



## Spécification Tuxdroid Gadget API

Le 1 mai 2008 à 17:32.

Ce document a pour objectif de définir le fonctionnement des Gadgets TuxDroid et du conteneur associé. Il a valeur de loi, et nul n'est sensé ignorer la loi 😊

### Licence

L'ensemble des développements est sous Licence GPL v2. Chaque fichier source dispose d'un en-tête décrivant sommairement la licence, le nom de l'auteur et les droits à la société Kysoh.

### Conteneur de Gadgets

#### Organisation générale

Le système proposé suit le schéma d'un framework. C'est à dire que les gadgets sont insérés dans le flot de traitement d'un ensemble plus large qui en fixe les règles de gestion, contrairement à une API ou une librairie qui s'insère dans le flot de traitement de l'appelant.

#### Aspect conteneur

L'architecture du Tuxdroid Gadget Framework est centré autour d'un conteneur. Cette philosophie est emprunté aux serveurs d'application, de même que le système de déploiement/retrait des gadgets du système. L'idée en est que le framework maintient en permanence la liste des gadgets qu'il connaît, de leur état, de leurs propriétés, etc. Le conteneur est le médiateur obligatoire entre l'application tiers et les gadgets. C'est lui qui les instancie, les exécute, les arrête, les détruit, etc.

#### Auto déploiement

Le conteneur de gadget observe en permanence un dossier spécial dit dossier d'auto déploiement. Lorsqu'il y détecte l'arrivée d'une nouvelle archive, il la décompresse dans un autre dossier dit dossier de travail. Cette phase est appelée déploiement.

A chaque lancement du conteneur, les gadgets sont redéployés. Si un gadget est détruit, le dossier correspondant l'est lui aussi. Enfin, si au démarrage un dossier n'a pas d'archive associée, il est automatiquement retiré.

#### Listeners

Le conteneur est un objet Java produisant un certain nombre de notifications à ceux qui s'y abonnent

```
void gadgetContainerError(Throwable throwable), si une erreur a été détecté par exemple en phase de déploiement.  
void gadgetLoaded(Gadget gadget), si un gadget a été chargé (voir le cycle de vie d'un gadget).  
void gadgetUnloaded(Gadget gadget), si le gadget a été déchargé, par exemple s'il a été supprimé.  
void deploymentTerminated(), si le déploiement d'un gadget vient juste d'être terminé.
```

#### Mise en oeuvre

La mise en oeuvre du conteneur est très simple

```
// Création du conteneur  
GadgetContainer container = new GadgetContainer();  
  
// Ajout du listener sur le conteneur  
container.addListener(new GadgetsContainerListener() {  
  
    @Override  
    public void gadgetUnloaded(Gadget gadget) {  
        // Appelé lorsqu'un gadget a été retiré du conteneur  
    }  
  
    @Override  
    public void gadgetLoaded(Gadget gadget) {  
        // Appelé lorsqu'un gadget a été ajouté (déployé) au conteneur  
    }  
  
    @Override
```

```

    public void gadgetContainerError(Throwable [1] throwable) {
        Appelé lorsqu'un exception c'est produite lors du processus de déploiement ou
d'enregistrement

    }

    @Override
    public void deploymentTerminated() {
        Appelé quant le déploiement initial est terminé.
    }
});

// Ajout d'un organe de déploiement
container.addAutoDeployer("/dossier_de_travail", "/dossier_de_gadgets");

```

## Définition du gadget

### Vue générale

Un gadget est programme écrit dans n'importe quel langage qui lors de son exécution communique avec le conteneur à l'aide d'un protocole normalisé. Comme nous l'avons vu plus haut, une fois le gadget déployé dans le conteneur, il est soit utilisable à travers une instance de la class `Gadget` . Cette instance peut soit être récupérée de manière asynchrone via le listener du conteneur, soit comme suit :

```

for (Gadget gadget:container.getGadgets()) {
    ...
}

```

### Format du gadget

Un Gadget est une archive au format ZIP portant l'extension `.tgf`. Son contenu est totalement libre. Le langage utilisé pour rédiger un gadget est lui aussi absolument libre (C, en Python, en Java, en Bash, etc.).

