Projet de Fin d'études

-

Interface adaptative sur la plateforme Android

-

Rapport Final

Durey Guerric

Lebert Valérian

-

Tutrice : Frédériqe Laforest

Sommaire

Introduction

Nous allons vous présenter dans ce rapport notre travail effectué au sein du laboratoire Liris de mars à juin 2009. Ce projet s'est déroulé sous la tutelle de Mme Frédérique Laforest, maitre de conférences à de l'Insa de Lyon. Il a été effectué dans le cadre de notre Projet de Fin d'Etude, projet technique en autonomie quasi complète.

L'objet de ce projet consiste à développer une interface adaptative, sur Google Android. Cela permettra à la fois au laboratoire d'obtenir une interface adaptative modulaire facilement réutilisable, mais aussi d'évaluer le potentiel de la plateforme Android. Ce projet s'inscrit dans la continuité des études du laboratoire (et plus particulièrement de Mme Laforest) puisque des travaux d'adaptation d'interface avait déjà étaient réalisés, et de nombreux travaux sur les environnements pervasifs sont en cours.

1. Android & OSGI

1.1. Description du projet

L'application à réaliser est une application tournant sur Google Android et servant d'interface graphique à un environnement pervasif. L'implémentation de cet environnement pervasif est le sujet de PFE d'un autre groupe de 5TC. Il s'agit d'un environnement de capteurs et de services adressables via un protocole basé sur HTTP. Les spécifications techniques de ce protocole seront vues plus en détail dans le chapitre 1.4.

L'application doit alors être adaptable à l'environnement sans connaissance préalable des services qui seront disponibles. Elle doit donc être capable de proposer un GUI sur mesure et adaptable en temps réel.

1.2. Présentation d'Android

1.2.1. Architecture d'Android

Android est un système d'exploitation Open Source pour terminaux mobiles conçu par Android, une startup rachetée par Google en juillet 2005. Cet OS se différencie principalement de ses concurrents par le fait qu'il est ouvert. Le modèle économique de Google semble très pertinent, l'adoption d'Android par les fabricants sera probablement rapide du fait de la gratuité d'utilisation pour le constructeur. Voilà pourquoi il nous semble pertinent de travailler sur cette plateforme.

Les applications Android sont développées en JAVA. Android dispose d'un set de librairies qui inclut la plupart des fonctionnalités présentes dans JAVA ainsi que des fonctionnalités supplémentaires pour la gestion de l'interface graphique par exemple. Cependant, les applications de s'exécutent pas dans la machine virtuelle java sun mais dans la "Dalvik VM". Il s'agit d'une machine virtuelle développée par Google pour Android et adaptée aux systèmes restreints en mémoire et puissance de processeurs. Les classes java doivent être converties au préalable au format dex (grâce à l'outil dx). Enfin, une autre particularité importance est que chaque application s'exécute dans sa propre instance de Dalvik VM. En effet la Dalvik VM est conçue pour que le système puisse exécuter en même temps plusieurs instances de VM. C'est un avantage pour la stabilité du système, mais complique le partage des ressources et des classes entre les applications.

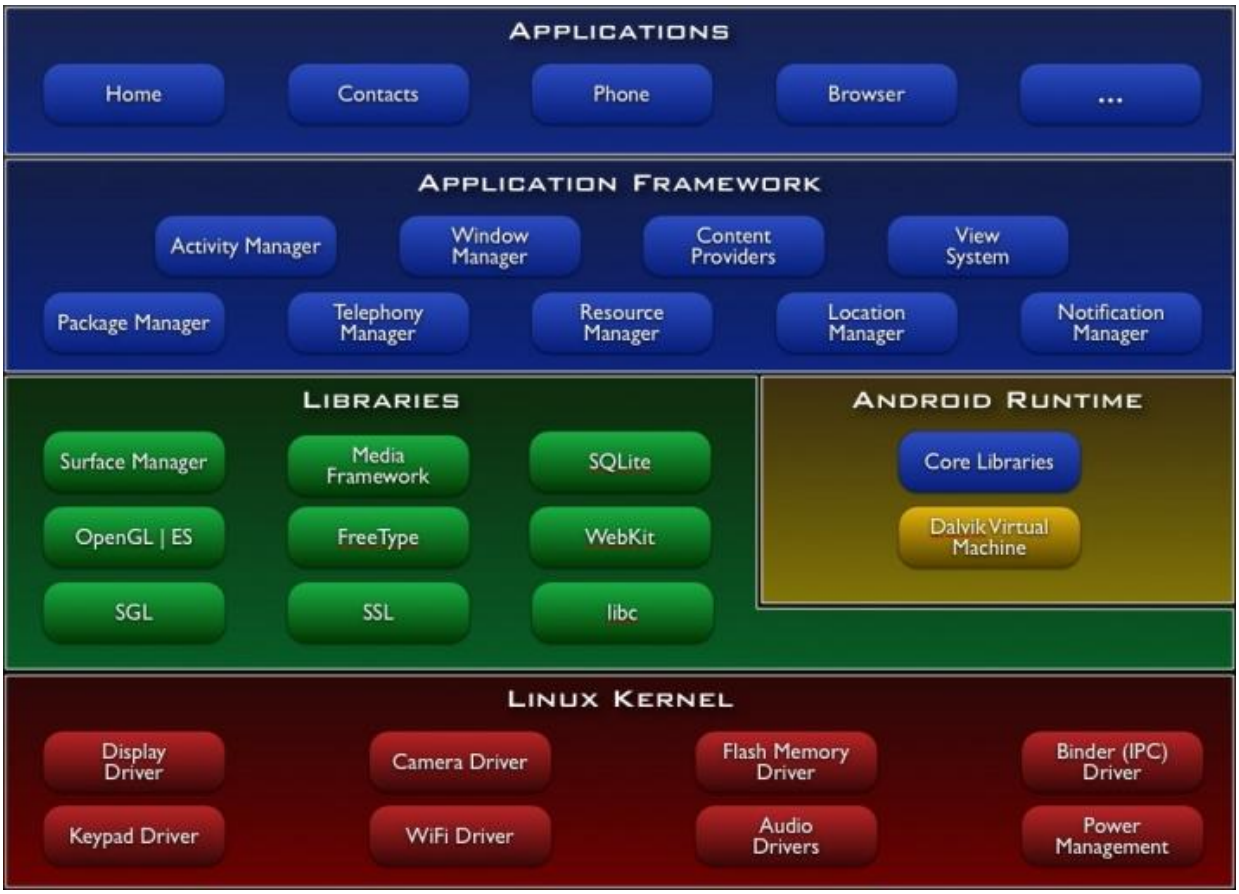


Figure 1: Architecture d'Android[1]

Les applications Android sont composées de 4 types de composants :

- *Activities*: Une Activity représente un écran de l'application. Une application peut avoir une ou plusieurs *activities* (par exemple pour une application de messagerie on pourrait avoir une Activity pour la liste des contacts et une autre pour l'éditeur de texte). Chaque Activity est implémentée sous la forme d'une classe qui hérite de la classe Activity.

- *Services*: Les services n'ont pas d'interface graphique et tournent en tâche de fond. Il est possible de s'inscrire à un service et de communiquer avec celui-ci en utilisant l'API Android.

- *Broadcast receivers*: Il se contente d'écouter et de réagir aux annonces broadcast (par exemple changement de fuseau horaire, appel entrant...)

- *Content providers* : Il permet de partager une partie des données d'une application avec d'autres applications.

En résumé Android permet de faire du partage de composant entre applications ainsi que de gérer leur cycle de vie, sans toutefois permettre la même souplesse qu'OSGI, comme nous le verrons dans la partie suivante.[2]

[1] http://developer.android.com/guide/basics/what-is-android.html
[2] http://developer.android.com/guide/topics/fundamentals.html

1.2.2. Interfaces graphiques:

Les Interfaces graphiques dans Android sont construites en utilisant les classes View et Viewgroup. La classe View sert de base à un ensemble de sous-classes appelées Widget qui permettent l'interface avec l'utilisateur (boutons, zone de texte...). La classe ViewGroup sert de base à un ensemble de sous-classes appelées layout qui servent à organiser les View dans l'espace. Ainsi, une interface graphique peut être représentée sous forme d'un arbre de Views:

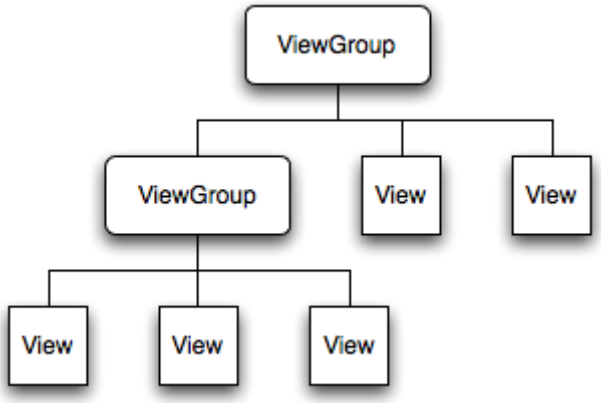


Figure 2 : Hiérarchie des vues

Pour faire apparaître cette interface graphique, la classe principale (Activity) doit appeler la méthode setContentView() en passant en référence le noeud racine.

