Notice de fonctionnement du gestionnaire Kassia

Auteur : Augustin Delale
Code : GUS05E001FR
Date : 09/05/2011

Ce document constitue la notice d'utilisation du gestionnaire Kassia et pourra servir de document de référence aux développeurs Java souhaitant utiliser ce composant dans leur programmation gus05. Plutôt qu'un simple pense-bête listant l'ensemble des caractéristiques et fonctionnalités du gestionnaire, j'ai voulu détailler son fonctionnement en profondeur en me référent au besoin à son code source au risque de paraître un peu verbeux de temps en temps.

Version prise en compte : kassia_20110507

Table des matières

1 Les deux tâches fondamentales du gestionnaire

Quel est le rôle du gestionnaire ? Quels conditions doit-il satisfaire pour être désigné comme tel ? Sur un plan général d'ordre fonctionnel, il doit pouvoir manipuler pleinement n'importe quelle entité en lui donnant les ressources nécessaires pour fonctionner et en la faisant interagir avec d'autres entités dans le but d'obtenir une application... Tout cela reste assez flou et ne permet pas de dégager des spécifications techniques précises.

Sur un plan pratique, un gestionnaire doit au moins accomplir deux tâches qui sont :

1. Prendre en charge la méthode de lancement de l'application : *public static void main*
2. Appeler la méthode statique *setManager* de la classe *Outside* en lui passant une instance de l'interface *Manager*.

Nous appellerons ses tâches les deux tâches fondamentales du gestionnaire tout en reconnaissant qu'elles ne sont pas suffisantes pour garantir un gestionnaire pleinement fonctionnel, lequel devra accomplir de nombreuses autres tâches (comme par exemple instancier des entités).

2 La première couche du gestionnaire

Le code source du gestionnaire Kassia se divise en trois couches dont les packages commencent respectivement par :

1. gus05.manager.gus.kassia.layer1
2. gus05.manager.gus.kassia.layer2
3. gus05.manager.gus.kassia.layer3

Il n'est évidemment pas nécessaire de lire et comprendre tout le code source du gestionnaire pour pouvoir l'utiliser mais débuter par une étude des principales classes qui composent la première

couche est une bonne entrée en matière pour avoir un aperçu de sa structure générale. Il s'agit en effet de la couche la plus basse du gestionnaire qui va en quelque sorte transcrire les deux tâches fondamentales en un nouveau contrat à remplir qui aura l'avantage de fixer les grandes lignes de la structure de l'application.

2.1 Les 5 organes du gestionnaire

Ce nouveau contrat défini par la première couche va être : fournir des implémentations pour les 5 interfaces du package *gus05.manager.gus.kassia.layer1.organs*, appelées les 5 organes du gestionnaire.

Voici ci-dessous les 5 interfaces définissant chacunes une unique méthode :

```
package gus05.manager.gus.kassia.layer1.organs;

import gus05.manager.gus.kassia.layer1.Source;

public interface KeyFinder {
    public String findKey(Source source) throws Exception;
}
```

```
package gus05.manager.gus.kassia.layer1.organs;

import gus05.manager.gus.kassia.layer1.Source;

public interface ResourceProvider {
    public Object provideResource(Source source, String key) throws Exception;
}
```

```
package gus05.manager.gus.kassia.layer1.organs;

import gus05.framework.core.Service;
import gus05.manager.gus.kassia.layer1.Source;

public interface ServiceWrapper {
    public Service wrapService(Source source, Object obj) throws Exception;
}
```

```
package gus05.manager.gus.kassia.layer1.organs;

import gus05.framework.core.Entity;

public interface ErrorHandler {
    public void handleError(Entity entity, String id, Exception e);
}
```

```
package gus05.manager.gus.kassia.layer1.organs;

import java.util.Date;

public interface AppLauncher {
```

```
    public void launchApp(Object appObject, Date startDate, String[] args);
}
```

2.2 KassiaManager : implémentation pour l'interface Manager

Pour comprendre les rôles tenus par ces 5 organes, il convient d'étudier la classe *KassiaManager* du package *gus05.manager.gus.kassia.layer1* qui implémente l'interface *Manager* et dont l'instance sera transmise à la classe *Outside*. Son code source est donné ci-dessous :

```
package gus05.manager.gus.kassia.layer1;

import gus05.framework.core.Entity;
import gus05.framework.core.Manager;
import gus05.framework.core.Service;
import gus05.manager.gus.kassia.layer1.organs.ErrorHandler;
import gus05.manager.gus.kassia.layer1.organs.KeyFinder;
import gus05.manager.gus.kassia.layer1.organs.ResourceProvider;
import gus05.manager.gus.kassia.layer1.organs.ServiceWrapper;

public final class KassiaManager implements Manager {

    private KeyFinder kf;
    private ResourceProvider rp;
    private ServiceWrapper sw;
    private ErrorHandler eh;

    public KassiaManager(KeyFinder kf, ResourceProvider rp, ServiceWrapper sw,
ErrorHandler eh)
    {
       this.kf = kf;
       this.rp = rp;
       this.sw = sw;
       this.eh = eh;
    }

    public Service callService(Entity entity, String id) throws Exception
    {
        Source src = new Source(entity,id);
        String key = kf.findKey(src);
        Object obj = rp.provideResource(src,key);
        return sw.wrapService(src,obj);
    }

    public Object callResource(Entity entity, String id) throws Exception
    {
        Source src = new Source(entity,id);
        String key = kf.findKey(src);
        return rp.provideResource(src,key);
    }

    public void sendError(Entity entity, String id, Exception e)
    {eh.handleError(entity,id,e);}
}
```

On retrouve les trois méthodes déclarées dans l'interface *Manager*. La méthode *sendError* se

contente de déléguer son implémentation à un objet de type *ErrorHandler* qui est l'un des 5 organes. Quant aux méthodes *callService* et *callResource*, elles font appel aux instances de 3 autres organes : *KeyFinder*, *ResourceProvider* et *ServiceWrapper*. Au final, la classe *KassiaManager* a besoin de 4 des 5 organes dont les instances sont récupérés en paramètres du constructeur.

Concernant la classe *Source* du package *gus05.manager.gus.kassia.layer1,* nous retiendrons pour le moment qu'il s'agit d'un encapsulateur de l'entité appelante et de l'identifiant d'appel transmis à la classe *Outside*.

Le traitement des appels de ressource est sindé en **2 étapes** : Une première étape prise en charge par l'organe *KeyFinder* qui fournit une clé de génération correspondant à la source de l'appel grâce aux règles de mapping, laquelle sera a priori interprétable par le gestionnaire. Une deuxième étape prise en charge par l'organe *ResourceProvider* qui reprend les données précédentes pour founir la ressource désirée par l'entité appelante.

Le traitement des appels de service est similaire au précédent à ceci près qu'une **troisième étape** prise en charge par l'organe *ServiceWrapper* permet d'envelopper la ressource dans un objet de type *Service* qui sera finalement retourné à l'entité appelante.

2.3 KassiaApp : le coeur de la première couche

Nous pouvons maintenant en venir à la classe abstraite *KassiaApp* qui constitue véritablement le contrat proposée par la première couche. En effet, cette classe contient 5 méthodes abstraites renvoyant respectivement les instances des 5 organes, qui devront naturellement être surchargées par autre chose que return null...

```
package gus05.manager.gus.kassia.layer1;

import java.util.Date;

import gus05.framework.core.Outside;
import gus05.manager.gus.kassia.layer1.organs.AppLauncher;
import gus05.manager.gus.kassia.layer1.organs.ErrorHandler;
import gus05.manager.gus.kassia.layer1.organs.KeyFinder;
import gus05.manager.gus.kassia.layer1.organs.ResourceProvider;
import gus05.manager.gus.kassia.layer1.organs.ServiceWrapper;


public abstract class KassiaApp {

    protected KassiaApp(String[] args)
    {
        Date date = new Date();

        KeyFinder kf = buildKeyFinder();
        ResourceProvider rp = buildResourceProvider();
        ServiceWrapper sw = buildServiceWrapper();
        ErrorHandler eh = buildErrorHandler();
        AppLauncher al = buildAppLauncher();
```

```
        Outside.setManager(new KassiaManager(kf,rp,sw,eh));
        al.launchApp(this,date,args);
    }

    protected abstract KeyFinder buildKeyFinder();
    protected abstract ServiceWrapper buildServiceWrapper();
    protected abstract ResourceProvider buildResourceProvider();
    protected abstract ErrorHandler buildErrorHandler();
    protected abstract AppLauncher buildAppLauncher();
}
```

Tout le code source de cette classe se trouve dans son contructeur dont le paramètre d'entré nous laisse deviner qu'il sera appelé directement dans la méthode *public static void main* de notre application. Que fait ce constructeur exactement ? Quatre choses :

1. Il crée un objet Date car c'est toujours intéressant de savoir à quel instant l'application a démarré.
2. Il appelle les 5 méthodes abstraites pour récupérer les instances des 5 organes.
3. Grâce à 4 de ces instances, il crée une instance de *KassiaManager* qu'il transmet directement à la classe *Outside*.
4. Enfin, il appelle l'unique méthode du dernier des 5 organes, *AppLauncher*, qui aura la modeste tâche de dérouler le reste de l'application. On remarquera au passage que cette méthode prend en paramètres les 3 données applicatives pouvant à ce stade être transmises au programme : la date de démarrage, le tableau d'arguments issu de la méthode *main* ainsi que l'objet *this* qualifié d'objet de démarrage de l'application parce que c'est celui qui est instancié par la méthode *main*.