A la racine de l'archive se trouve fichier XML, nommé `gadgets.xml` donnant :

- La description du gadget
- Son mode d'exécution
- Ses données
- Ses commandes
- Ses notifications

Voici un exemple pour un tel fichier :

```

<gadgets>
<gadget>
  <interpreter
    kind="java">
    <executable>net.karmaLab.tuxDroid.gadgets.WeatherGadget</executable>
  </interpreter>
  <description>
    <name>Weather Gadget</name>
    <description>Google Weather Gadget</description>
    <author>Y. Brault (C) http://artisan.karma-lab.net 2008</author>
    <version>1.0</version>
    <iconFile>resources/WeatherGadget.png</iconFile>
    <uuid>f63af23e-7ae0-4389-b89b-bc5a8185b0b8</uuid>
  </description>
  <parameters>
    <parameter
      name="location"
      description="Weather location"
      type="string"
      defaultValue="paris" />
    <parameter
      name="unit"
      description="Temperature unit (celcius, fareneight)"
      type="enum(celcius, fareneight)"

```

```

        defaultValue="celcius" />
<parameter
  name="tomorrow"
  description="Give tomorrow weather too"
  type="boolean"
  defaultValue="false" />
</parameters>

<commands>
  <command name="play" description="play"/>
  <command name="stop" description="stop"/>
</commands>

<notifications>
  <command name="test" description="Test notification"/>
</commands>
</gadget>
</gadgets>

```

Comme vous le voyez, ce fichier peut définir plusieurs gadgets contenus dans la même archive.

La description d'un gadget n'est JAMAIS à lire à travers le fichier XML. En fait toute application utilisant un gadget doit passer impérativement par son instance de la classe `Gadget` .

### Interpreteur

La section `interpreter` définit :

#### *L'interpréteur*

sh, perl, java, python, etc..

#### *L'URI de la ressource*

./mon\_gadhet.sh

Notez que certains interpréteurs font plus de choses que d'autre. Par exemple en mode `java` , le dossier `./librairies` est automatiquement inclus dans le classpath.

## Identification

Le gadget fournit aussi des données fondamentales :

- Son nom
- Sa description
- Son auteur
- Sa version
- Son UUID, qui représente un ID unique pour ce gadget

Chaque commande exécuté doit rendre la main le plus rapidement possible.

Pour accéder à la description d'un gadget, il faut passer par son objet `Gadget` , par exemple :

```

// récupération de la description du gadget
gadget.getDescription().getDescription();

```

La description et le fichier d'aide (`gadget.getHelpFile()`) sont automatiquement traduits par le framework.

### Internationalisation

Les gadgets utilisent le système `gettext` standard dans le monde de l'openSource (utilisé par Drupal, Linux, Gnome, etc.).

En fait le concept est ultra simple. L'idée est que toutes les chaînes à traduire sont passées par une fonction unique qui maintient les traductions en mémoire.

Pour du code Java ça donne quelque chose comme cela :

```

// Chargement du moteur de traduction
I18n i18n= new I18n("./resources");

// Traduction d'un message

```

```
| System [2].out.println (i18n.tr("This is a message with one parameter : %d", 1));
```

La fonction `i18n.tr(...)` va chercher dans son dictionnaire la traduction de `This is a message with one parameter : %d`. S'il la trouve, il renvoie sa valeur, sinon il renvoie la chaîne d'origine.

Il est donc possible de voir ce qui n'est pas encore traduit en basculant par exemple en français et en localisant toutes les chaînes encore en anglais. L'avantage c'est que si la traduction n'a pas encore été faite, les chaînes restent lisibles pour l'utilisateur ET dans le code.

Les dictionnaires sont de simples fichiers textes que l'objet `I18n` va chercher dans le dossier qui est passé en paramètre de son constructeur (`./resources`). Pour ce faire, il se base sur la fonction `java Locale.getDefault().getLanguage()` qui renvoie la langue courante du système.

Pour un système français cela renvoie "fr". L'objet `i18n` va donc chercher un fichier appelé `./resources/fr.po`.