Il y a 2 méthodes pour implémenter une interface graphique dans l'API Android:

- L'arbre peut être décrit selon un langage XML, qui est converti à la compilation du projet en une ressource "View" instanciable depuis le programme principal. L'interface peut alors être chargée avec la méthode setContentView().
- On peut instancier directement les éléments dans le code de l'application et en manipulant leurs méthodes pour aboutir au résultat voulu.

L'avantage de décrire l'interface graphique en XML est que cela permet de mieux séparer la partie présentation de la partie exécution. Cependant, la description XML doit être faite avant la compilation du projet. C'est pourquoi nous utiliserons la seconde méthode afin de charger dynamiquement des morceaux d'interface graphique durant l'exécution.

1.3. Pourquoi utiliser OSGI sur Android

1.3.1. Présentation d'OSGI

L'**OSGi** est une organisation fondée en mars 1999. L'Alliance et ses membres ont spécifié une plate-forme de services basée sur le langage Java qui peut être gérée de manière distante. Le cœur de cette spécification est un framework (cadriciel) qui définit un modèle de gestion de cycle de vie d'une application, un référentiel (registry) de services et un environnement d'exécution.

OSGI introduit une architecture orientée service permettant une plus grande indépendance entre les différents composants d'un programme. Ainsi, on répond aux problématiques de réutilisabilité, d'interopérabilité et de réduction de couplage. On parle ainsi de fournisseurs de services et de consommateurs de services, les fournisseurs mettant à disposition des composants. Chaque composant est donc facilement modifiable, permettant un débug et des mises à jour sans difficultés.

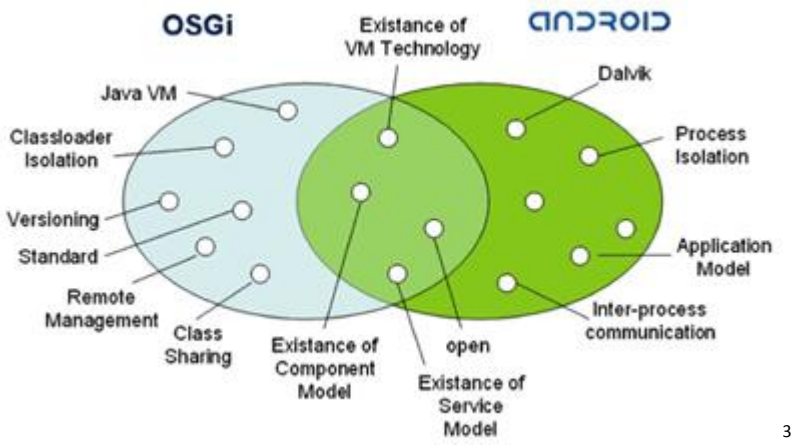
OSGI étant une spécification, il en existe plusieurs implémentations, telles que Apache Felix, Equinox ou Knopflerfish.

1.3.2. Comparaison OSGI/Android

OSGi et Android définissent une plate-forme de développement orientée composants et fournissent une mise en œuvre de l'architecture orientée services, au sein de l'appareil mobile. OSGi peut être utilisé dans un certain nombre d'environnements Java standards. Il est intéressant de noter que s'il est converti en Exécutable Dalvik, OSGi peut être exécuté sur Android.

La différence clé entre les 2 plateformes provient du fait que toutes les applications Android sont exécutées dans des machines virtuelles séparées (une instance de VM par application), alors qu'OSGI exécute toutes les applications dans la même VM. Exécuter les applications dans des VM séparés – c'est-à-dire des processus d'OS séparés – a l'avantage de procurer une gestion de ressources plus rigoureuse. Cependant, ce modèle exige plus de mémoire et le partage de services entre les composants est plus complexe et plus couteux.

Avec OSGI, le partage de ressources entre les différents composants (bundle) se fait par une simple déclaration dans le fichier manifest. Durant l'exécution, tous les bundles partagent le même contexte (classe BundleContext) et peuvent accéder aux ressources auxquelles ils ont droit.



[3]

Figure 3: Comparaison d'Android et d'OSGI

[3] http://picisblog.blogspot.com/2008/02/introduction-google-and-open-handset.html

Les deux plates-formes donnent des solutions très différentes pour la même question. Le modèle basé sur l'isolement Processus dans Android est un avantage dans la gestion des ressources, mais la difficulté du partage de classes limite sérieusement l'ergonomie de cette solution.

Un point fort d'OSGI est qu'il peut être utilisé sur de nombreuses plateformes (dont Google Android). Nous avons donc choisi d'utiliser OSGI pour le développement d'une interface adaptative.

1.4. Présentation de l'environnement pervasif

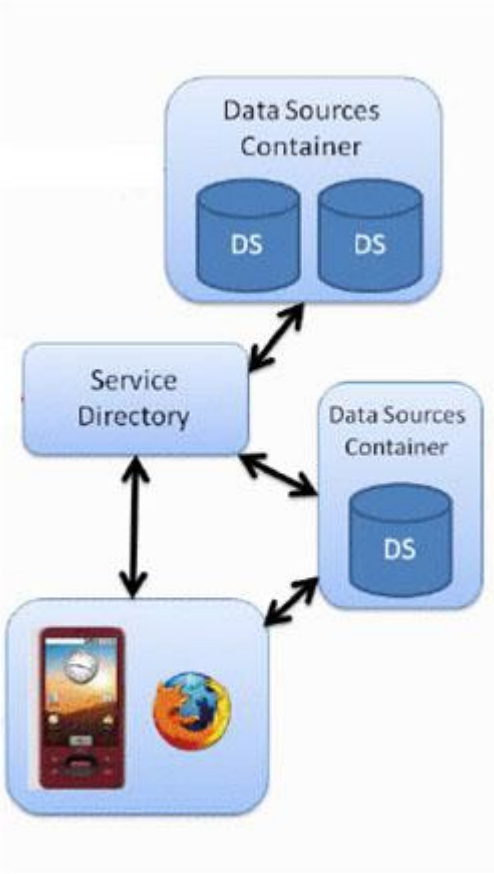


Figure 4: Architecture du composant services de l'environnement pervasif

Le sujet de notre PFE était initialement « développement d'une interface adaptative sous Google Android ». Après avoir découvert la plateforme Android et l'environnement OSGI, il nous fallait trouver quelle application nous souhaitions concrètement réaliser.

Nous avons alors décidé de nous appuyer sur le projet d'un autre groupe de PFE travaillant dans notre salle sur un composant de gestion de service pour un environnement pervasif.

Comme l'on peut le voir sur le schéma ci-contre, leur architecture comporte un annuaire de services (Service Directory) et plusieurs containers contenant 1 ou plusieurs services (DataSources).

Le client s'adresse au Service Directory pour avoir la liste des services ainsi que l'IP/Port du DSContainer associé. Ensuite il s'adresse directement au container pour utiliser le service.

La communication avec l'annuaire et les services se fait par des requêtes HTTP. Les clients existant sont donc un simple navigateur dans lequel on tape les requêtes dans la barre d'adresse ou bien notre application Android.

Les requêtes permettent d'obtenir une description logique et une description « human readable » des services proposés. Pour l'instant les services proposent à l'utilisateur un ensemble de commandes avec un certain nombre d'inputs et d'outputs.

Voici un exemple communications possible entre un client et l'environnement pervasif :

Liste des DataSource *disponibles*	`HTTP_GET(IP:Port Service Directory) /datasource`
	Réponse:
	`/datasources`
	`/time:134.214.107.71:8080`
	`/math:134.214.107.71:8080`
	`/automatic:134.214.107.71:8080`

Description d'un DataSource	`HTTP_GET(134.214.107.71:8080) /time/desc`
	Réponse
	`This DataSource provides local time and is updated in real time`

Structure logique d'une commande	`HTTP_GET(134.214.107.71:8080) /time/command`
	Réponse
	`/getTime/withoutDisplay/Display current time`
	`/inputs`
	`/outputs`
	`/time:AT_string:Current time`

Execution d'un commande	`HTTP_GET(134.214.107.71:8080)`
	`/time/command/getTime`
	Réponse
	`/time`
	`/Tue Jun 23 14:53:32 CEST 2009`

En premier lieu, l'annuaire retourne le nom des services disponibles et l'IP du DScontainer associé. Ensuite il faut s'adresser directement au DScontainer pour utiliser un service. On peut obtenir une description du service ainsi que la structure de ses commandes. Chaque input ou output a un nom, un type et une description. Dans notre projet le seul type utilisé est la chaîne de caractère (AT_string).