2.4 KassiaMain : point de départ de l'application

Voici enfin le code source de la classe *KassiaMain* du package *gus05.manager.gus.kassia.layer2* qui hérite de *KassiaApp*, qui contient la méthode *public static void main* et qui sert à instancier l'objet de démarrage passant le tableau des arguments comme paramètre.

```
package gus05.manager.gus.kassia.layer2;

import gus05.manager.gus.kassia.layer1.KassiaApp;
import gus05.manager.gus.kassia.layer1.organs.*;
import gus05.manager.gus.kassia.layer2.organs_impl.*;


public class KassiaMain extends KassiaApp {

    public static void main(String[] args)
    {new KassiaMain(args);}

    public KassiaMain(String[] args) {super(args);}
```

```
    protected KeyFinder buildKeyFinder()
    {return new Impl_KeyFinder();}

    protected ServiceWrapper buildServiceWrapper()
    {return new Impl_ServiceWrapper();}

    protected ResourceProvider buildResourceProvider()
    {return new Impl_ResourceProvider();}

    protected ErrorHandler buildErrorHandler()
    {return new Impl_ErrorHandler();}

    protected AppLauncher buildAppLauncher()
    {return new Impl_AppLauncher();}
}
```

Les 5 méthodes devant être surchargées renvoient chacune une instance d'organe, en instanciant les classes d'implémentation du le package *gus05.manager.gus.kassia.layer2.organs_impl*.

Après avoir vu les classes *KassiaApp* et *KassiaMain*, nous validons donc que :

1. La première couche prend en charge les deux tâches fondamentales du gestionnaire
2. La prospection du code source se poursuit obligatoirement par les 5 implémentations d'organes (le nouveau contrat proposé par la première couche à la deuxième couche) :

    1. Impl_KeyFinder
    2. Impl_ServiceWrapper,
    3. Impl_ResourceProvider
    4. Impl_ErrorHandler
    5. Impl_AppLauncher

3 La deuxième couche du gestionnaire

3.1 L'organe AppLauncher

Que dire en premier lieu sur les 5 implémentations d'organes ? Il s'agit de "boites de traitement" sans aucune variable de classe et munies de constructeurs vides. Leurs instanciations respectives ne donnent donc pas lieu à une éxecution de code qu'il faudrait décortiquer. Le seul intérêt de ces organes résident dans l'appel de leurs méthodes.

Nous en venons donc à la méthode *launchApp* de l'organe *AppLauncher* qui est appelée pour dérouler la suite de l'application. Voici ci-dessous le code source de son implémentation :

```
package gus05.manager.gus.kassia.layer2.organs_impl;

import java.lang.Thread.UncaughtExceptionHandler;
```

```
import java.util.Date;

import gus05.manager.gus.kassia.layer1.organs.AppLauncher;
import gus05.manager.gus.kassia.layer2.KassiaSystem;

public class Impl_AppLauncher extends KassiaSystem implements AppLauncher, Runnable,
UncaughtExceptionHandler {

    public void launchApp(Object appObject, Date startDate, String[] args)
    {
       initSystem(appObject,startDate,args);

       Thread.setDefaultUncaughtExceptionHandler(this);
       initThread = new Thread(this,INIT_THREADID);
       initThread.start();
    }

    public void run()
    {startApplication();}

    public void uncaughtException(Thread t, Throwable e)
    {if(t==initThread) fatalError_uncaughtException(e);}
}
```

Nous constatons que la classe *Impl_AppLauncher* est intimement liée à une nouvelle classe du gestionnaire Kassia qui aura une importance particulière : *KassiaSystem* (classe située dans le même package que *KassiaMain*). Cette classe centrale ne possède que des constantes, des variables et des méthodes statiques, et constitue une sorte de colonne vertébrale pour le gestionnaire dont quasiment l'intégralité des classes des deuxième et troisième couches vont hériter (à commencer par les classes d'implémentations des organes).

Le code source de la méthode *launchApp* appelle tout d'abord la méthode statique de *KassiaSystem* : *initSystem* qui initialise le gestionnaire, comme nous allons le voir juste après.

Après quoi, la classe s'enregistre comme *UncaughtExceptionHandler* par défaut au près de la classe *Thread*, dans le but de récupérer et traiter les erreurs qui pourraient causer la fin prématurée du thread de lancement de l'application.

Finalement, le thread de lancement de l'application (correpondant à la variable statique *initThread* de *KassiaSystem*) est crée et démarré, en prenant le nom de Thread : "KASSIA_INIT_THREAD" (constante *INIT_THREADID* de *KassiaSystem*) et en prenant l'objet Runnable "this" correspondant à l'organe lui-même.

Le thread de lancement de l'application, se lance alors et exécute la méthode *run* de la classe *Impl_AppLauncher*, laquelle exécute la méthode statique *startApplication* de *KassiaSystem*. On retiendra de tout cela que notre organe délègue le déroulement de l'application aux méthodes de la classe *KassiaSystem* en deux temps :

1. par l'appel de la méthode *initSystem* (dans le thread main)
2. par l'appel de la méthode *startApplication* (dans le thread "KASSIA_INIT_THREAD")

Que dire de la méthode *uncaughtException* de notre organe ? Si le thread qui est à l'origine de l'exception non gérée est le thread de lancement, alors une autre méthode statique de KassiaSystem est appelée : *fatalError_uncaughtException*, sur laquelle nous reviendrons un peu plus tard.

3.2 Initialisation du gestionnaire

Dans la suite du déroulement de l'application, la méthode *initSystem* est appelée en passant 3 arguments qui sont ceux transmis à l'organe *AppLauncher* par la classe *KassiaApp*. Le rôle de cette méthode va être de créer et remplir une variable statique de type Map appelée *internal* et prévue pour être la racine du stockage de l'ensemble des données et objets internes de l'application. De manière schématique, la méthode initSystem va réaliser les 3 actions suivantes :

1. Mise en place de la structure de stockage (la map *internal* et son contenu)
2. Stockage des 3 données applicatives initiales transmises par l'organe *AppLauncher*
3. Chargement et stockage des ressources internes à l'application

Le résultat de cette initialisation est donc une variable statique de type *Map* (en fait de type *LockableMap* : sous-classe de *Map* rendue non modifiable après appel de la méthode *lock*) qui s'appelle *internal* et qui contient en tout et pour tout **13 objets** appelés objets internes et identifiés par leurs clés de stockage, que nous allons détailler ci-dessous.

3.2.1 Objet interne internal.errorlist

La clé de stockage **internal.errorlist** correspond à un objet de type *SupportList* (sous-classe de *List* munie de la caractéristique *Support*) initialement vide et prévu pour stocker les exceptions survenues durant le déroulement de l'application, sous forme de tableau Object[] contenant les informations suivantes : source, identifiant, exception, date d'apparition.

3.2.2 Objet interne internal.mapping

La clé de stockage **internal.mapping** correspond à un objet de type *SupportMap* (sous-classe de *Map* munie de la caractéristique *Support*) initialement vide et prévu pour accueillir le mapping de l'application. Son remplissage à partir de ressources internes comme externes est effectué ultérieurement.

3.2.3 Objet interne internal.poolmap

La clé de stockage **internal.poolmap** correspond à un objet de type *SupportMap* (sous-classe de *Map* munie de la caractéristique *Support*) initialement vide mais qui va servir à stocker tous les objets internes et toutes les données de l'application lors des phases suivantes du démarrage. On peut considérer cet objet (couramment appelé "pool") comme la zone de stockage extensible du gestionnaire dans laquelle les entités pourront aussi stocker leurs données de diverses manières.

3.2.4 Objet interne internal.internalmap

La clé de stockage **internal.internalmap** correspond à l'objet *internal* lui-même !! (j'avoue ne plus trop me souvenir pourquoi j'ai décidé que la map *internal* devait se contenir elle-même, si ce n'est peut être pour qu'elle soit considérée comme un objet interne).

3.2.5 Objet interne internal.managerid

La clé de stockage **internal.managerid** correspond à une chaine de caractères qui prend la valeur de la constante *MANAGERID* définie dans *KassiaSystem*. Il s'agit de l'identifiant du gestionnaire obtenu par la concaténation de son nom et sa date de version. Par exemple : Kassia_20110507

```
public static final String MANAGERBUILD = "20110507";
public static final String MANAGERID = "Kassia_"+MANAGERBUILD;
```

3.2.6 Objet interne internal.startdate

La clé de stockage **internal.startdate** correspond à l'objet de type *Date* crée au tout début du constructeur de la classe *KassiaApp*, assimilié à l'instant de démarrage de l'application et transmis par l'organe *AppLauncher* à la méthode *initSystem*.