Ce fichier `fr.po` contient une série de traductions sous la forme :

```
| msgid " This is a message with one parameter : %d"
| msgstr "Ceci est un message avec paramètre %d"
|
| msgid " Another message"
| msgstr "Un autre message"
```

Ce sont ces traductions que l'objet `I18n` va charger en mémoire et que la fonction `i18n.tr` va utiliser.

Pour les fichiers XML, il n'y a évidemment pas de fonction `i18n.tr` directement dans le fichier. C'est le framework qui avant de renvoyer une chaîne provenant du fichier `gadgets.xml`, par exemple au `ControlCenter`, va la faire passer dans `i18n.tr`.

De la même manière le framework va passer à la fonction `i18n.tr` toutes les notifications de gadget de type `message`. Du coup, le code gadget suivant sera automatiquement traduit :

```
| throwMessage("This is a message with one parameter : %d", 1);
```

Le `Control Center` recevra cette notification déjà traduite donc, sauf si bien sur cette chaîne n'est pas dans le fichier `.po` de la langue courante ou que la langue courante n'a pas encore de fichier `.po`.

Le format des fichiers `.po` est très évolué, il permet de chercher des chaînes simples, des chaînes "floues", des chaînes avec pluriels, etc. Pour l'instant, la lecture des fichiers `.po` se fait dans `karmalab-commons` (la librairie d'utilitaires utilisé par le framework) avec une version très simple et très rapide d'un parser de `.po`.

Si les besoins évoluent vers plus de complexité, on fera évoluer cet objet pareillement.

En somme, les avantages l'approche `.po` par rapport aux classique `Bundles` de ressources Java sont :

- Le code reste lisible de manière naturelle.

- Même si des chaînes ne sont pas traduites, la lecture de l'interface reste possible.

- Les fichiers `.po` ne sont pas inclus dans les `.jar`, mais sont des fichiers que chacun peut lire, modifier, étendre. Cela facilite les contributions par **de purs non-informaticiens**.

- Il existe de nombreux outils permettant d'éditer des `.po` : <http://drupal.org/node/11131>

Pour l'aspect développeur, La génération de `.po` est elle aussi assez simple, y compris à partir du code Java.

La première étape consiste à générer un fichier `.pot` (Portable Object Template) qui n'est rien d'autre qu'un `.po` dont tous les "msgstr" sont vides. Cela se fait par la commande suivante

```
| xgettext -k_ -o po/messages.pot src/Translatable.java
```

Ensuite on duplique le `.pot` en, par exemple, `fr.po` pour rédiger une traduction ou envoyer les fichiers à un traducteur.

Ensuite pour la mise à jour, quand le code source évolue et donc que de nouveaux messages apparaissent, ou que d'anciens messages disparaissent, il suffit de régénérer le `.pot` et d'appliquer cette nouvelle mouture aux fichiers `.po` existants :

```
| msgmerge -U po/fr.po po/messages.pot
```

Concernant le fichier d'aide du gadget, il est lui aussi traductible mais de manière un peu différente. L'idée est que par défaut le gadget doit avoir un fichier `./resources/help.html` et optionnellement plusieurs fichiers `./resources/help_xx.html`. De la même manière lorsque l'hôte demande la référence au fichier d'aide du gadget, celui-ci lui renvoie la version traduite automatiquement si elle existe, la version anglaise le cas échéant ou `null` si aucun fichier n'est présente.

La variable `tgp_language`, fournit au gadget le langage courant à utiliser lors des traductions (voir paramètres du gadget plus bas).

A travers les sections `commands` et `notifications`, le gadget indique au conteneur les commandes qu'il prend en charge et les notifications qui peut générer. Le lancement de ces commandes et la prise en charge des notifications est à la charge du Control Center.

Dans les deux cas, chaque commande/notification est caractérisée par

- Un ID
- Une description

Pour accéder à ces données, il faut passer par son objet `Gadget`. Dans les deux cas l'objet retourné est de classe `GadgetToken`, par exemple :

```
// récupération de l'id de la première commande
gadget.getCommands().get(0).getId()

// récupération de l'id de la première notification
gadget.getNotification().get(0).getId()
```

La description est automatiquement traduite par le framework.