Le contexte étant maintenant posé, nous allons passer à la partie réalisation.

2. Travail Réalisé

2.1. Intégration d'OSGI dans une application Google Android

Pour notre projet, nous avons choisi l'implémentation d'OSGI « Apache Felix ». A priori, Android étant compatible avec les applications JAVA nous aurions pu choisir n'importe quelle implémentation. Cependant, quelques légères adaptations sont tout de même nécessaires, en particulier pour le chargement dynamique des classes, qui doit se faire à partir de bytecode au format dex et non au format JAVA standard.

Nous avons trouvé 2 implémentations d'OSGI compatible avec Android:

- **ProSyst mBedded Server Android Edition** : C'est une implémentation basée sur equinox. Elle est livrée avec plusieurs outils graphiques permettant d'administrer le serveur OSGI. Cependant dans l'état elle ne permet pas aux bundles de point d'entrée sur l'interface graphique d'Android. De plus, n'ayant pas le code source de cette implémentation nous n'avons pas pu réaliser les modifications nécessaires.

- **Apache Felix** : Cette implémentation est compatible Android depuis la version 1.0.3. De plus, plusieurs sources nous ont permis réaliser quelques exemples de bundles réalisant un affichage via l'API graphique proposé par Android.

Bien que Felix soit compatible avec android, il est en fait assez compliqué de faire tourner un serveur Felix sur l'émulateur et de l'utiliser via le shell d'Android (cf. annexe 1). Malheureusement, ce mode ne permet aucun affichage sur le téléphone. La plus grosse difficulté est de fusionner une application Android avec une instance d'Apache Felix de manière à pouvoir réaliser une application graphique sur une architecture OSGI. Il est en effet nécessaire de partager à la fois le contexte OSGI (BundleContext) et le contexte Android (Activity).

Dans nos recherches nous avons trouvé une ébauche de solution, relativement vieille (les 2 technologies Android et Felix évoluent très vite) dont nous nous sommes inspirés pour notre application. Malheureusement celle-ci utilisait une version béta de l'émulateur Android et certaines fonctions avaient disparues. Nous avons donc choisi de disséquer les sources afin de comprendre comment l'application était capable d'embarquer Apache Felix. Après confrontation avec un autre bout de code trouvé sur le net et utilisant IPOJO, nous avons finalement compris que le serveur OSGI était configuré avec un « FileInstall », bundle qui charge tous les bundles dans un répertoire donné. Nous avons donc tout d'abord vidé celui-ci afin de vérifier que le simple fait de lancer cette version de Felix sur Android était possible, et que le programme Android n'utilisait que des fonctions supportées par notre version de l'émulateur. Ce fut heureusement le cas, et bien que notre programme Android n'affichait alors qu'un écran noir, il ne plantait pas. Nous nous sommes alors basée sur cette version et configuration d'apache Felix (les fichiers seront donné dans notre archive).

Afin de s'assurer du bon déroulement de l'opération et de tenter quelques chargements de bundles, nous avons commencé par chercher à mettre en place un shell en utilisant les bundles présents sur le site de Felix, à savoir le « remote shell ». Cependant, cette tentative fut un échec, car l'ouverture d'une session telnet n'a pas fonctionné aussi facilement que prévu. Nous avons donc décidé de développer notre propre shell graphique. Cela nous à pris énormément de temps, car nous ne

savions pas quels étaient les causes de nos erreurs (faute de shell !). Pour cela, nous avons repris la procédure trouvé dans les différents exemples collectés jusqu'ici, afin d'avancer à tâtons jusqu'à réussir à créer un bundle affichant une fenêtre, puis étant capable d'offrir un GUI pour le shell Felix.

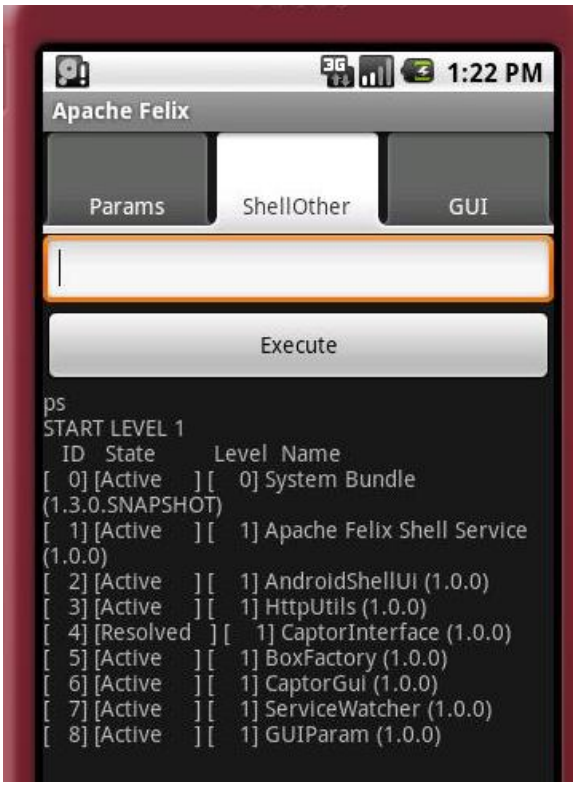


Figure 5 : Résultat d'une commande OSGI sur notre shell

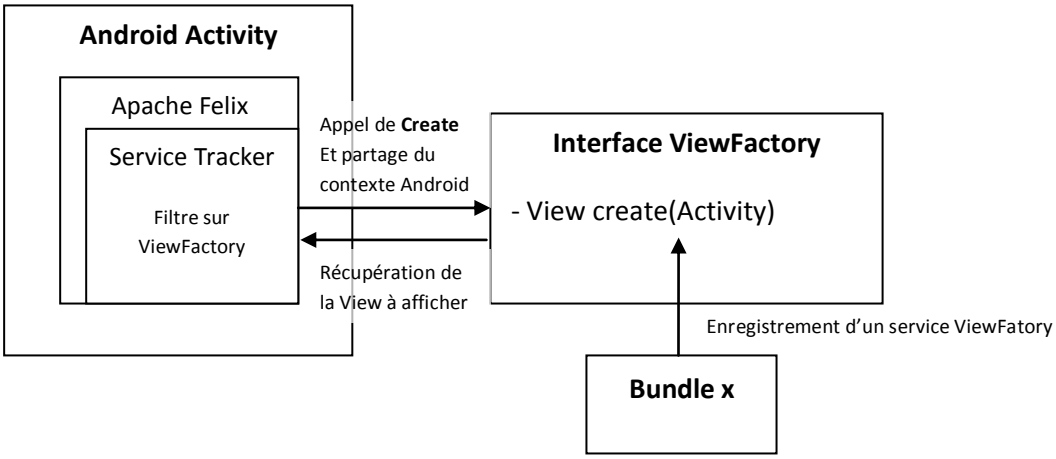
A partir de là, le projet a pu réellement avancer. Nous avions compris comment interagir avec la partie graphique et étions capable de charger et démarrer des bundles à chaud. Afin d'avoir en parallèle le shell et notre projet, nous avons modifié notre application pour que chaque bundle souhaitant accéder à l'affichage graphique dispose de son propre onglet.

A son lancement, l'application Android (classe Activity) lance une instance de Felix. Celui-ci crée au démarrage un « ServiceTracker » (org.osgi.util.tracker.ServiceTracker) [4]avec un filtre sur les services que nous appelons « ViewFactory ». Dès qu'un service ViewFactory est enregistré dans l'environnement OSGI, le ServiceTracker utilise la fonction ViewFactory.create(Activity) qui permet de transmettre le contexte Android et de récupérer en retour une View à Afficher.

Pour que chaque bundle qui souhaite afficher une ViewFactory sur l'interface graphique dispose de son onglet qui lui est propre, le serveur Felix doit être lancé dans une TabActivity (et non simple Activity). Nous enregistrons alors les services ViewFactory avec une propriété « TabName=nomDuTab ». Ainsi nous utilisons cette propriété pour que le ServiceTracker crée un Tab pour chaque ViewFactory).

[4] Un service tracker sert à « traquer » l'apparition, la disparition ou la modification des services dans l'environnement OSGI et réagir en fonction.

