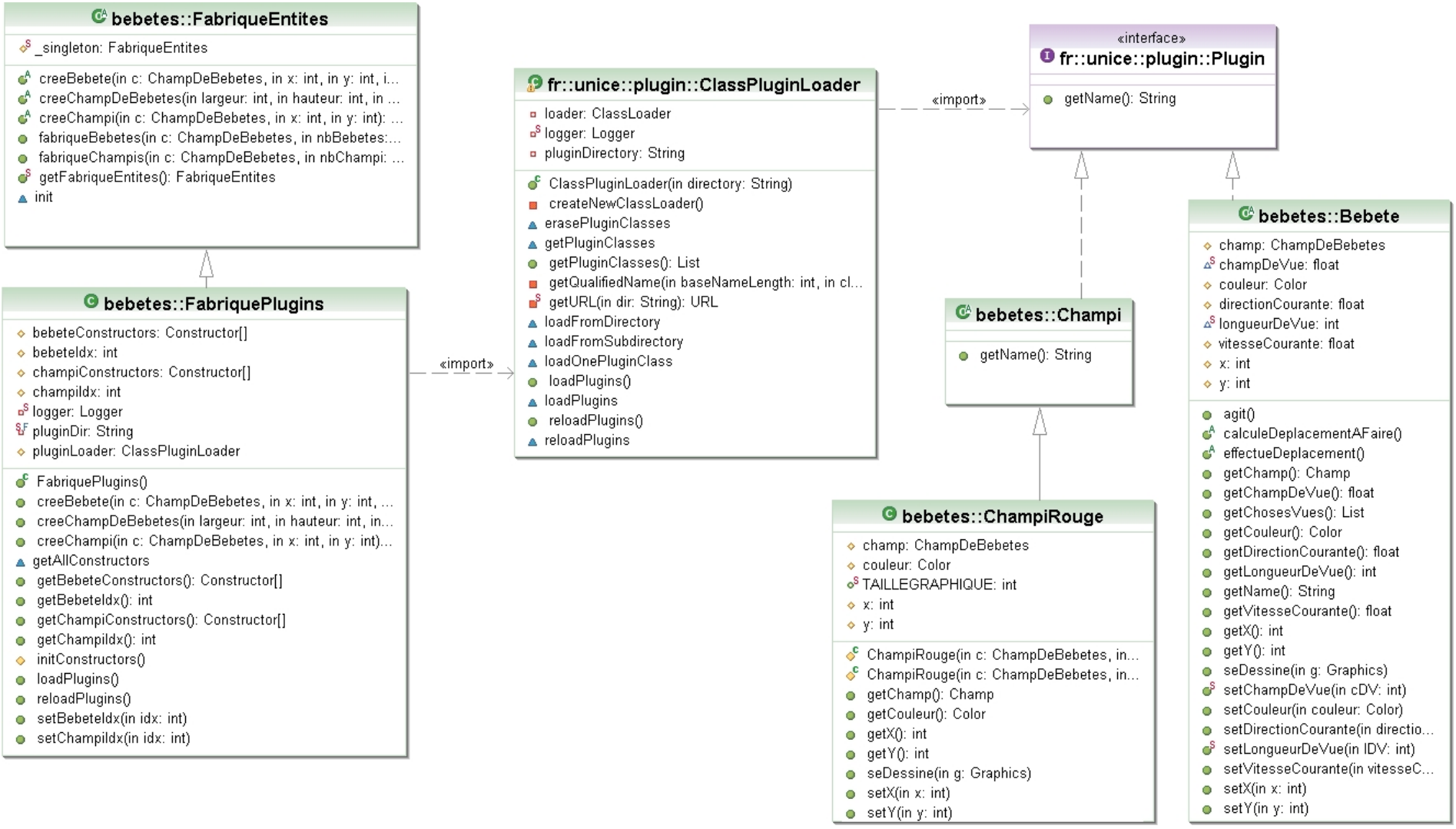
Changer les plugins sans relancer le programme principal

- Principe des serveurs de Servlets/Jsps : on ajoute des classes mais on ne relance pas le programme…
- Exemple
  - Serveurs d'application en Java
  - On ne l'arrête jamais…
  - On dépose des plugins et ils sont « découverts » à chaud…

(Re-)Chargement à chaud

- Principe : changer le class Loader à chaque chargement de plugin…
  - Chaque class loader gère un « cache » des classes, si on re-instancie le class loader, on peut charger/ recharger des plugins à chaud…
  - On utilisera un URLClassLoader pour ça…
  - Il suffit de re-scanner toutes les 5 secondes par exemple le répertoire qui contient les plugins… ou bien à la demande (clic sur un bouton)…

Application au simulateur



Explications (1)

- **ClassPluginLoader :**
  - Classe qui charge les plugins qu'elle trouve dans un répertoire. Ce répertoire est passé en paramètre au constructeur.
  - Elle conserve dans une liste la classe de chaque plugin chargé. On peut ensuite l'interroger pour lui demander cette liste.
  - Le chargement des classes est délégué à une instance de URLClassLoader qui est conservé dans une variable d'instance de PluginLoader. Pour avoir les nouvelles versions des plugins, il suffit de ne plus garder les références aux anciens plugins dans la liste des plugins et d'utiliser un nouveau chargeur de classes pour recharger tous les plugins.

- **Plugin :**
  - Les plugins sont des classes qui doivent implémenter l'interface Plugin. Les plugins peuvent être de différents types, représentés par une classe ou interface. Cette application utilise des plugins de type Champi et Bebete.

Explications (2)

- **FabriquePlugins :**
  - Une usine concrète, qui hérite donc de FabriqueEntites. Elle s'occupe de créer deux chargeurs de plugin, un pour les bébêtes, un pour les champis.
  - Elle récupère pour chaque plugin le "Constructor" qui lui permet d'implémenter la factory méthode correspondante (creeBebete, etc.)
  - Comme il peut y avoir plusieurs plugins de Bebetes (ou de champis), elle gère une liste et un indice dans la liste (par défaut 0). Les méthodes de création utilise le constructeur de plugin de l'indice courant. Cela permet de changer quelles bébêtes vont être créées par l'usine au fur et à mesure des (re-)chargements de plugins.
- **TestPluginsBebetes :**
  - Initialise la FabriquePlugins, crée des boutons qui pilote la FabriquePlugins (load, reload et redémarre la simulation).
  - Un menu par type de plugin est créé et contient la liste des classes de plugin chargées. Des écouteurs sur les menus permettent de configurer la FabriquePlugins comme indiqué ci-dessus. Les menus sont mis a jour en cas de chargement/ rechargement des plugins.
- **PluginMenuItemBuilder :**
  - Petit utilitaire pour construire les menus des plugins.