### Données du Gadget

Dans la section `parameters`, le Gadget fournit l'ensemble des données dont il a besoin pour fonctionner. Cela correspond à une liste où chaque élément est composé comme suit :

- Nom, le nom de la donnée.
- Description, la description (en anglais !!) de la donnée.
- Type, le type de la donnée pouvant être : `string`, `integer`, `double`, `boolean`, `enum(litem1,item2...,itemn)`
- `defaultValue`, la valeur par défaut de la donnée.

La description et la valeur par défaut sont automatiquement traduites par le framework.

La modification et la sauvegarde des données est à la charge du Control Center. La création d'un jeu de donnée pour un Gadget va nous permettre d'instancier le Gadget.

Pour accéder à ces données, il faut passer par son objet `Gadget` et récupérer l'objet `GadgetParameter` correspondant, par exemple :

```
for (GadgetParameter parameter gadget.getParameters()) {
    parameter.getKind();
    parameter.getDescription();
    ...
}
```

### Gadget et Instance de Gadget

Un gadget dans l'état une fois connu du conteneur est dans un état "chargé" et n'est pas instancié en mémoire (cad exécuté). Seule sa description (cf `gadgets.xml`) est connue du conteneur et des applications tiers.

Pour exécuter une commande du Gadget, il convient donc de transformer le Gadget en Instance de Gadget que le Control Center va ensuite pouvoir exécuter et même stopper en cours d'exécution si besoin était.

A ce titre on comprend qu'une instance est composée :

- D'une référence au gadget "chargé".
- D'un jeu de valeurs correspondant à ses paramètres d'exécution.
- Du nom de la commande à exécuter.

La conséquence de cette approche est qu'il n'est pas besoin de faire des gadgets à la configuration complexes mais d'utiliser les instances pour réduire cette complexité. Prenons l'exemple d'un Gadget lisant les flux RSS. Il a pour données l'URL à lire et comme commande `checkRSS`. Pour effectuer le traitement, il va falloir créer une instance du Gadget RSS, prenant en paramètre la commande à exécuter ET les valeurs des paramètres de l'instance. On en déduit que si l'on désire lire plusieurs flux RSS, il suffira de créer plusieurs instances associées chacune à un jeu de données unique.

Pour créer une instance de gadget à travers le framework, la syntaxe est la suivante

```
GadgetInstance instance = gadget.create();
```

Ensuite une commande est invocable de la manière suivante :

```
instance.start(command, parameters);
```

Si `command` est à `null`, c'est la commande par défaut du gadget qui est utilisée, sinon il s'agit d'un objet `GadgetToken` que l'on récupère par `gadget.getCommands().get(XXX)`. Les paramètres sont une liste de `GadgetParameter` contenant les valeurs des paramètres pour cette exécution de commande.

### Protocole de communication

---

La communication entre le gadget et l'hôte, doit être suffisamment simple pour permettre la rédaction dans tous les langages. L'idée de base est que les commandes sont passées en argument, les données dans les variables d'environnement, et les notifications sont renvoyés sur le canal `StdOut`.

Les données dans les variables d'environnement sont préfixées avec la chaîne de caractère `tgp_`. Si le gadget déclare une données `message`, il recevra donc sa valeur dans la variable `$tgp_message`.

La variable `tgp_version`, fournit au gadget la version du protocole. ex. `1.0`. Charge au gadget de correctement prendre en charge la version du protocole.

Lorsqu'un gadget est exécuté sans paramètres, il renvoie sa description sous la forme d'un flux XML. Ensuite exécuté avec en paramètre un ID de commande, il exécute la commande correspondante.

Les notifications sont émises sur le canal `stdOut` sous la forme :

```
ID "PARAM1" "PARAM2" ... "PARAM2"
```

Les paramètres sont des chaînes qui sont échappées sur le caractère `"`.