Les bundles peuvent donc désormais créer leur propres Views en enregistrant un service « ViewFactory ». Comme dit précédemment, le premier bundle que nous avons réalisé est une interface graphique pour le shell de Felix (org.apache.felix.shell-1.0.0.jar en raison de la version de notre Felix). Nous disposons alors d'un environnement Android-OSGI pleinement fonctionnel.



2.2. Explication de notre solution

2.2.1. Liste des classes

Bundle	Felix4Android	CaptorGui	BoxFactory	httpUtils
Classes	ApacheFelix	CaptorGui CaptorGuiViewFactory	Box BoxFactory BoxFactoryActivtor BoxDestroyer CommandProcesser InfoButtonListener	Activator SimpleHttpClient
Interfaces	ViewFactory		BoxFactoryService	SimpleHttpClientInterface
OSGI Imports *		Boxfactory	captorInterface	

Bundle	CaptorInterface	ServiceWatcher	GuiParam
Classes	Activator Attribute Command CaptorInterface	Activator ServiceInfo ServiceWatcher	GuiParam GuiParamViewFactory
Interfaces	CaptorInferfaceInterface	ServiceWatcherInterface	
OSGI Imports *	httpUtils	httpUtils	ServiceWatcher

* Les imports du framework OSGI et Android ne sont pas détaillés ici.

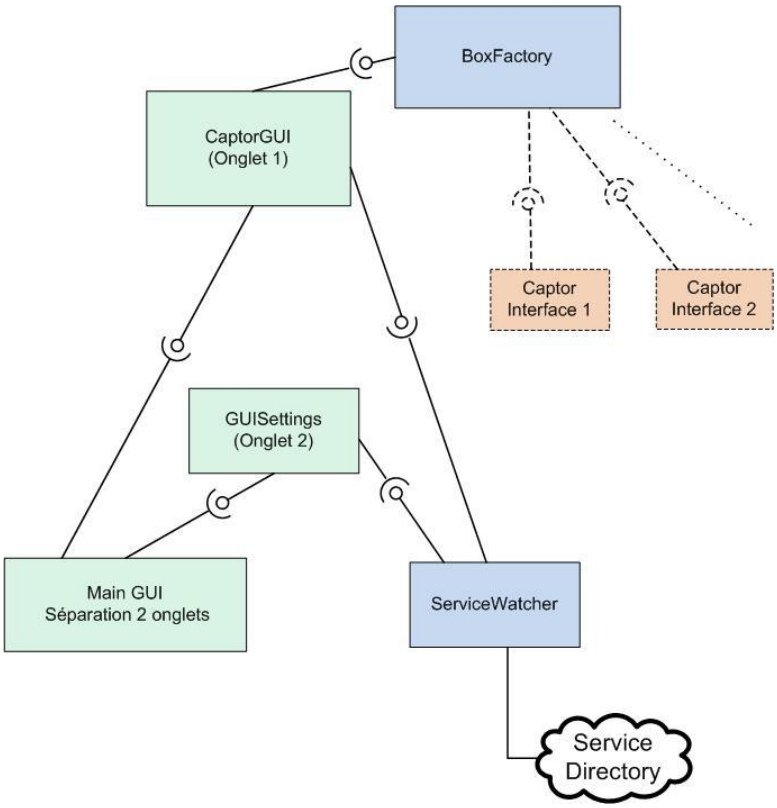


Figure 6 : Interaction des composants de l'application

2.2.2. Explication

Pour éviter toutes confusions entre la notion de service au sens OSGI et celle de service de l'environnement pervasif nous utiliserons respectivement l'appellation service OSGI et DataSource.

Pour l'instant nous n'implémentons que la partie « commandes » de l'environnement pervasif car les DataSource ne disposent pour l'instant pas de la notion de flux. Notre application doit donc être capable de détecter les DataSource présents et d'afficher une interface graphique permettant d'utiliser les commandes que ces DataSource proposent.

Nous avons donc tout d'abord un service OSGI « ServiceWatcher » chargé de la découverte des DataSource disponibles dans l'environnement pervasif. Un onglet de l'interface graphique « Params » utilise alors ce service pour permettre à l'utilisateur de choisir les services qu'il veut utiliser.

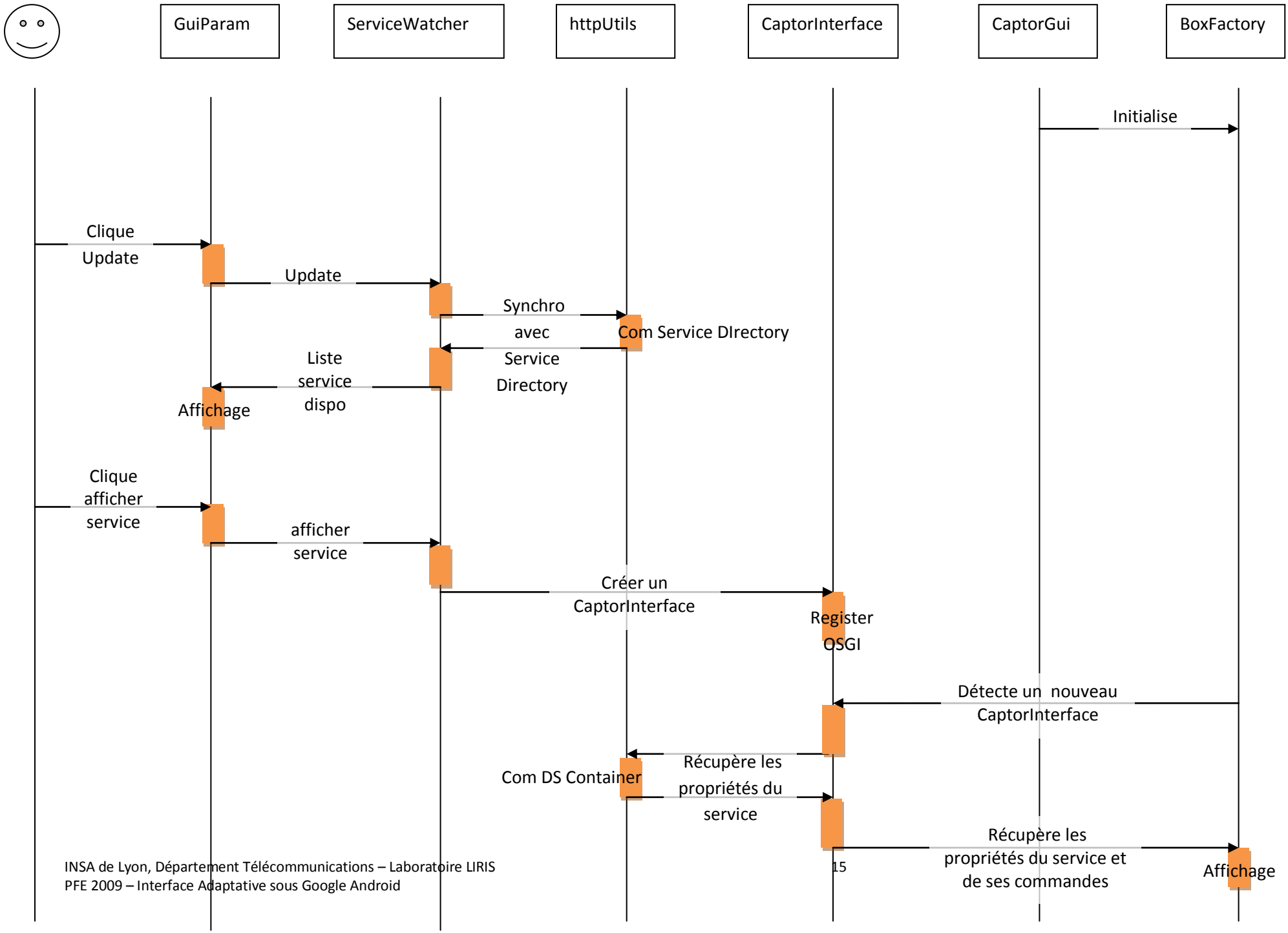
Lorsque l'utilisateur choisit d'utiliser un DataSource, « ServiceWatcher » enregistre alors un nouvel élément dans l'environnement OSGI : un service de « CaptorInterface » (qui aurait dû s'appeler en fait DatasourceInterface) qui permet de s'adresser à ce DataSource de l'environnement pervasif.