3.2.7 Objet interne internal.object

La clé de stockage **internal.object** correspond à l'instance de la classe *KassiaMain* transmise par l'organe *AppLauncher* à la méthode *initSystem*. Il est appelé "objet de démarrage" parce qu'il s'agit du premier objet crée par le code source gestionnaire, dont la classe contient la méthode *main*.

3.2.8 Objet interne internal.args

La clé de stockage **internal.args** correspond au tableau de String des arguments de l'application, transmis par l'organe *AppLauncher* à la méthode *initSystem*, mais ayant préalablement été reconstruit en prenant le point virgule comme délimiteur, grâce à la méthode *formatArgs* de *KassiaSystem* ci-dessous :

```
private static void formatArgs(String[] args)
{
    if(args.length==0)
    {
          appArgsLine = "";
          appArgs = new String[0];
          return;
    }
    StringBuilder b = new StringBuilder(args[0]);
    for(int i=1;i<args.length;i++) b.append(" "+args[i]);
    appArgsLine = b.toString();

    appArgs = appArgsLine.replace(";;","\\u152").split(";",-1);
    for(int i=0;i<appArgs.length;i++) appArgs[i] = appArgs[i].replace("\\u152",";");
}
```

La méthode reconstruit la chaîne de caractères des arguments telle que passée dans la commande de lancement de l'application Java, puis interprète cette chaine en prenant le point-virgule comme délimiteur. Deux point-virgules accolés sont considérés comme un point-virgule devant être ignoré par le découpage.

Pourquoi avoir redécoupé ainsi les arguments de l'application ? Pour donner la possibilité de prendre comme argument des chemins d'accès de fichiers ou de répertoires susceptibles de contenir des espaces.

3.2.9 Objet interne internal.params

La clé de stockage **internal.params** correspond à un objet de type *LockableMap* (sous-classe de *Map* non modifiable) construit à partir des arguments de l'application qui sont de la forme : *<clé>=<valeur>*. Les valeurs ainsi récupérées sont potentiellement utilisés comme paramétrage du gestionnaire, y compris dans la phase d'initialisation actuelle. La map est construite grâce à la méthode *initParams* de *KassiaSystem* ci-dessous :

```
private static void initParams() throws Exception
{
   for(int i=0;i<appArgs.length;i++)
   if(appArgs[i].contains("="))
   {
          String[] n = appArgs[i].split("=",2);
          String key = n[0];
          String value = n[1];
```

```
            if(params.containsKey(key))
            throw new Exception("Param ["+key+"] is found many times inside args");
            params.put(key,value);
      }
   }
```

Seuls les arguments de type <clé>=<valeur> sont pris en compte, les valeurs <clé> et <valeur> étant alors stockées dans la map. Si deux arguments présentent la même valeur de clé, alors une exception est générée et l'initialisation du gestionnaire échoue entrainant une erreur fatale (que nous verrons après)

3.2.10 Objet interne internal.inside

La clé de stockage **internal.inside** correspond un objet de type *Inside*, une classe définie dans le package *gus05.manager.gus.kassia.layer2.tools* qui implémente l'interface *Retrieve*. Cet objet est utilisé par le gestionnaire comme point d'entré pour accéder aux ressources internes de l'application (ressources contenues dans le jar), et est initialisé grâce à la méthode *initInside* de *KassiaSystem* ci-dessous :

```
private static void initInside()
{
   String id = param(PARAM_INSIDE,PARAMDEFAULT_INSIDE);
   inside = new Inside(id);
}

public static final String PARAM_INSIDE = "inside";
public static final String PARAMDEFAULT_INSIDE = "/gus05/resource/gus/kassia/";
```

Le constructeur de *Inside* prend en paramètre une chaîne de caractères qui lui indique quel est le chemin racine dans l'arborescence des package de l'application à partir duquel il doit rechercher les ressources internes. Cette valeur est par défaut égale à la constante *PARAMDEFAULT_INSIDE*, (c'est à dire : */gus05/resource/gus/kassia/*) mais est substituée par la valeur associé au paramètre *inside*, si celui-ci est défini. Si vous voulez modifier le chemin racine de recherche des ressources internes, il vous suffit donc d'ajouter dans les arguments de lancement de l'application : *inside=<nouveau-chemin-root>;...*

A quoi cela peut-il servir, me demanderez vous, si le chemin racine */gus05/resource/gus/kassia/* est déjà prévu pour contenir les ressources ? Cela donne la possibilité de prévoir plusieurs emplacements de ressource dans l'application : l'un par défaut, et d'autres officieux qu'on activerait en ajoutant l'argument adéquate par exemple pour lancer l'application en mode débug, en mode secret, ou que sais-je.

3.2.11 Objet interne internal.propmap

La clé de stockage **internal.propmap** correspond à un objet de type *LockableMap* (sous-classe de *Map* non modifiable) contenant les propriétés internes de l'application, chargées à partir d'une éventuelle ressource interne. Son contenu est initialisé grâce à la méthode *initProps* de *KassiaSystem* ci-dessous :

```
private static void initProps() throws Exception
{
    String id = param(PARAM_PROP,PARAMDEFAULT_PROP);
    Properties p = inside.prop(id);
    if(p!=null) props.putAll(p);

    initProp(PROPKEY_APPNAME,APPDEFAULT_NAME);
    initProp(PROPKEY_APPTITLE,APPDEFAULT_TITLE);
    initProp(PROPKEY_APPVERSION,APPDEFAULT_VERSION);
    initProp(PROPKEY_APPBUILD,APPDEFAULT_BUILD);
}

private static void initProp(String key, String value)
{if(!props.containsKey(key)) props.put(key,value);}


public static final String PARAM_PROP = "prop";
public static final String PARAMDEFAULT_PROP = "prop";

public static final String PROPKEY_APPNAME = "app.name";
public static final String PROPKEY_APPTITLE = "app.title";
public static final String PROPKEY_APPVERSION = "app.version";
public static final String PROPKEY_APPBUILD = "app.build";

public static final String APPDEFAULT_NAME = "Kassia_core1";
public static final String APPDEFAULT_TITLE = "Kassia application";
public static final String APPDEFAULT_VERSION = "N";
public static final String APPDEFAULT_BUILD = "N";
```

Comme pour *initInside*, une valeur est récupérée par défaut égale à *prop* et modifiable par l'ajout d'un argument *prop=<nouvelle-valeur>;...* Cette valeur correspond au chemin relatif utilisé par l'objet *inside* pour rechercher la ressource interne suceptible d'initialiser *internal.propmap*. Avec des valeurs par défaut de *inside* et *prop*, l'objet *inside* va donc regarder à l'emplacement suivant : */gus05/resource/gus/kassia/prop*

Si cet emplacement se révèle inexistant, l'objet Properties renvoyé par la méthode prop de l'objet inside sera null et aucune propriété ne sera ajouté à *internal.propmap*. La méthode initProps termine par l'initialisation des 4 propriétés internes obligatoires de la manière suivante :

- **app.name** ajoutée avec la valeur par défaut **Kassia_core1** si non trouvée.
- **app.title** ajoutée avec la valeur par défaut **Kassia application** si non trouvée.
- **app.version** ajoutée avec la valeur par défaut **N** si non trouvée.
- **app.build** ajoutée avec la valeur par défaut **N** si non trouvée.

Il faut noter que ces propriétés ne sont ajoutées avec leurs valeurs par défaut que si elles n'étaient pas définies dans la ressource interne ou qu'il n'y avait pas de ressource interne. La map *internal.propmap* a donc toujours une taille minimale de 4.

3.2.12 Objet interne internal.entitymap

La clé de stockage **internal.entitymap** correspond à un objet de type *LockableMap* contenant les classes d'entités internes éventuelles, et initialisé grâce à la méthode *initEntities* de *KassiaSystem* ci-après.

3.2.13 Objet interne internal.multiplicitymap

La clé de stockage **internal.multiplicitymap** correspond à un objet de type *LockableMap* contenant les valeurs de multiplicité associées aux entités internes dont les classes sont stockées dans l'objet *internal.entitymap*.

