OSGI

OSGi est un groupe des spécifications qui définient un système modulaire et une plateforme de service pour langage Java , dans le cas quand elle est utilisé pour implémenter un modèle de component dynamique et complète qui n'existe pas dans l'environnement Java/VM. Ces components ont la forme des logiciels *bundlers* prêt a déployer avec l'autre logiciels (components jar avec extra headers), qui peuvent installer en distance, démarrer, arrêter, faire update ou désinstaller sans le besoin d'un reboot.

Architecture

Une Framework qui implémente les standards OSGi offre un environnement adapté pour modéliser l'application en petit bundlers. Chaque bundler est une collection des classes, jars et fichiers de configuration ou sont déclarés les dépendances externes.

Framework se divise en :

Bundlers : jars avec extra manifest headers

Services : qui connecte les bundlers dynamiquement en offrant POJO ou POJI (modelés des liaisons entre Java interfaces)

Services Registry : L'API pour ménager les services.

Cycle de la vie : L'API pour ménager le cycle de vie des bundlers (install, start, stop, update, uninstall)

Modules : Définie l'encapsulation et la déclaration des dépendances, comment un bundler peut importer ou exporter du code

Sécurité : Controlle les aspects de la sécurité en limitant les fonctionnalités de bundler jusqu'à ses capabilités.

Environnement d'exécution : Définie quelles méthodes et classes seront disponibles pour un plateforme spécifique

Dropin folders

Une manière de bas niveau d'effectuer des nouvelles installations en Eclipse (sauf update software ou p2 tools ou APIs) . p2 supporte la notion de voir, trouver les fichiers, ou sont déposés des scripts, contents exécutés pendant le startup ou running. Ce dossier s'appelle *watched directory* en général et *dropins* dans le cas d'Eclipse. Il est configuré d'être scanné dans chaque startup, comme le fichier des plugins. P2 permet de contrôler si les scripts ou plug-ins mis dans dropins n'ont pas des conflits avec les autre components et il peut aussi demandes des *pre-requisites* pour les nouveaux plug-ins.

Ils ont proposés des *layouts* différents pour le fichier dropins, des plus faciles, où on peut ajouter directement les plug-ins dans dropin directory, aux plus bien organisés avec des fichiers séparés.

```
eclipse/
   dropins/
```

```
emf/
  eclipse/
    features/
    plugins/
gef/
  eclipse/
    features/
    plugins/
... etc ...
```

Dans ce dernière organisation, on trouve des fichiers .link : **eclipse/dropins/emf.link.** Ça c'est un fichier .link comme le traditionnel fichier d'Eclipse pour mettre des files.link. Le format de ce fichier c'est le même comment le fichier java.util.Properties avec une clef *path* qui pointe vers la location externe. C'est une autre clef *optional*, dont le valeur est boolean, qui dit si le répertoire est considère optionnel ou pas.

Code injection

// on a cherché mais on a pas trouvé de la documentation sur ça pour l'instant

Plug-In and Package Dependencies

Eclipse permet de définir des dépendances entre les plug-ins basées sur le niveau de Bundle ou de Packages. Dans le MANIFEST.MF du plugin.xml on peut savoir si notre plug-in dépends d'un autre plug-in (et de tout ses packages exportés) ou bien il dépends seulement de la disponibilité de certains packages, peu importe quel plug-in exporte ce package.

Cela correspond dans MANIFEST.MF aux termes Require-Bundle : ou Import-Package :

Quelle approche est préférable ?

Apparemmment selon cet article qui a l'air sérieux :

http://blog.vogella.com/2009/03/27/required-bundle-import-package/

le mieux serait d'utiliser Import-Package à case du couplage très étroit entre les plug-ings. Au fait, si c'est un seul plug-in qui exporte le package, il ne subira aucun changement si jamais le plug-in qui l'a importé le remplace plus tard. Par contre, si on a le même package pour des plug-ins différents, alors on doit utiliser la dépendance entre plug-in.

L'avantage des dépendances entre plug-ins est qu'elles sont explicites, alors que avec IMPORT-PACKAGE n'est pas immédiatement évident de voir quel plug-in contribue ce package.

En conclusion, je pense que dans notre projet on devra faire appel à toutes les deux approches : celle du packages et du plug-in selon le besoin.