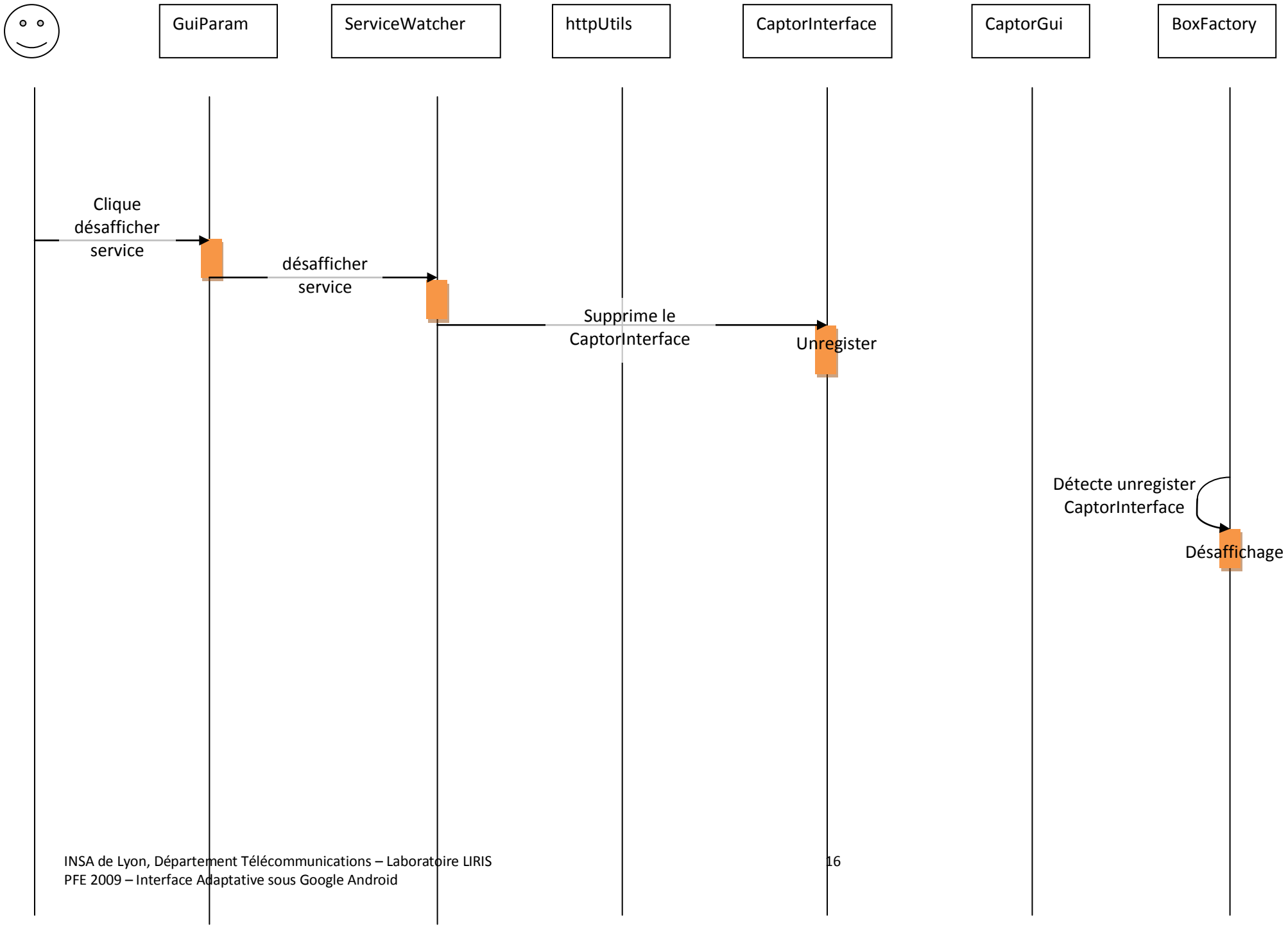
D'un autre coté, un service OSGI « BoxFactory » est chargé de générer une interface graphique à partir de la description d'un service. Ainsi grâce à un ServiceTracker, il guette l'apparition d'un service « CaptorInterface » dans l'environnement OSGI. Puis, il s'adresse à lui pour récupérer la description des commandes proposées par le DataSource.

CaptorGui est l'onglet d'affichage des Box. Pour que BoxFactory sache dans quelles View afficher les Box, il est au préalable initialisé par CaptorGui.

Pour supprimer une Box (si l'utilisateur choisit de ne plus utiliser un DataSource), nous utilisons encore une fonctionnalité d'OSGI : le ServiceListener. Au moment de la création d'une Box, nous disposons alors d'une référence sur la View correspondante. Nous créons alors un ServiceListener, qui, à la disparition du service CaptorInterface correspondant détruira la View. Il suffit alors de faire un « unregister » (méthode OSGI pour supprimer un service) pour que la Box associée au CaptorInterface disparaisse.

Ci dessous un diagramme de fonctionnement de l'application :


GuiParam
ServiceWatcher
httpUtils
CaptorInterface
CaptorGui
BoxFactory
Initialise
Clique
Update
Update
Synchro
avec
Service
Directory
Com Service DIrectory
Liste
service
dispo
Affichage
Clique
afficher
service
afficher
service
Créer un
CaptorInterface
Register
OSGI
Détecte un nouveau
CaptorInterface
Récupère les
propriétés du
service
Com DS Container
Récupère les
propriétés du service et
de ses commandes
Affichage

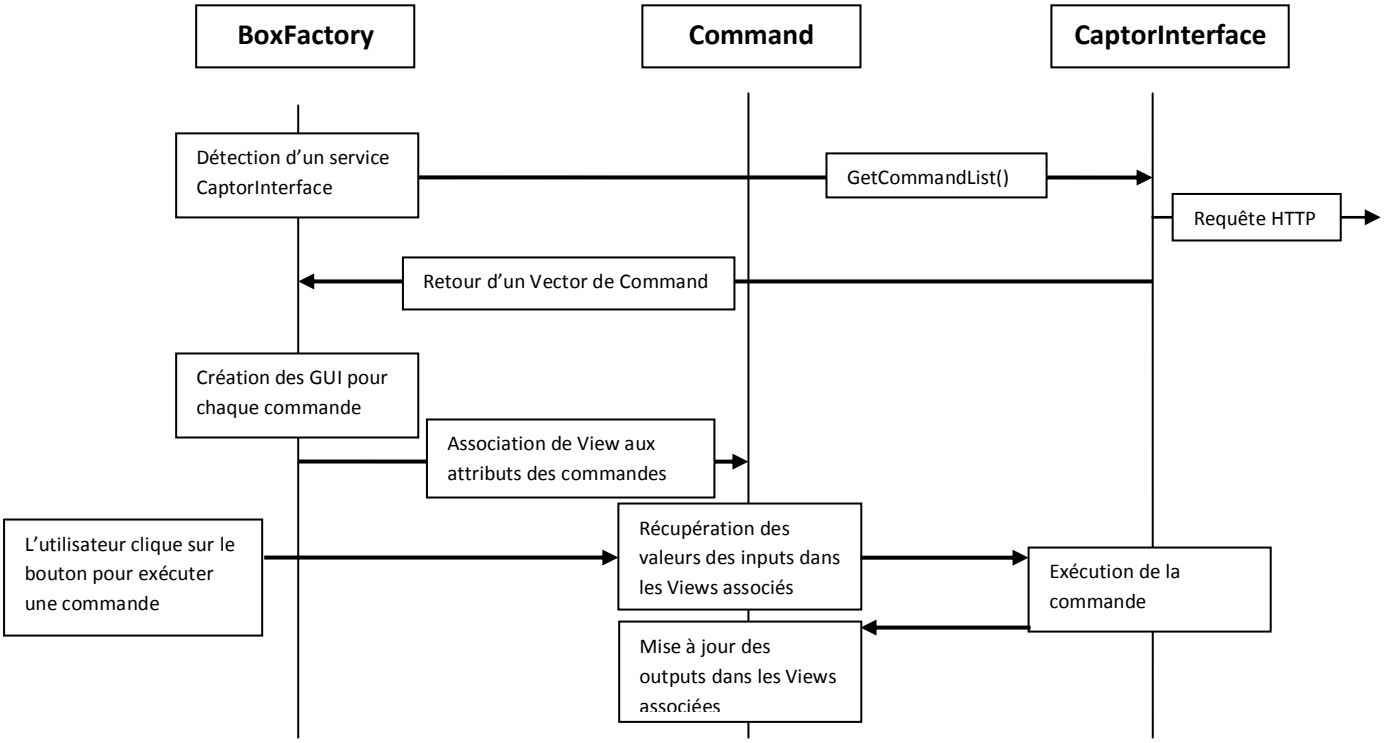

GuiParam
ServiceWatcher
httpUtils
CaptorInterface
CaptorGui
BoxFactory
Clique désafficher service
désafficher service
Supprime le CaptorInterface
Unregister
Détecte unregister CaptorInterface
Désaffichage

Pour utiliser les commandes, nous avons créé une classe « Command » ainsi qu'une classe « Attribute ».