Les objets internes *internal.entitymap* et *internal.multiplicitymap* sont initialisés en même temps grâce à la méthode *initEntities* ci-dessous :

```
private static void initEntities() throws Exception
{
    String id = param(PARAM_LOAD,PARAMDEFAULT_LOAD);
    Properties p = inside.prop(id);
    if(p==null) return;

    Iterator it = p.keySet().iterator();
    while(it.hasNext())
    {
        String key = (String)it.next();
        String value = p.getProperty(key);
        if(!key.startsWith("#")) initEntity(key,value);
    }
}

public static final String PARAM_LOAD = "load";
public static final String PARAMDEFAULT_LOAD = "load";
```

Comme pour *initProps*, une valeur est récupérée par défaut égale à *load* et modifiable par l'ajout d'un argument *load=<nouvelle-valeur>;...* valeur correspondant au chemin relatif utilisé par l'objet *inside* pour rechercher la ressource interne suceptible d'initialiser les entités internes. Avec des valeurs par défaut de *inside* et *load*, l'objet *inside* va donc regarder à l'emplacement suivant : */gus05/resource/gus/kassia/load*

Si cet emplacement se révèle inexistant, l'objet Properties renvoyé par la méthode *prop* de l'objet

inside sera null et aucune entité interne ne sera chargée (même si leurs classes sont présentes dans le fichier jar de l'application). Dans le cas contraire, les couples clé/valeur de la map transmise sont interprétés de la manière bien connue :

<entity-name>=<entity-classname><multiplicity>
avec *<multiplicity>* pouvant prendre les deux valeurs possibles : + ou *
Voici la méthode qui effectue le chargement d'une entité interne à partir du couple clé/valeur. Cette méthode complète à la fois la map *internal.entitymap* et la map *internal.multiplicitymap*.

```
    private static void initEntity(String key, String value) throws Exception
    {
       if(!value.contains("."))
              value = key+"."+value;
       if(!value.startsWith(ENTITYNAMESTART))
              value = ENTITYNAMESTART+value;

       if(value.endsWith("+"))
              multiplicities.put(key,MULTIPLICITYVALUE_UNIQUE);
       else if(value.endsWith("*"))
              multiplicities.put(key,MULTIPLICITYVALUE_MULTIPLE);
       else throw new Exception("Invalid internal entity definition: ["+value+"] for
entity ["+key+"]");

       value = value.substring(0,value.length()-1);
       Class c = Class.forName(value);
       entities.put(key,c);
    }

    public static final String ENTITYNAMESTART = "gus05.entity.";
    public static final String MULTIPLICITYVALUE_MULTIPLE = "MULTIPLE";
    public static final String MULTIPLICITYVALUE_UNIQUE = "UNIQUE";
```

Il faut noter que cette méthode est susceptible de générer une exception et donc de mettre en echec l'initialisation du gestionnaire conduisant à une erreur fatale.

3.2.14 Conclure sur l'initialisation

Nous sommes à présent en mesure de découvrir le code source la méthode *initSystem* :

```
    public static void initSystem(Object object, Date startDate, String[] args)
    {
        try
        {
            errors = new SupportList("errors");
            mapping = new SupportMap("mapping");
            pool = new SupportMap("pool");

            props = new LockableMap("props");
            params = new LockableMap("params");
            entities = new LockableMap("entities");
            multiplicities = new LockableMap("multiplicities");
            internal = new LockableMap("internal");
```

```
            appStartDate = startDate;
            appObject = object;

            formatArgs(args);
            initParams();
            initInside();
            initProps();
            initEntities();
            initInternal();

            props.lock();
            params.lock();
            entities.lock();
            multiplicities.lock();
            internal.lock();
        }
        catch(Exception e)
        {fatalError_initSystem(e);}
    }
```

Ainsi que la méthode *initInternal* qui ajoute à la map *internal* les 13 objets ayant préalablement été initialisés :

```
    private static void initInternal()
    {
        internal.put(INTERNAL_APPARGS,appArgs);
        internal.put(INTERNAL_APPOBJECT,appObject);
        internal.put(INTERNAL_APPSTARTDATE,appStartDate);
        internal.put(INTERNAL_MANAGERID,MANAGERID);
        internal.put(INTERNAL_MAP_INTERNAL,internal);
        internal.put(INTERNAL_MAP_POOL,pool);
        internal.put(INTERNAL_MAP_PROPS,props);
        internal.put(INTERNAL_MAP_CLASSENTITY,entities);
        internal.put(INTERNAL_MAP_MULTIPLICITY,multiplicities);
        internal.put(INTERNAL_MAP_MAPPING,mapping);
        internal.put(INTERNAL_MAP_PARAMS,params);
        internal.put(INTERNAL_LIST_ERROR,errors);
        internal.put(INTERNAL_INSIDE,inside);
    }

    public static final String INTERNAL_APPARGS = "internal.args";
    public static final String INTERNAL_APPOBJECT = "internal.object";
    public static final String INTERNAL_APPSTARTDATE = "internal.startdate";
    public static final String INTERNAL_MANAGERID = "internal.managerid";
    public static final String INTERNAL_MAP_INTERNAL = "internal.internalmap";
    public static final String INTERNAL_MAP_POOL = "internal.poolmap";
    public static final String INTERNAL_MAP_PROPS = "internal.propmap";
    public static final String INTERNAL_MAP_CLASSENTITY = "internal.entitymap";
    public static final String INTERNAL_MAP_MULTIPLICITY = "internal.multiplicitymap";
    public static final String INTERNAL_MAP_MAPPING = "internal.mapping";
    public static final String INTERNAL_MAP_PARAMS = "internal.params";
    public static final String INTERNAL_LIST_ERROR = "internal.errorlist";
    public static final String INTERNAL_INSIDE = "internal.inside";
```

Une fois l'ensemble des objets initialisés, la méthode *initSystem* se termine en vérouillant les 5 objets *LockableMap* qu'il contient par l'appel de leurs méthodes *lock*, les rendant désormais non modifiables.

Parmi les 13 éléments de la map *internal*, les seuls qui pourront désormais être modifiées sont : **internal.errorlist**, **internal.mapping** et **internal.poolmap,** les autres éléments étant soit issus de ressources internes, soit issus du démarrage de l'application, soit des constantes.

3.3 Les deux modes de démarrage de l'application

Une fois le gestionnaire initialisé, le thread de démarrage "KASSIA_INIT_THREAD" appelle la méthode *startApplication,* qui se contente en mode normal d'appeler la méthode *start* de la classe *KS4_start* située dans le package *gus05.manager.gus.kassia.layer3*, passant ainsi la main à la troisième et dernière couche de l'application. Il est toutefois possible de paramétrer un mode spécial en ajoutant dans les arguments : *start=<nom-entité-interne>;...*

De cette façon, la méthode *startApplication_special* est appelée et se contente alors d'instancier l'entité interne dont le nom est spécifié en valeur de paramètre. Il s'agit là néanmoins d'un cas très particulier qui n'est utilisé que rarement et sur lequel nous ne nous attarderons pas ici.

```
public static void startApplication()
{
     try
     {
          String startKey = param(PARAM_START,PARAMDEFAULT_START);
          if(startKey.equals(PARAMDEFAULT_START)) KS4_start.start();
          else startApplication_special(startKey);
     }
     catch(Exception e)
     {fatalError_startApplication(e);}
}


private static void startApplication_special(String startKey)
{
     try
     {
          if(!entities.containsKey(startKey))
     throw new Exception("Unknown internal entity for special start: ["+startKey+"]");
          Class c = (Class) entities.get(startKey);
          c.newInstance();
     }
     catch(Exception e)
     {fatalError_startApplication_special(e);}
}

public static final String PARAM_START = "start";
public static final String PARAMDEFAULT_START = "start";
```

Je me permet ici un petit apparté sur la méthode *param* de la classe *KassiaSystem*, qui est appelée au moment de l'initialisation de plusieurs objets internes et dans la méthode *startApplication* :

```
public static String param(String key, String defaultValue)
{
   if(params.containsKey(key)) return (String) params.get(key);
   if(props.containsKey("param."+key)) return (String) props.get("param."+key);
   return defaultValue;
}
```

Nous constatons qu'il est possible d'ajouter dans les propriétés internes de l'application des propriétés commençant par "param." pour fixer des valeurs par défaut aux paramètres lorsqu'ils ne sont pas définis dans les arguments.

Les paramètres *inside* et *prop* étant exploités avant le chargement éventuel des propriétés internes, il est vain de vouloir ajouter des propriétés *param.inside* et *param.prop* pour les altérer. En revanche, ajouter des propriétés *param.load* et surtout *param.start* permet d'influencer le fonctionnement de l'application puisque les valeurs par défaut *load* et *start* ne seront plus considérées. Notamment, ajouter la propriété interne : *param.start = <entity-name>* permettra de créer une application se lançant par défaut en mode spéciale avec instanciation de l'entité *<entity-name>*.