Il existe un certain nombre de d'ID de notification standard que le gadget n'a pas besoin de déclarer pour les utiliser :

```
message chaîne param1 param2 ... paramN , indique à l'hôte que ce qui suit est un message en anglais, de la forme
message {0} suite {1} . Les paramètres sont neutres du point de vue de la langue ce qui fait que le message peut être
traduit sans risque. Le fait de transformer ce message en parole par le Tux est à la charge du Control Center.
trace chaîne param1 param2 ... paramN , fonctionne comme message mais fournit une trace d'exécution.
error chaîne param1 param2 ... paramN , fonctionne comme message mais fournit une erreur d'exécution.
```

## Questions/réponses

---

### Comment afficher plus d'information lors d'une erreur sur un gadget

---

Si une erreur se produit, le message de l'exception est remonté par le framework sous la forme d'une notification "error". En activant le mode "traces" du gadget (propriété `traces` à `true`), c'est la trace complète qui est remontée avec numéros de lignes & co. Pour l'activer, la solution est peu simple pour l'instant, il faut passer la valeur à `true` à la ligne

### Pourquoi ne pas faire parler et utiliser la télécommande du TUX directement dans un gadget ?

---

Comme base à ce framework, il y a deux constats induits par les précédentes générations :

1. 70% des actions physiques sur le Tux faites par les gadgets est de le faire parler.
2. Les 20% restant correspondant à prendre en charge la télécommande, les boutons du Tux et les capteurs pour effectuer des actions sur les tuxlets.

Du coup pour simplifier le développement des gadgets, il semble pertinent de remonter ces deux aspects sous la forme de services offerts aux gadgets évitant ainsi un code redondant :

Sous la forme d'une fonction qui fait parler le Tux (`throwMessage`) qui utiliserait toujours la même voix, le même timbre et la même langue.

En mettant en place dans le manager un système permettant d'associer UN évènement (de type "Bouton X de la télécommande", "Niveau Y de la lumière", etc) à UNE commande pour UNE instance d'UN gadget.

Ainsi le Gadget fournit des commandes, qui sont associées, PAR le Control Center, à UN évènement. L'utilisateur peut donc décider de relier par exemple le fait de taper sur la tête du Tux pour lire ses eMails sans que moi, développeur du gadget, n'ai eu une seule ligne de code à écrire pour que cela soit rendu possible. Tout est à la charge du Control Center.

En somme, rien n'empêche le gadget de contrôler lui-même le Tux, mais en faisant cela de manière systématique, on s'expose 1/ à un code plus complexe 2/ à des conflits entre les différents gadgets sur une ressource physique donnée (ex. deux gadgets utilisent le même bouton XX de la télécommande)

D'un point de vue ergonomique, l'idée est en trois/quatre click d'associer une commande à un évènement qui peut être :

- Une alarme temporelle (ex. tout les N minutes, à 10h chaque jour, etc) en utilisant la librairie Quartz
- Un bouton du Tux (ex. tête, aile, etc..)
- Un bouton de la télécommande
- Un seuil de capteur lumineux
- Un seuil de capteur sonore
- Une notification venant d'un autre gadget

...

Avec un tel système il est possible grâce au control center, et sans qu'un code soit inclus dans le gadget lui-même, d'opérer par exemple une lecture des mails lorsque la lumière du jour a suffisamment baissé. Il s'agit juste d'un exemple, mais cela permet de comprendre la, je pense, puissance de l'approche "par évènement" à contrario de l'approche "par code gadget".

Un exemple plus crédible du même ordre serait un gadget qui allume les yeux du Tux que l'utilisateur peut décider d'associer à l'arrivée de la nuit et un gadget qui fait simplement dire un message au tux qui lui fait faire "Cocorico" le matin 😊

En somme l'approche par évènements simplifie les gadgets et les rends beaucoup plus souple, et le fait de laisser le control center faire parler le tux par le biais de notification permet de centraliser proprement le paramétrage des voix sans avoir à envoyer à chaque instance de gadget un paramétrage de voix juste pour qu'il sache correctement parler. De même le gadget n'a pas à gérer de configuration complexes de boutons, horloge, etc, car tout est factorisé dans le Control Center.