Un Attribute représente un paramètre d'entrée ou de sortie d'une commande. Un attribut à

- Un nom
- Un type
- Une valeur
- Une View android associée
- Une description
- Les méthodes pour la manipulation de ces variables

Une Command possède un Vector d'Attributs d'entrée, un Vector d'attributs de sortie ainsi que des fonctions de manipulation. Ces objets sont créés suite au parsing de la réponse à la requête HTTP « /<service>/command » par CaptorInterface.



Quand BoxFactory demande la description d'une commande à CaptorInterface celui-ci retourne un objet de classe Command. BoxFactory génère donc l'interface en se basant sur le nom des attributs, leur type et au moment de la création de l'interface il leur associe leur View respective (TextView et EditText pour les strings).

Ainsi pour exécuter la commande on peut récupérer les valeurs des inputs et afficher la sortie des outputs dans leur View respectives.

2.3. Adaptation d'interface

2.3.1. Adaptation à l'environnement

L'application sert d'intermédiaire entre l'environnement pervasif et l'utilisateur. Cet environnement évolue dans le temps et l'application doit s'adapter automatiquement aux services disponible à l'instant T. Ces services sont à l'origine inconnus, c'est pourquoi ils doivent mettre à disposition une description complète pour que l'application puisse générer une interface compréhensible par l'utilisateur.

Le module central pour l'adaptation est le BoxFatory. A partir de la structure et description du service il dispose sur l'écran les éléments graphiques nécessaires à son utilisation. La version actuelle de l'environnement pervasif utilise uniquement des inputs et outputs de type String et les éléments graphiques nécessaires sont donc principalement des zones de texte éditables et non éditables pour les inputs/outputs.

Figure 7: Interface utilisateur du service "math"

Bien que nous n'utilisions à ce jour que des attributs de type string, la classe Attribute dispose d'ors et déjà du champ « type » et il serait facilement envisageable d'utiliser d'autres widgets selon les types (curseur glissant pour les float par exemple…).

Pour que l'interface générée soit compréhensible par l'utilisateur, il faut que les noms des commandes et des paramètres des services soit parlant. Une description plus complète est accessible à l'utilisateur en cliquant sur un élément. Nous utilisons dans ce cas le widget « Toast » de l'API Android qui est parfaitement adaptée à cet usage.

Figure 8: Toast

2.3.2. Adaptation à l'utilisateur

Le module BoxFactory étant un service OSGI, il est envisageable de charger un BoxFactory différent en fonction des préférences de l'utilisateur (tailles de l'écran, option d'affichage). Cependant comme nous le verrons dans le paragraphe suivant nous avons rencontré des difficultés à réaliser des interfaces complexes car Android gère mal une hiérarchie de View trop étendue.

Nous avons donc envisagé un cas plus simple : l'utilisateur peut choisir entre 2 modes :

- **Multi-services mode** : L'utilisateur choisit les services qu'il souhaite utiliser et les « Box » correspondantes sont affichés dans un second onglet.

Figure 10 : Onglet de choix des datasources

Figure 9 : Onglet d'utilisation des datasources

- **Mono-service mode** : L'utilisateur dispose d'un menu déroulant pour choisir l'unique service qu'il souhaite utiliser (dans le même onglet).

Figure 11: Vue en "Single service mode"

L'utilisateur change de mode en cliquant sur le bouton « switchview ».

Pour ce faire il suffit d'une simple réinitialisation du BoxFactory (BoxFactory dispose d'une méthode «initialize » pour lui dire dans quelle View afficher les Box). Nous l'initialisation avec une View sur l'onglet « Params », puis rendons invisible les onglets.

Par la suite nous avons également modifié BoxFactory pour que selon le mode, il n'affiche plus en titre le nom du service (le nom est déjà affiché dans le menu-déroulant). Le but de cette modification est de simplifier l'arbre des View (on réduit d'un niveau) et d'éviter ainsi un plantage d'Android.

2.4. Problèmes rencontrés

2.4.1. Deep View Tree

Quand nous avons cherché à optimiser la présentation de l'application pour l'écran du terminal mobile, nous avons été confrontés à des erreurs « StackOverflow Exception » qui apparaissaient de manière assez aléatoire.

En recherchant sur internet, nous avons découvert qu'il s'agit d'un problème de Google Android déjà connu : Android supporte mal les arbres de Views trop grand, les problèmes apparaissant au delà de 13 niveaux (Deep View Tree). Nous avons alors essayé de simplifier au maximum l'arbre. Cependant, l'utilisation d'onglet avec un titre d'application nécessite déjà la création d'une hiérarchie à 6 niveaux. De même, si l'on veut disposer certain éléments horizontalement et d'autres verticalement il faut imbriquer un layout (conteneur) horizontal dans un layout vertical, soit 3 niveaux en comptant les widgets. Il faut encore ajouter un layout pour avoir une barre de défilement (scrolling). La limite des 13 vues est dès lors rapidement atteinte et nous n'avons pas pu travailler sur la présentation aussi facilement que nous l'aurions souhaité. Un outil fourni avec le kit de développement permet d'afficher en tant réel la hiérarchie des vues :

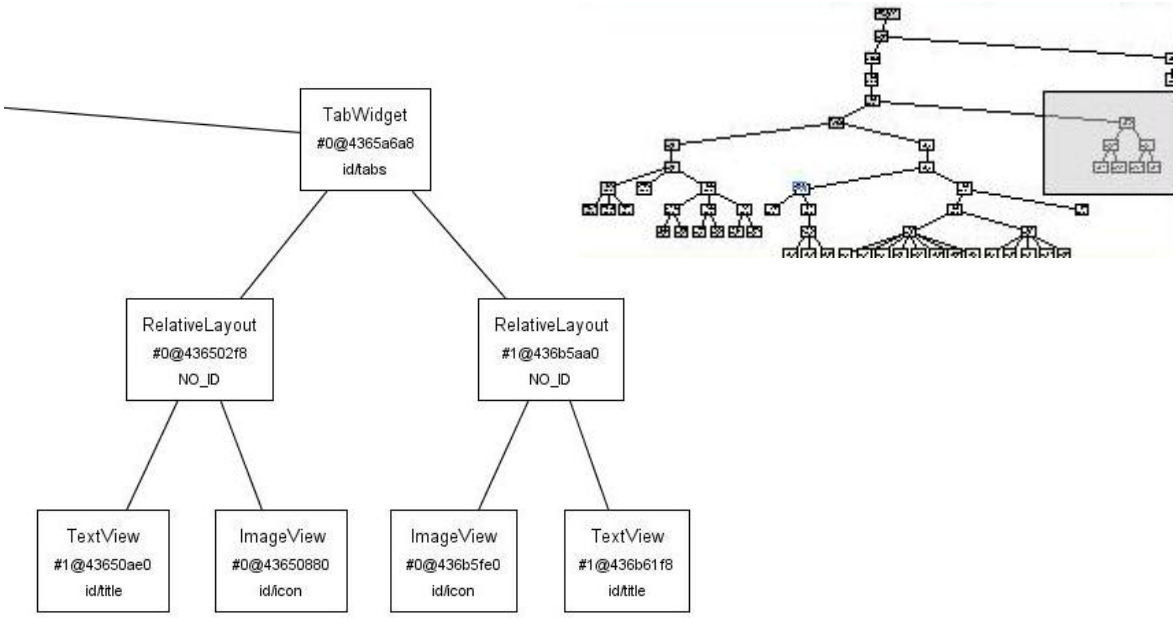


Figure 12: Arbre des Views

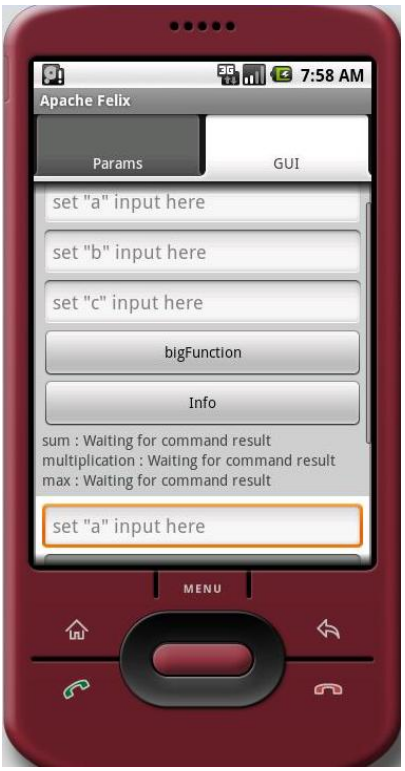
Cette capture issue de notre application montre à quel point l'arbre se complexifie rapidement.