3.4 A propos de erreur fatale

L'erreur fatale survient dans le gestionnaire Kassia lorsqu'une exception est générée dans l'un des 4 cas suivants :

1. méthode *initSystem* de *KassiaSystem*
2. méthode *startApplication* de *KassiaSystem*
3. méthode *startApplication_special* de *KassiaSystem*
4. méthode *uncaughtException* de *Impl_AppLauncher*

Les méthodes statiques suivantes de *KassiaSystem* sont alors respectivement appelées en passant à chaque fois l'objet Throwable récupéré à partir du *try catch*.

```
protected static void fatalError_initSystem(Throwable e)
{new FatalErrorHolder("initSystem",1,e);}

protected static void fatalError_startApplication(Throwable e)
{new FatalErrorHolder("startApplication",2,e);}

protected static void fatalError_startApplication_special(Throwable e)
{new FatalErrorHolder("startApplication_special",3,e);}

protected static void fatalError_uncaughtException(Throwable e)
{new FatalErrorHolder("uncaughtException",4,e);}
```

Ces méthodes se contentent d'instancier la classe *FatalErrorHolder* contenue dans le package *gus05.manager.gus.kassia.layer2.tools*, laquelle est prévue pour gérer l'erreur fatale. Les paramètres passés dans l'objet sont : un identifiant indiquant le nom de la méthode à l'origine de

l'exception, un nombre entier utilisé comme "exit code" dans l'appel la méthode System.exit(..), et l'exception elle-même.

Quels sont les différents cas dans lesquels peut survenir une erreur fatale ?

1. Lorsque **l'initialisation du gestionnaire échoue**, c'est à dire :
   1. Lorsque les arguments de l'application définissent deux fois la même clé de param
   2. Lorsque le chargement d'une ressource interne génère une exception (classe *Inside*)
   3. Lorsque la ressource interne de chargement des entités contient des erreurs (mauvaise syntaxe, noms d'entités internes inexistantes...)
   4. Lorsque l'accés dynamique aux classes des entités internes échoue (méthode *forName* de la classe *Class*)
2. Lorsque **le démarrage normale échoue**, c'est à dire : Une erreur survient dans la méthode *start* de la classe *KS4_start* et est considérée comme suffisamment grave pour faire échouer tout le démarrage.
3. Lorsque **le démarrage spécial échoue**, c'est à dire : Une erreur survient à l'instanciation de l'entité de démarrage spécial.
4. Lorsqu'une **erreur inhabituelle** survient dans le thread de démarrage forçant celui-ci à se terminer prématurément. (StackOverflow...)

Que fait la classe *FatalErrorHandler* ?

Cette classe va :
1. Ecrire un rapport d'erreur fatale dans un fichier texte à coté du jar applicatif
2. Afficher une boite de dialogue d'erreur intitulé "FATAL ERROR"
3. Quitter l'application avec l'instruction *System.exit(exitCode);*

Le fichier texte de rapport d'erreur fatale permettra au gestionnaire de consigner l'ensemble des informations disponibles, susceptibles d'aider à comprendre ce qu'il s'est passé : la version du gestionnaire Kassia, le contexte de lancement de l'application, l'environnement Java, ou encore les informations liées à l'exception elle-même. Le nom du fichier correspond à la règle syntaxique suivante : *fatalerr_<timestamp>_<managerid>.txt*

Voici à titre d'exemple un fichier de rapport d'erreur fatale généré sur l'une de mes applications gus05 (je ne dirai pas laquelle...), le 8 mai 2011 à 14H48 :

Fichier *fatalerr_20110508_144849_Kassia_20110507.txt* apparu à coté du fichier jar applicatif et contenant le texte suivant :

```
FATAL ERROR REPORT
------------------
manager ID: Kassia_20110507
args line: root=app_gusophtalmo
launch dir: C:\GUS\KASSIA
```

```
internal state: [er,ma,pr,po,pa,en,in]=[0,367,4,76,1,0,13]
start time: 20110508_144835
abort time: 20110508_144849
exit code: 2
------------------
java version: 1.6.0_21
java home: C:\Program Files\Java\jre6
------------------
exception source startApplication
exception time: 20110508_144849
exception type: java.lang.NullPointerException
exception message: null
------------------
java.lang.NullPointerException
       at java.awt.Dimension.<init>(Unknown Source)
       at java.awt.Component.getMaximumSize(Unknown Source)
       at java.awt.Container.getMaximumSize(Unknown Source)
       at javax.swing.JComponent.getMaximumSize(Unknown Source)
       at javax.swing.BoxLayout.checkRequests(Unknown Source)
       at javax.swing.BoxLayout.preferredLayoutSize(Unknown Source)
       at javax.swing.JToolBar$DefaultToolBarLayout.preferredLayoutSize(Unknown Source)
       at java.awt.Container.preferredSize(Unknown Source)
       at java.awt.Container.getPreferredSize(Unknown Source)
       at javax.swing.JComponent.getPreferredSize(Unknown Source)
       at java.awt.BorderLayout.layoutContainer(Unknown Source)
       at java.awt.Container.layout(Unknown Source)
       at java.awt.Container.doLayout(Unknown Source)
       at java.awt.Container.validateTree(Unknown Source)
       at java.awt.Container.validateTree(Unknown Source)
       at java.awt.Container.validateTree(Unknown Source)
       at java.awt.Container.validateTree(Unknown Source)
       at java.awt.Container.validateTree(Unknown Source)
       at java.awt.Container.validateTree(Unknown Source)
       at java.awt.Container.validate(Unknown Source)
       at javax.swing.JLayeredPane.addImpl(Unknown Source)
       at java.awt.Container.add(Unknown Source)
       at javax.swing.JRootPane.setContentPane(Unknown Source)
       at javax.swing.JFrame.setContentPane(Unknown Source)
       at gus05.manager.gus.kassia.layer3.v04.initmaingui.V04.initMainPanel(V04.java:20)
       at gus05.manager.gus.kassia.layer3.KS4_start.buildApp(KS4_start.java:284)
       at gus05.manager.gus.kassia.layer3.KS4_start.start(KS4_start.java:122)
       at
gus05.manager.gus.kassia.layer2.KassiaSystem.startApplication(KassiaSystem.java:335)
       at
gus05.manager.gus.kassia.layer2.organs_impl.Impl_AppLauncher.run(Impl_AppLauncher.java:2
3)
       at java.lang.Thread.run(Unknown Source)
```

Dans cet exemple, on apprend que le démarrage normal de l'application a échoué à cause d'une exception *NullPointerException* survenue au moment précis le ou gestionnaire appelle la méthode setContentPane sur la fenêtre principale pour y ajouter l'interface graphique de l'application. La raison ? Le *NullPointerException* est survenu au moment de l'appel de l'un des composants Swing de mon interface graphique qui ne s'était apparemment pas initialisé correctement. Il s'agit d'un bug graphique qui survient de manière non systématique et dont la cause est donc par essence difficilement appréhendable.

3.5 La gestion des erreurs classiques

Les erreurs fatales ne représentent pas la majorité des cas d'erreur fort heureusement. La plupart du temps, les erreurs non fatales (ne nécessitant pas l'arrêt immédiat de l'application) surviennent pendant le fonctionnement des entités et sont alors transmises à la méthode *err* de la classe *Outside*, laquelle les transmet à l'organe *ErrorHandler* qui les transmet à son tour à la classe *KassiaSystem*, comme nous le constatons ci-dessous :

```
package gus05.manager.gus.kassia.layer2.organs_impl;

import gus05.framework.core.Entity;
import gus05.manager.gus.kassia.layer1.organs.ErrorHandler;
import gus05.manager.gus.kassia.layer2.KassiaSystem;

public class Impl_ErrorHandler extends KassiaSystem implements ErrorHandler {

    public void handleError(Entity entity, String id, Exception e)
    {err(entity,id,e);}
}
```

La méthode statique *err* de la classe *KassiaSystem* (ci-dessous) récupère toutes les informations qui lui sont transmises : la source, l'identifiant et l'exception. En ajoutant un objet Date correspondant à l'instant de stockage de l'exception (qui peut être assimilée à l'instant ou l'exception a été générée), ces 4 données sont rangées dans un tableau Object[], lequel est ajouté à la liste *errors* (objet interne *internal.errorlist* de la map *internal*).

```
    public static void err(Object source, String id, Exception e)
    {errors.add(new Object[]{source,id,e,new Date()});}
```

Et la gestion de l'exception s'arrête là ? Ne pourrait-on souhaiter qu'un message d'erreur soit consigné dans la log de l'application au moment même ou cette erreur survient ? Sachant que la liste de stockage est de type *Support*, il sera toujours possible d'y ajouter par la suite un listener afin de réagir en temps réel à l'apparition des exceptions, de les récupérer et de les traiter de la manière souhaitée.

Il est intéressant de remarquer que l'objet *source* peut être de n'importe quel type et pas seulement une entité. De fait, la méthode *err* est appelée partout dans le gestionnaire ou des erreurs classiques peuvent survenir (création d'une entité de démarrage qui échoue...)