#### Pourquoi les librairies du Control Center/Gadget Tester ne sont pas héritées par le Gadget ?

---

Dans le cas des gadgets écrits en Java, il faut bien comprendre une chose sur le mode de lancement des gadgets. Ils sont exécutés dans **un processus différent de l'application hôte** (Control Center ou Gadget Tester). Cela implique dans le cas d'un gadget en Java qu'il ne partage pas la même machine virtuelle que le contrôle Center. L'avantage est que si le gadget plante, cela n'affecte en rien le Control Center. La contrepartie est que le gadget démarrant sur une machine virtuelle vierge, n'hérite pas des dépendances qu'utilise le Control Center. Cela implique aussi qu'un Gadget peut utiliser des librairies de version différente du CC.

Ainsi, un gadget écrit en BASH qui dit "Hello World" ressemblera seulement à cela :

```
#!/bin/sh
echo message Hello World !! : $tgp_message
```

Ce que l'on comprend du coup, c'est qu'un gadget n'a, a priori, besoin de rien d'autre qu'un langage capable de lire les variables d'environnement (paramètres du gadget) et d'écrire dans la sortie standard (les notifications).

Dans le cas des gadgets en Java, j'ai écrit une classe appelée SimpleGadget qui va simplifier la fabrication, en java, de gadget en fournissant des fonctions qui vont un peu cacher ce fonctionnement de base variables/écriture dans la sortie standard. Du coup la fonction "throwMessage" n'est rien d'autre qu'un simple :

```
System.out.println(message)
```

Là où un malentendu est apparu est que cette classe SimpleGadget est dans le .Jar du framework. C'est pour cela qu'il faut inclure le framework dans les gadgets pour qu'ils puissent fonctionner. Mais le gadget n'utilise RIEN D'AUTRE dans ce .Jar QUE la classe SimpleGadget. Au fond, il serait peut-être moins déroutant de mettre cette classe toute seule dans un .jar du genre "SimpleGadget.Jar". Cela générerait moins de confusion.

#### Pourquoi le packaging via ANT plante sous Windows ?

---

Maintenant pour ce qui est de du packaging via ANT, j'ai déjà donné deux fois il me semble la résolution de ce problème lié à Windows. C'est même à la suite de cela que j'ai ajout le truc sur mon site à la fin de [cette page](#) <sup>[3]</sup>

#### Pourquoi utiliser UTF-8 sous Windows plutôt qu'ISO-8859-15 ?

---

Déjà l'utilisation d'UTF8 est un pré requis effectivement pour la librairie des PO. Je vous conseille d'ailleurs vivement de ne pas laisser eclipse dans son mode standard sous windows consistant à utiliser ISO-8859-15. Car dès que vous allez avoir à utiliser des langues exotiques, vous allez être dans le "caca".

UTF-8 est aujourd'hui une norme absolue et universellement utilisée. Même Windows n'utilise plus que cela en interne. Le fait qu'eclipse utilise ISO par défaut est juste un effet de bord malheureux du à ce que Java par défaut sous Windows utilise ISO.

Une fois que vous avez défini UTF-8 comme étant l'encodage par défaut dans eclipse (préférences du Workspace), il faut aussi que vous traduisiez tous vos sources avec un utilitaire comme uconv. Je sais, c'est un peu casse pied, mais obligatoire sur un gros projet. Sinon, je vous prédit de gros soucis lorsque vous devrez intégrer des contributions de la communauté.

Il faut donc impérativement basculer les sources en UTF8 car sinon on aura des gros problèmes lors de l'intégration de contributions. Pour cela changer la propriété "Text file encoding" dans eclipse/windows/preferences/general/workspace ) UTF-8 et utiliser un utilitaire comme "iconv" pour ré-encoder les sources dans un format lisible par tous

#### Pourquoi n'y a-t-il pas de en.po ?

---

Les ID des chaînes sont pas norme en anglais, le en.po est donc inutile car si l'on ne trouve pas de traduction, c'est les ID qui sont utilisés. En revanche, il est de bon ton de mettre dans le dossier ./resources un gadgets.pot (PO Template) permettant de créer facilement une traduction.

#### Pourquoi ne pas coller les .Jar des sous-projets TUX dans les dépendances d'un gadget ?

---

L'idée de base est de ne pas ajouter d'autres bibliothèques dans le projet eclipse que celles dont on ne dispose pas des sources (ex. javamail). Les bibliothèques dont les sources sont disponibles via tuxisalive ou karma-lab.net doivent être checkouté à partir des dépôts et liés dans l'onglet "projects".