Voila ce que l'on peut trouver sur le blog officiel de google :

http://android-developers.blogspot.com/2009/04/future-proofing-your-apps.html

Technique to Avoid, #3: Going Overboard with Layouts

Due to changes in the View rendering infrastructure, unreasonably deep (more than 10 or so) or broad (more than 30 total) View hierarchies in layouts are now likely to cause crashes. This was always a risk for excessively complex layouts, but you can think of Android 1.5 as being better than 1.1 at exposing this problem. Most developers won't need to worry about this, but if your app has very complicated layouts, you'll need to put it on a diet. You can simplify your layouts using the more advanced layout classes like FrameLayout and TableLayout.



Comme nous pouvons le voir, ces restrictions nous limitent énormément dans notre développement. Notre interface étant adaptative, il est en plus beaucoup plus simple d'utiliser de nombreux layout afin de pouvoir hiérarchiser notre programmation facilement. Cela nous a contraints à utiliser un affichage des plus basiques. La limite des 13 vues étant atteintes, nous somme par exemple obligés de disposer les arguments et les résultats les uns en dessous des autres.

La superbe fonction math en est le plus bel exemple : impossible d'afficher « bigFonction » et « Info » côte à côte, et il en va de même pour les paramètres. L'écran étant déjà petit, nous sommes obligés de gâcher de la place et d'utiliser un scrolling.

N'étant bien sur pas au courant de cette limitation lors de notre premier design d'interface, nous avons choisis d'utiliser des onglets. Cependant, si l'on souhaite réellement construire une interface ergonomique, nous pensons qu'il faudrait repenser la présentation des différentes fonctions, et se passer d'onglet.

Pour ce qui est des FrameLayout et des TableLayout, ceci ne résout pas notre problème : Les FrameLayout on été conçus pour afficher des éléments un par un, alors que les TableLayout reviennent à utiliser une vue pour la table, plus une vue pour chaque colonnes (2 niveaux dans le View tree). Cela réduit effectivement le nombre de vue totales, mais pas la profondeur de l'arbre.

2.4.2. Utilisation de l'API graphique

Le problème précédent est lié à une autre difficulté que nous avons rencontrée : Android permet d'implémenter l'interface graphique de 2 manières (cf.chapitre 1.2.2) :

- L'arbre peut être décrit selon un langage XML, qui est converti à la compilation du projet en une ressource "View". L'interface peut alors être chargée avec la méthode setContentView().
- En instanciant directement les éléments dans le code de l'application et en manipulant leurs méthodes pour aboutir au résultat voulu (comme Swing en java).

L'utilisation du fichier XML permet de séparer la partie présentation de la partie application. Cependant elle n'est pas adaptée à notre cas car la présentation doit être définie avant la compilation, et ne peut donc pas s'adapter à l'environnement.

Nous avons donc utilisé la seconde méthode, seulement certaines possibilités ne sont utilisables qu'avec l'utilisation du langage XML. Par exemple la notion de « weight » permet aux vues d'occuper un espace proportionnel à leur poids, et donc de s'adapter à la taille de l'écran. L'utilisation de telles fonctionnalités aurait pu nous permettre de simplifier la complexité de l'arbre de View tout en offrant une meilleure disposition sur l'écran.

L'implémentation de l'interface graphique directement dans le code Java complexifie le code d'une part, et ne permet pas d'utiliser pleinement les possibilités d'Android pour la présentation à l'écran.

2.5. Mode d'emploi, Tests et résultats

Pour utiliser notre application, il faut installer l'application Android « Felix4Android » sur l'émulateur. Cette application comprend simplement un affichage vide et lance un serveur Apache Felix qui à son tour lance dans l'ordre les bundles nécessaires à l'application. L'ensemble des bundles décrits au chapitre 2.2.1 doivent être copiés sur l'émulateur dans le dossier « /data/felix2/bundles ». Le chemin de ce dossier peut être changé en modifiant la configuration du serveur Felix dans l'application « Felix4Android ». Il ne faut pas oublier de changer les droits du dossier « /data/dalvik-cache » (chmod 777) car felix a besoin d'utiliser ce cache pour fonctionner correctement. Ensuite, on peut lancer l'application en utilisant l'interface du téléphone et les 2 onglets se chargent.

Notre client à été testé avec succès avec tous les services développés par nos collègues. L'ensemble des fonctions peuvent être utilisées telles que :

- Ouverture et fermeture d'une porte (représentée par un légo)
- Utilisation des lumières du composant sunspot
- Récupération de valeurs (température, luminosité, puissance du signal)
- …

Le but recherché a été atteint puisque nos collègues pouvaient développer de nouveaux services qui étaient directement utilisable par notre client sans modification de celui-ci.

3. Retour d'expérience

3.1. Environnement de travail

Android met à disposition un environnement de travail performant pour le développement d'applications. Il s'agit d'un plugin pour Eclipse permettant de packager facilement une application Android et de connecter Eclipse à l'émulateur pour uploader l'application et afficher sur la console « debug » les sorties texte. Cette console « debug » devient rapidement indispensable. Il est également possible d'exécuter l'application pas à pas avec des points d'arrêt pour débugger une application.

Pour travailler à 2, nous avons utilisé un SVN sans lequel nous n'aurions pu avancer simultanément sur le projet.

Pour packager nos bundles OSGI, nous avons préféré utiliser de simple scripts **.bat* plutôt qu'on outil comme Maven qui nous a semblé bien trop complexe pour notre projet. De plus étant à l'origine novice sur l'architecture OSGI il nous est apparu nécessaire de faire le maximum d'étapes manuellement afin de bien comprendre l'architecture OSGI et l'utilisation des bundles.

3.2. Gestion de projet

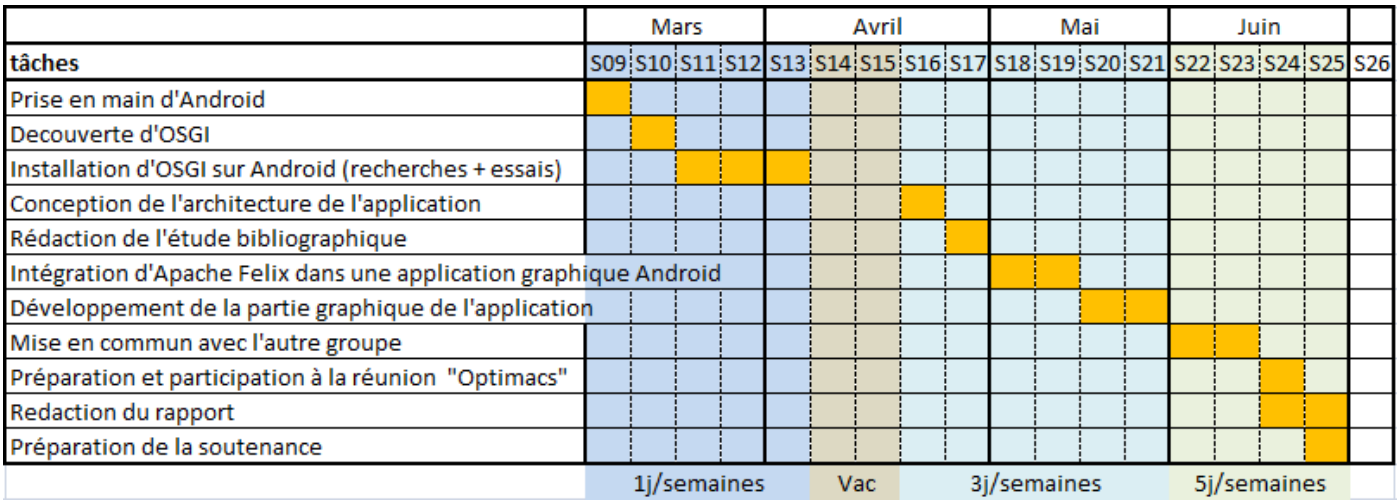


Figure 13: Diagramme de Gant de notre Projet.

Les technologies Android et OSGI étant totalement nouvelles pour nous, il nous a fallu un certain temps de découverte et de prise en main. C'est pourquoi nous avons commencé à travailler réellement sur notre application seulement fin avril, pour définir les grandes lignes de l'architecture que nous souhaitions réaliser.

En mai, nous avions plus de temps pour le PFE et avons donc pu commencer l'implémentation. Il fut assez ardu de mettre en place un serveur Apache Felix permettant d'utiliser les fonctions graphiques d'Android. Une fois cela réalisé, nous avons pu commencer à développer notre application.

Dès que le travail de l'autre groupe fût suffisamment avancé, nous avons procédé à une mise en commun de manière à ce que notre application réponde à leur implémentation d'environnement pervasif.