3.6 L'accès aux objets internes du gestionnaire

On appelle objet interne, les objets qui sont stockés dans la map *internal* du gestionnaire ou dans la map *pool* (objet interne *internal.poolmap* de la map *internal*). Comme nous l'avons vu précédemment, les objets internes directement stockés dans la map *internal* ne peuvent être remplacés. Il est cependant possible de les substituer simplement en ajoutant dans la map *pool*

des objets équivalents avec les mêmes clés de stockage. En effet, le mécanisme d'accès aux objets internes (que ce soit pour satisfaire la demande d'une entité ou la demande d'un autre objet interne) est conçu pour passer systématiquement par la méthode statique *find* de la classe *KassiaSystem*, dont le code est donné ci-dessous :

```
public static Object find(String key)
{
    if(pool.containsKey(key)) return pool.get(key);
    if(internal.containsKey(key)) return internal.get(key);
    return null;
}
```

La map *pool* est d'abord consultée et si et seulement si elle ne contient pas l'objet désiré, la map *internal* est consultée. Ainsi dans le cas ou la valeur *key* passée en paramètre correspond à une clé de stockage présente dans les deux maps, c'est l'objet de la map *pool* qui est renvoyé. Par exemple, il suffira pour "falsifier" la valeur du *managerid* d'ajouter la valeur souhaitée dans la map *pool* en utilisant la clé de stockage *internal.managerid* (mais encore reste-t-il à trouver une utilité à cela...).

Il faudra noter tout de même que les implémentations d'organes ne se font pas "duper" par ce mécanisme de substitution puisqu'elles font appel directement aux variables statiques de la classe *KassiaSystem*. Par exemple, ajouter dans la pool une map avec la clé de stockage *internal.mapping* permettra de tromper une entité qui cherche à obtenir le mapping de l'application, mais pas l'organe *Impl_KeyFinder* qui utilise directement la variable statique *mapping* de *KassiaSystem*, comme nous allons le voir par la suite.

3.7 Object internes prédéfinis utilisés dans la deuxième couche

La map pool étant laissée vide dans la deuxième couche, les seuls objets internes actuellement prédéfinis sont les 13 objets de la map *internal*. Pourtant, on trouve dans le code source de la classe *KassiaSystem*, les méthodes statiques suivantes, prévues pour renvoyer d'autres objets internes qui n'existent pas encore.

```
public static Retrieve findResourceProvider()
{return (Retrieve) find(POOL_RESOURCEPROVIDER);}

public static Transform findResourceChanger()
{return (Transform) find(POOL_RESOURCECHANGER);}

public static Register findMappingRecorder()
{return (Register) find(POOL_MAPPINGRECORDER);}

public static Transform findMappingHandler()
{return (Transform) find(POOL_MAPPINGHANDLER);}

public static Transform findServiceWrapper()
{return (Transform) find(POOL_SERVICEWRAPPER);}
```

```
public static final String POOL_RESOURCEPROVIDER = "app.resourceprovider";
public static final String POOL_RESOURCECHANGER = "app.resourcechanger";
public static final String POOL_MAPPINGRECORDER = "app.mappingrecorder";
public static final String POOL_MAPPINGHANDLER = "app.mappinghandler";
public static final String POOL_SERVICEWRAPPER = "app.servicewrapper";
```

En fait, il s'agit (à l'exception notable de *app.resourceprovider*) d'objet internes utilisés de manière optionnelle dans la deuxième couche par les implémentations des 3 organes *Impl_KeyFinder*, *Impl_ResourceProvider* et *Impl_ServiceWrapper* que nous allons étudier très prochainement.

A l'exception de l'objet interne *app.resourceprovider* qui sera initialisé dans la troisième couche du gestionnaire, les 4 autres objets ne sont initialisés que sous l'effet d'un paramétrage spécifique. Il s'agit donc pour ces 4 cas de possibilités d'évolution dans le fonctionnement normal des organes, comme nous nous en rendrons compte bientôt.

3.8 Les deux méthodes statiques restantes de KassiaSystem

Si vous jetez un oeil à la classe KassiaSystem, vous constaterez que nous l'avons presque intégralement parcourue, à l'exception de deux méthodes statiques pas vraiment cruciales que je vous propose de voir rapidement.

```
public static String sourceName(Object s)
{
    if(s==null) return "null";
    if(s instanceof Source) return s.toString();
    if(s instanceof Entity) return ((Entity)s).getName();
    if(s instanceof Class) return ((Class)s).getSimpleName();
    return s.getClass().getSimpleName();
}
```

Cette méthode qui fournit une représentation texte d'un objet considéré comme une source, est utilisée à divers endroits dans le gestionnaire Kassia, notamment dans l'implémentation de l'organe *ServiceWrapper* (ce qui explique sa présence dans la deuxième couche).

```
public static String systemState()
{
   return "[er,ma,pr,po,pa,en,in]=["+
                errors.size()+","+
                mapping.size()+","+
                props.size()+","+
                pool.size()+","+
                params.size()+","+
                entities.size()+","+
                internal.size()+"]";
}
```

Cette méthode qui fourni une représentation texte de l'état de stockage général du gestionnaire Kassia et n'est utilisé que dans les messages de log de la troisième couche et dans la classe *FatalErrorHolder* .

3.9 Les 3 organes : KeyFinder, ResourceProvider, ServiceWrapper

Pour clore l'étude de la deuxième couche du gestionnaire, il reste à présent à étudier le fonctionnement des organes *KeyFinder*, *ResourceProvider* et ServiceWrapper qui interviennent dans la résolution des appels d'entités pour obtenir de la part du gestionnaires ressources et services souhaités par les entités.

3.9.1 Etude de l'organe KeyFinder

L'implémentation de l'organe *KeyFinder*, la classe *Impl_KeyFinder* (qui hérite de *KassiaSystem*) contient la méthode *findKey* définie par l'interface *KeyFinder* de l'organe :

```
public String findKey(Source src) throws Exception
{
   String value = findValue(src);

     if(value==null)
          throw new Exception("Invalid mapping for source ["+src+"]: null value");
     if(value.equals(""))
          throw new Exception("Invalid mapping for source ["+src+"]: empty value");

     recordCall(src,value);
     return value;
}
```

La méthode *recordCall* permet d'enregistrer la correspondance de mapping établie sous réserve que l'objet interne *app.mappingrecorder* ait été préalablement initialisé. Il s'agit de l'une des possibilités d'évolution du gestionnaire précédemment mentionées.

```
private void recordCall(Source src, String value)
{
     try
     {
          Register r = findMappingRecorder();
          if(r!=null) r.register(src.toString(),value);
     }
     catch(Exception e)
     {err(this,"recordCall(Source,String)",e);}
}
```

Cette méthode susceptible de générer une exception considérée comme une erreur non fatale, fait appel à la méthode err de *KassiaSystem*, qui comme je l'avais précisé n'est pas exclusivement réservée à l'organe *ErrorHandler*.

Le traitement de recherche de la clé de génération à partir du mapping est executé par la méthode *findValue* dont voici le code source :

```
private String findValue(Source src) throws Exception
{
    Transform t = findMappingHandler();
    if(t!=null) return (String) t.transform(toDataTab(src));

    Map map = determineMap(src);
    String value = determineValue(map,src);
    if(map.containsKey(value)) return (String)map.get(value);
    return value;
}
```

Là aussi, on voit apparaître une possibilité d'évolution puisque la méthode délègue son traitement à un autre "mapping handler" sous réserve que l'objet interne *app.mappinghandler* ait été initialisé.

Le traitement normal de recherche de valeur de clé de génération se fait en 3 étapes :

1. Recherche de la map de correspondance à utiliser (méthode *determineMap*)
2. Recherche de la valeur correspondant à la source dans la map (méthode *dermineValue*)
3. Application du principe d'alias si la valeur renvoyée est elle-même contenue dans la map

La première étape est assurée par la méthode *determineMap* que voici :

```
private Map determineMap(Source src)
{
    String mapID = "mapping."+src.getEntityName();
    Map m = (Map) find(mapID);
    if(m!=null) return m;
    return mapping;
}
```

Cette méthode détermine quelle map de correspondance (autrement dit, quel mapping) doit être utilisée pour rechercher la clé de génération. A priori on pourrait croire qu'il s'agit du mapping de l'application, l'objet interne internal.mapping, correspondant aussi à la variable statique mapping de la classe KassiaSystem. C'est habituellement le cas, sauf si vous décidez d'ajouter à la pool un mapping de substitution pour l'entité appelante, en utilisant la clé de stockage suivante :
mapping.<entity-name>

La deuxième étape est assurée par la méthode *determineValue* que voici :

```
private String determineValue(Map map, Source src)
{
    String key1 = src.toString();
    if(map.containsKey(key1)) return (String)map.get(key1);
    String key2 = src.toString2();
    if(map.containsKey(key2)) return (String)map.get(key2);
    String key3 = src.toString3();
    if(map.containsKey(key3)) return (String)map.get(key3);
    return src.getId();
}
```