## Changements sur le framework

---

### Gadget-Java-Kit

---

Pour éviter les confusions, une nouvelle bibliothèque a été créée : Tuxdroid-Gadget-JavaKit. Elle permet de construire plus rapidement des gadgets en java et ne dépend d'AUCUNE AUTRE bibliothèque. Ainsi les gadgets en java ne dépendent plus de Gadget Framework.

Je JavaKit est maintenant totalement standalone et ne dépend d'aucune bibliothèques. Du coup, un gadget minimal ne fait plus que 40ko. Autre améliorations :

- plus besoin de getter/setter dans les configuration, un champ, même privé, suffit
- Une fonction readState et writeState permettant d'écrire l'état d'un objet dans le bon dossier de configuration.

### Gadgets.xml

---

gadgets.xml est maintenant localisé dans le dossier ./ressources d'un gadget. C'est plus cohérent ainsi.

### Editeur de propriétés

---

Les paramètres d'un gadgets ont maintenant une propriété "visible" permettant de masquer certains paramètres comme "traces". Par défaut cette propriété est à "true". C'est à usage interne mais il est possible de spécifier un `<visible>false</visible>` dans un gadgets.xml.

### Locuteur & co

---

Tant que l'implémentation de throwMessage n'est pas finalisée dans le Control Center (pile de textes), on garde locuteur, coudry & co dans le conteneur. Cependant, dans la mesure où tous les gadgets ont le même paramétrage, les données ont été déplacées de l'interpréteur au GadgetsContainer (méthode setLocale). Cet appel ne change pas dynamiquement la langue des gadgets déjà chargés mais celle de ceux qui suivent l'appel.

Les propriétés langages, locutor, country et pitch ont été normalisés en tant que paramètre invisible du gadget. Dans un gadget, pour lire la langue par exemple, il suffit de faire un `configuration().getLanguage()`.

### Gadget Tester

---

Ce dernier a beaucoup changé. Tout d'abord il supporte maintenant le changement de langues effectif pour la gadgets suivant. On peut télécharger un gadget et, via le menu, demander le rechargement.

En revanche, le mode test dit "stand alone" a été viré. Maintenant pour tester dynamiquement un gadget, il faut passer par l'application Gadget Tester en mode "gadget" avec une commande du genre :

```
java \  
-jar Gadget-Tester.jar \  
--mode=gadget \  
--path=../tuxdroid-gadget-clock \  
--classpath=../targets/eclipse:\br/>../tuxdroid-gadget-java-kit/targets/eclipse
```

Dans cet exemple, les dossiers ./targets/eclipse et ../tuxdroid-gadget-java-kit/targets/eclipse, relatifs à ../tuxdroid-gadget-clock sont insérés en tête du classpath, surchargeant ainsi d'éventuelles bibliothèques.

### Traductions

---

Vu les problèmes de Java pour formater les chaînes contenant des %, l'option a été prise de remplacer les %s par le formalisme Java/MessageFormat en utilisant des {0}, {1}, etc. Dans le même esprit, les chaînes contenant des doubles quotes échappées et des simples quotes sont maintenant utilisables sans problèmes.

### ThrowMessage synchrone

---

Pour éviter les catapultage de message dans la pile TTS, une file de notifications synchrones a été ajoutée au Conteneur de Gadgets. L'idée est que les notifications de type "message" continue d'être postées par les gadgets en asynchrone, mais que le conteneur les place dans une file qu'il dépile dans un thread en le notifiant le suivant que si le précédent a été traité par l'application hôte.

- [1] [http://www.google.com/search?hl=en&q=allinurl:Throwable java.sun.com&btnl='m Feeling Lucky](http://www.google.com/search?hl=en&q=allinurl:Throwable+java.sun.com&btnl='m+Feeling+Lucky)
- [2] [http://www.google.com/search?hl=en&q=allinurl:System java.sun.com&btnl='m Feeling Lucky](http://www.google.com/search?hl=en&q=allinurl:System+java.sun.com&btnl='m+Feeling+Lucky)
- [3] <http://artisan.karma-lab.net/node/1156>