Durant la phase de développement nous faisions un point hebdomadaire d'avancement avec notre tutrice ainsi qu'avec l'autre groupe sur la fin lors de la mise en commun.

3.3. Avis sur Android

Android est réellement un système d'exploitation d'avenir, pour les utilisateurs, comme pour les développeurs. Le système est bien pensé, et il est facile et rapide de développer pour cette plateforme, notamment grâce à la possibilité d'implémenter la partie graphique dans un langage XML.

Cependant, les choses se gâtent lorsque notre application sort des chantiers battus. La création graphique est quasi-dépendante d'une description en XML avant la compilation. Il est vrai que lors de nos premiers pas sur la plateforme, nous avons trouvé ce système simple et puissant. Mais afin de développer une interface totalement adaptative, nous avons dus faire l'impasse sur l'utilisation du XML. Cela fut extrêmement contraignant tout d'abord d'un point de vue réalisation, puisque toutes les options ne sont pas accessibles en java, mais aussi d'un point de vue développement : tous les exemples et tutoriels utilises le XML. Nous avons passé du temps à chercher « la fonction java qui fait à peu près la même chose que le XML ».

De plus, Android est encore jeune, et connait donc quelques problèmes de jeunesse. Certaines fonctions ne sont pas implémentées, et pas mal de limitations que nous espérons temporaire : arbre des vues limités, impossibilité de supprimer un seul onglet… Si bien que l'on se demande parfois s'il n'est pas trop tôt pour coder sur Android.

Cependant, malgré ces quelques défauts, nous avons tout de même réussi à développer une interface adaptative fonctionnant sur un serveur OSGI porté sur Android. C'est bien la preuve que L'OS est hautement flexible.

Bien qu'Apache Felix n'ait à la base pas été conçu pour Android nous avons prouvé que ces 2 technologie restent compatible entre elles au prix de quelques « tours de passe-passe ». Durant notre projet ces 2 technologies ont évolué (nous avons commencé avec Felix 1.4 et Android 1.1, aujourd'hui Felix est à la version 1.8 et Android 1.5). Il semble que la compatibilité se soit améliorée[5] avec ces dernières versions. De plus, la communauté des développeurs commence à s'intéressé au mariage Felix/Android. Ezdroid (http://www.ezdroid.com) est un projet récent utilisant Apache Felix et Android et peut être une source d'inspiration pour l'avenir de ce « couple ».

[5] http://blog.luminis.nl/roller/luminis/entry/osgi_on_google_android_using (lire UPDATE en haut de la page)

3.4. Possibilités d'amélioration

Notre application en l'état est fonctionnelle et permet d'utiliser pleinement les services développés par nos collègues. Nous avons pensé à quelques améliorations possibles, que nous n'avons pas pu développer faute de temps :

Au niveau de la présentation, il serait intéressant d'avoir une plus grande adaptabilité aux préférences utilisateurs. Comme nous l'avons dit précédemment nous avons été un peu bloqués dans cette direction car Android ne peut pas pour l'instant supporter des arbres de Views trop grands. Il faudrait optimiser au maximum l'utilisation des views et peut être séparée l'application en plusieurs Activity afin de pouvoir présenter l'interface utilisateur de manière plus complexe et compacte.

Notre service chargé de la découverte des DataSource s'actualise lorsque l'utilisateur clique sur le bouton update. Il serait par la suite intéressant que la liste s'actualise automatiquement, soit par un système d'abonnement, soit en envoyant une requête à intervalles de temps régulier. L'utilisateur pourrait alors être averti de l'arrivé de nouveaux services dans l'environnement, ou bien que le service qu'il utilise n'est plus à porté.

Nous avons choisi une architecture OSGI car cela permet de charger des classes à chaud et nous semblait intéressant pour la conception d'une interface adaptative. Nous avons alors construit notre application autour d'un serveur Apache Felix dont la compatibilité à Android est désormais prouvée. Nous pouvons alors envisager une utilisation plus poussée des possibilités d'OSGI. Par exemple un DataSource complexe et nécessitant une interface plus spécifique pourrait mettre à disposition un bundle OSGI que le client téléchargerait et intègrerait dans son interface. Ainsi ce n'est plus le client qui génère l'interface à partir de la description du DataSource mais le DataSource qui embarque son interface. L'intérêt ici d'OSGI est de pouvoir charger l'interface du service sans devoir redémarrer l'application.

Les services ne proposent à l'heure actuelle que des commandes, c'est-à-dire une fonction qui à partir de x inputs renvoie y outputs. Il reste à implémenter aussi bien coté client que coté environnement l'environnement une notion de flux. Par exemple au lieu de devoir appeler la commande « getTime » pour recevoir l'heure à un instant donné, le client devrait pouvoir s'abonner à un flux Time qui se met à jour toutes les secondes par exemple.

Enfin une amélioration intéressante aussi bien du coté de l'environnement pervasif que du client serait l'utilisation de variables de tous types (et pas seulement string comme dans notre implémentation) afin de proposer à l'utilisateur une interface plus ergonomique.

Lexique

Activity	Classe principale d'une application Android.
Box	Objet comportant la vue d'un DataSource et des fonctions de manipulation de son affichage
BoxFactory	Service OSGI fabricant des Box
Bundle	Composant d'une application OSGI
CaptorInterface	Bundle permettant de créer une interface en fonction des paramètres logiques renvoyés par un DataSource
Commande	Dans l'environnement pervasif, les DataSource proposent plusieurs commandes à l'utilisateur.
DalvikVM	Nom de la machine virtuelle spécifique utilisée dans Android
Datasource	Service de l'environnement pervasif. Il peut s'agir d'un capteur, d'une fonction...
DS container	Serveur contenant un ou plusieurs DataSource
EditText	Widget Android. Boite de texte éditable
Input	Paramètre d'entré d'une commande
IPOJO	Modèle de composant au dessus d'OSGI plus flexible que celui-ci. IPOJO = *injected Plain Old Java Object*
Layout	ViewGroup servant a disposer des View. On a entre autre des LinearLayout (horizontaux et verticaux), TableLayout…
Output	Paramètre de sortie d'une commande
Service	Au sens OSGI, un service est une instance de classe qui peut être retrouvée et utilisée par les autres composants grâce aux fonctions OSGI.
Service Directory	Annuaire de service dans l'environnement pervasif
ServiceListener	Interface OSGI. Il permet d'effectuer une action en cas de modification d'un service (création, suppression, modification).
ServiceTracker	Objet permettant de traquer l'apparition de services dans l'environnement OSGI.
TextView	Widget Android. Boite de texte non éditable
View	Base de tous les éléments graphiques d'Android. Layout et widgets extend la classe View.

Viewfactory	type de bundle permettant de créer une vue en la renvoyant à l'application Android
Widget	Au sens Android, nom donnée aux View permettant une interaction avec l'utilisateur.

Annexe 1 : Installation et utilisation d'Apache Felix sur Android en utilisant le shell

Il est possible de connecter un shell Android à notre émulateur grâce à la commande :

```
adb shell
```

Une fois connecté il suffit de se créer un répertoire de travail (/data/felix par exemple) dans lequel nous copieront les fichiers suivants:

./bin/felix.jar	→archive jar du serveur Apache Felix
./conf/config.properties	→configuration standard de Felix lançant les bundles ci dessous au démarrage du serveur **(mettre en annexe).**
./bundle/org.apache.felix.shell-1.0.2.jar	→Shell felix
./bundle/org.apache.felix.shell.tui-1.0.2.jar	→Console pour le shell de felix

Pour copier les fichiers, on utilise la commande adb push:

```
adb push c:\[chemin en local]\felix.jar /data/felix/bin/felix.jar
```

ATTENTION : Tous les jars utilisés sous android doivent être adaptés pour pouvoir tournés sur la DalvikVM. Pour cela google fournit un utilitaire dx permettant de convertir les class au format dalvik :

```
dx --dex –output=[chemin complet]/classes.dex [chemin
complet]JAR_file.jar
cd [chemin complet]
aapt add JAR_file.jar classes.dex
```

ATTENTION : Il est important de se mettre dans le dossier de travail pour utiliser la commande aapt add, sinon classes.dex ne sera pas ajouté correctement à l'archive (l'arborescence complète sera insérée dans l'archive) et elle sera inutilisable pour la DalvikVM.

Ensuite nous pouvons lancer notre serveur apache felix:

```
adb shell → pour lancer un shell
cd /data/felix
```

La commande java n'existe pas sous android, on utilise la commande suivante :

```
/system/bin/DalvikVM -Xbootclasspath:/system/framework/core.jar \
-classpath bin/felix.jar org.apache.felix.main.Main
```

Nous avons alors un serveur felix OSGI et la possibilité de charger d'autres bundles.