C'est ici qu'il devient intéressant de regarder le code source de la classe *Source* qui participe au travers de ses 3 méthodes *toString*, *toString2* et *toString3* au mécanisme de granularité des règles de mapping.

```
package gus05.manager.gus.kassia.layer1;

import gus05.framework.core.Entity;

public final class Source {

    private Entity entity;
    private String name;
    private String pseudo;
    private String id;

    private String s1;
    private String s2;
    private String s3;


    public Source(Entity entity, String id) throws Exception
    {
        if(id==null)
throw new Exception("Invalid call for entity ["+entity.getName()+"]: null id");
        if(id.equals(""))
throw new Exception("Invalid call for entity ["+entity.getName()+"]: empty id");

        this.entity = entity;
        this.id = id;

        name = entity.getName();
        pseudo = name.split("\\.")[0];
        s3 = "@"+id;
        s1 = name+s3;
        s2 = pseudo+s3;
    }

    public String getId()           {return id;}
    public Entity getEntity()       {return entity;}
    public String getEntityName()   {return name;}
    public String getPseudo()       {return pseudo;}

    public String toString()        {return s1;}
    public String toString2()       {return s2;}
    public String toString3()       {return s3;}
}
```

Ainsi on peut constater qu'un objet source qui a été crée à partir d'une entité <entity-name> et d'un identifiant d'appel <id> renvoit à travers ces méthodes toString, toString2 et toString3, les valeurs suivantes respectivement :

<entity-name>@<id>
<entity-pseudo>@<id>
@<id>

Il s'agit là des trois niveaux de représentation de la source du plus specifique (le nom exacte de la

source) au plus général (son identifiant d'appel). Le code source de la méthode *determineValue* corresponds alors à l'algorithme du mécanisme de granularité des règles de mapping qui veut que pour une source particulière, on recherche d'abord une règle de mapping correspondant au nom exacte de la source, puis à défaut une règle correspondant à sa représentation intermédiaire (*<entity-pseudo>@<id>*), puis à défaut une règle correspondant à sa représentation générale (*@<id>*), puis à défaut on renvoie l'identifiant d'appel lui même.

La troisième étape consiste à appliquer le principe d'alias, à savoir que si la valeur renvoyée à l'issu de la deuxième étape est elle-même contenue dans le mapping, alors on considère qu'il s'agit d'un alias et on renvoie finalement sa valeur associé dans le mapping. Le code source de cette troisième étape se résume dans les deux dernières lignes de la méthode *findValue* :

```
        if(map.containsKey(value)) return (String)map.get(value);
        return value;
```

3.9.2 Etude de l'organe ResourceProvider

L'implémentation de l'organe *ResourceProvider*, la classe *Impl_ResourceProvider* (qui hérite de *KassiaSystem*) contient la méthode *provideResource* définie par l'interface *ResourceProvider* de l'organe :

```
    public Object provideResource(Source src, String key) throws Exception
    {
        if(key.equals("null")) return null;
        if(key.equals("pool")) return pool;
        if(key.equals("internal")) return internal;

        Object resource = searchForResource(src,key);
        return changeResource(src,key,resource);
    }
```

Les règles de générations que doit pouvoir interpréter cet organe seront très variées et pour certaines complexes. Cependant le code source de la méthode provideResource nous permet déjà de prendre note de trois règles très simples et indépendantes de la source, que nous pourrons utiliser au besoin dans le mapping de nos applications gus05 :

La règle de génération **null** permet de renvoyer la valeur **null**.
La règle de génération **pool** permet de renvoyer l'objet interne **pool** (non substituable).
La règle de génération **internal** permet de renvoyer l'objet interne **internal** (non substituable).

Pour la suite, tout se passe dans la méthode *searchForResource*, la méthode *changeResource* n'étant qu'une possibilité d'évolution du gestionnaire, comme nous allons le voir ci-dessous :

```
    private Object changeResource(Source src, String key, Object resource)
    {
        try
        {
             Transform t = (Transform) findResourceChanger();
             if(t!=null) return t.transform(toDataTab(src,key,resource));
        }
        catch(Exception e)
        {err(this,"changeResource(Source,String,Object)",e);}
        return resource;
    }
```

Venons en maintenant à la méthode searchForResource qui permet de trouver (ou générer ?) la ressource correspondant à la règle de génération transmise (laquelle peut s'avérer très complexe). Avec un peu de malchance, cette méthode va encore déléger la résolution vraiment délicate du problème à d'autres objets internes. C'est effectivement le cas, mais à chaque fois, les choses se précisent un peu plus.

```
    private Object searchForResource(Source src, String key) throws Exception
    {
        key = removeTail(key);
        Object obj = find(key);
        if(obj!=null) return obj;

        Retrieve provider = findResourceProvider();
        if(provider==null) throw new Exception("No resource provider available inside
pool");

        Register provider_reg = (Register) provider;
        provider_reg.register("entity",src.getEntity());
        provider_reg.register("id",src.getId());

        return provider.retrieve(key);
    }
```

Tout d'abord, la clé de génération subit une petite réctification éventuelle en passant à travers la méthode *removeTail* dont voici le code source :

```
    private String removeTail(String key)
    {
       if(!key.contains("-")) return key;
       String[] n = key.split("-",2);
       if(!isInteger(n[1])) return key;
       return n[0];
    }

    private boolean isInteger(String s)
    {
       try{Integer.parseInt(s);return true;}
       catch(Exception e){return false;}
    }
```

Que fait cette méthode removeTail ? Si la clé de génération se termine par -1 (ou -2, ou -3 ...) alors cette partie est retirée de la clé. Je vais maintenant tenter de vous expliquer l'intérêt que peut avoir un tel rabotage de la clé.

La clé de génération peut avoir deux origines : le mapping ou l'entité elle-même (lorsque l'identifiant d'appel est considéré par défaut comme règle de génération, faute de mapping). C'est ce deuxième cas qui nous intéresse ici.

Lorsqu'un développeur Java développe des entités, il choisit généralement comme identifiants d'appel les noms des entités qu'il souhaite utiliser par défaut comme service pour la nouvelle entité. Dans le cas particulier ou il souhaite utiliser plusieurs la même entité pour obtenir ainsi plusieurs services identiques, il écrira un code qui pourra ressembler à ceci :

s1 = Outside.service(this,"entity1");
s2 = Outside.service(this,"entity1");

Le fait d'utiliser pour une même entité plusieurs fois le même identifiant d'appel rend les différents appels indifférentiables au niveau du mapping qui s'avèrait incapable de leur appliquer des règles différentes si le besoin s'en faisait sentir. Pour éviter cet eccueil, il vaut mieux donc ne jamais utiliser plusieurs fois le même identifiant d'appel dans une entité.

Mais alors comme faire si on souhaite utiliser plusieurs fois la même règle par défaut ? La réponse réside dans le code ci-dessous :

s1 = Outside.service(this,"entity1-1");
s2 = Outside.service(this,"entity1-2");

Au niveau du mapping, il s'agit bien d'idenfiants différents auxquels peuvent être assignées des règles distinctes. Au niveau du resourceProvider, ces règles par défaut seront considérées comme équivalentes à "entity1" grâce au mécanisme de rabotage assuré par la méthode *removeTail*. Le tour est joué !

Revenons à notre méthode searchForResource... Tout d'abord, la règle de génération est passée dans la méthode find de la classe KassiaSystem pour vérifier si un objet interne ne correspondrait pas par hasard à cette clé. Le cas échéant, l'objet est renvoyé.

```
Object obj = find(key);
if(obj!=null) return obj;
```

Le reste de la méthode consiste à faire appel à l'objet interne "app.resourceprovider" qui doit être de type Retrieve et éventuellement aussi Register.

S'il s'avère être du type Register, les informations de la source (entité et identifiant) lui sont transmis de la manière suivante :

```
provider_reg.register("entity",src.getEntity());
provider_reg.register("id",src.getId());
```

Après quoi, sa méthode retrieve est appelée pour récupérer l'objet souhaité à partir de la clé de génération.

Finalement l'organe *ResourceProvider* se contente de vérifier si la clé de génération n'est pas égale à une des trois valeurs particulières "null", "pool" et "internal" ou à l'identifiant d'un objet interne, avant de déléguer le traitement de génération proprement dit (faisant intervenir à la fois la clé de génération et l'objet source) à l'objet interne nommé "app.resourceprovider", lequel sera ajouté ultérieurement à la pool par la troisième couche du gestionnaire.

3.9.3 Etude de l'organe ServiceWrapper

L'implémentation de l'organe *ServiceWrapper*, la classe *Impl_ServiceWrapper* (qui hérite de *KassiaSystem*) contient la méthode *wrapService* définie par l'interface *ServiceWrapper* de l'organe. Voici le code source de cette implémentation :

```
package gus05.manager.gus.kassia.layer2.organs_impl;

import gus05.framework.core.Service;
import gus05.framework.features.Transform;
import gus05.manager.gus.kassia.layer1.Source;
import gus05.manager.gus.kassia.layer1.organs.ServiceWrapper;
import gus05.manager.gus.kassia.layer2.KassiaSystem;
import gus05.manager.gus.kassia.layer2.services.*;

public class Impl_ServiceWrapper extends KassiaSystem implements ServiceWrapper {


    public Service wrapService(Source src, Object obj) throws Exception
    {
        Transform t = findServiceWrapper();
        if(t!=null) return (Service) t.transform(toDataTab(src,obj));

        if(obj==null) return new EmptyService();
        return new DefaultService(src,obj);
    }

    private Object[] toDataTab(Source src, Object obj)
    {return new Object[]{src.getEntity(),src.getId(),obj};}
}
```

Les deux premières lignes de la méthode ne sont qu'une possibilité d'évolution du gestionnaire en faisant appel à un éventuel objet interne *app.servicewrapper*. Ensuite, le code source de la méthode utilise deux classes *EmptyService* et *DefaultService*, toutes deux contenues dans le package : *gus05.manager.gus.kassia.layer2.services*

Nous distinguons deux cas de figure :

Si l'objet à envelopper est la valeur null, alors une instance de *EmptyService* est renvoyée à l'entité appelante. Il 'agit d'une coquille vide, un objet service inerte qui ne fait rien et se contente de

renvoyer la valeur null lorsqu'on lui demande quelque chose.

```
package gus05.manager.gus.kassia.layer2.services;

import java.awt.event.ActionListener;
import java.util.List;
import javax.swing.JComponent;
import gus05.framework.core.Service;

public class EmptyService implements Service {

    public void run() {}
    public void execute() throws Exception {}
    public void give(Object obj) throws Exception {}
    public void register(String key, Object obj) throws Exception {}
    public boolean filter(Object obj) throws Exception {return false;}
    public double function(double value) throws Exception {return 0;}
    public Object transform(Object obj) throws Exception {return null;}
    public Object retrieve(String key) throws Exception {return null;}
    public Object provide() throws Exception {return null;}
    public JComponent gui() {return null;}
    public void addActionListener(ActionListener listener) {}
    public void removeActionListener(ActionListener listener) {}
    public List listeners() {return null;}
}
```

Dans le cas contraire, une instance de *DefaultService* est créée à partir de l'objet à envelopper mais aussi de la source, puis renvoyée à l'entité appelante. Il s'agit cette fois d'une véritable enveloppe qui teste préalablement les différentes caractéristiques disponibles sur l'objet pour les enregistrer dans des variables distinctes. L'appel des différentes méthodes définies dans l'interface Service délègue à chaque fois le traitement vers l'objet en se référant à la variable correspondante. Si cette dernière n'a pas été initialisée parce que la caractéristique correspondante n'est pas disponible sur l'objet, alors une exception est générée s'il y a lieu. Le code source ci-dessous vous éclairera un peu plus sur son mécanisme.

```
package gus05.manager.gus.kassia.layer2.services;

import java.awt.event.ActionListener;
import java.util.List;
import javax.swing.JComponent;
import gus05.framework.core.Service;
import gus05.framework.features.*;
import gus05.manager.gus.kassia.layer1.Source;
import gus05.manager.gus.kassia.layer2.KassiaSystem;


public class DefaultService implements Service {

    private Object target;
    private Source source;

    private Execute execute;
    private Filter filter;
    private Function function;
    private Give give;
    private Graphic graphic;
    private Provide provide;
```

```
    private Register register;
    private Retrieve retrieve;
    private Support support;
    private Transform transform;
    private Runnable runnable;

    private void notAvailable(String feature) throws Exception
    {throw new Exception(feature+" not available for "+source+"
(="+KassiaSystem.sourceName(target)+")");}


    /*
     * AMELIORATION POSSIBLE DU SYSTEME :
     * permettre la créaction de l'objet target au moment du premier appel de méthode
     * en passant non pas l'objet target lui même mais un "provide target" de type
Provide
     */
    public DefaultService(Source source, Object target)
    {
        this.source = source;
        this.target = target;

        if(target instanceof Execute)         execute = (Execute)target;
        if(target instanceof Filter)          filter = (Filter)target;
        if(target instanceof Function)        function = (Function)target;
        if(target instanceof Give)            give = (Give)target;
        if(target instanceof Graphic)         graphic = (Graphic)target;
        if(target instanceof Provide)         provide = (Provide)target;
        if(target instanceof Register)        register = (Register)target;
        if(target instanceof Retrieve)        retrieve = (Retrieve)target;
        if(target instanceof Support)         support = (Support)target;
        if(target instanceof Transform)       transform = (Transform)target;
        if(target instanceof Runnable)        runnable = (Runnable)target;
    }

    public void execute() throws Exception
    {
        if(execute==null) notAvailable("execute");
        execute.execute();
    }

    public boolean filter(Object obj) throws Exception
    {
        if(filter==null) notAvailable("filter");
        return filter.filter(obj);
    }

    public double function(double value) throws Exception
    {
        if(function==null) notAvailable("function");
        return function.function(value);
    }

    public void give(Object obj) throws Exception
    {
        if(give==null) notAvailable("give");
        give.give(obj);
    }

    public Object provide() throws Exception
    {
        if(provide==null) notAvailable("provide");
```

```
        return provide.provide();
    }

    public void register(String key, Object obj) throws Exception
    {
        if(register==null) notAvailable("register");
        register.register(key,obj);
    }

    public Object retrieve(String key) throws Exception
    {
        if(retrieve==null) notAvailable("retrieve");
        return retrieve.retrieve(key);
    }

    public Object transform(Object obj) throws Exception
    {
        if(transform==null) notAvailable("transform");
        return transform.transform(obj);
    }

    public JComponent gui()
    {
        if(graphic==null) return null;
        return graphic.gui();
    }

    public void addActionListener(ActionListener listener)
    {
        if(support==null) return;
        support.addActionListener(listener);
    }

    public void removeActionListener(ActionListener listener)
    {
        if(support==null) return;
        support.removeActionListener(listener);
    }

    public List listeners()
    {
        if(support==null) return null;
        return support.listeners();
    }

    public void run()
    {
        if(runnable==null) return;
        runnable.run();
    }
}
```

3.10 Récapitulatif des deux premières couches

Dans le déroulement de l'application, nous en sommes rendus à l'appel de la méthode statique *start* de la classe *KS4_start* (là ou commence la troisième couche), dans la méthode *startApplication* de *KassiaSystem*, et avant d'aborder cette troisième et dernière couche, je vous propose de récapituler rapidement ce que nous avons appris sur les deux premières couches.

La première couche prend en charge le démarrage de l'application et l'initialisation de la classe Outside (appel de la méthode setManager), en exigeant de la deuxième couche qu'elle lui fournisse des implémentations pour les 5 organes du gestionnaire.

La deuxième couche quant à elle s'occupe d'initialiser une structure de stockage (la map *internal*) en la remplissant avec les données fondamentales du gestionnaire (issues de ressources internes ou directement du démarrage). La map pool laissée vide jusqu'à présent est prévue pour accueillir des objets internes supplémentaires parmi lesquels l'objet *app.resourceprovider,* objet indispensable pour le bon fonctionnement de l'organe ResourceProvider. Un mécanisme de possibilité d'évolution permet aussi à la deuxième couche d'altérer le fonctionnement normal des organes dans le cas où des objets internes optionels sont ajoutés, correspondant aux identifiants :

app.resourcechanger (alteration du fonctionnement de l'organe *ResourceProvider*)
app.mappingrecorder (enregistrement des règles de mapping effectives dans *KeyFinder*)
app.mappinghandler (alteration du fonctionnement de l'organe dans *KeyFinder*)
app.servicewrapper (alteration du fonctionnement de l'organe dans *ServiceWrapper*)

On remarquera au passage que seul les organes participant au traitement des appels d'entité peuvent être altérés : *KeyFinder, ResourceProvider* et *ServiceWrapper*

Avant d'entrer dans la troisième couche nous avons donc :

1. une map internal non modifiable, contenant 13 objets (dont la map pool)
2. une map pool vide qui attend des objets internes supplémentaires
3. une map de mapping vide qui attend de recevoir les règles de mapping de l'application
4. un mécanisme de prise en compte des erreurs classique qui les enregistre dans une liste
5. un mécanisme de gestion d'erreur fatale qui met fin à l'application et écrit un rapport

Il est temps à présent de franchir le pas et passer à la dernière couche de l'application !

4 La troisième couche du gestionnaire

Avant d'aborder la troisième couche, quelques constations doivent être faites. Si nous regardons la répartition du code source du gestionnaire en terme de nombre de classes et taille mémoire, nous obtenons ceci :

1. Première couche : 8 classes Java, 5 Ko
2. Deuxième couche : 15 classes Java, 41 Ko
3. Troisième couche : 82 classes Java, 290 Ko

Autant se l'avouer tout de suite, nous n'avons pas encore étudié la moitié du gestionnaire ! Si jusqu'à présent nous avons pu parcourir et commenter l'essentiel du code source, nous allons devoir abandonner cette prétention avec la troisième couche. Ceci étant, sans rentrer dans le détail du code source de chaque objet interne, nous allons expliquer leurs rôles et relations au sein du gestionnaire pour vous permettre d'en comprendre et d'en exploiter les moindres aspects.

4.1 La structure de la troisième couche

KassiaSystem :
constantes statiques : 40
variables statiques : 14
méthodes statiques : 11

KS1_Ref :

Etape préparatoire
Etape de remplissage de la pool
Etape de construction de l'application
Etape de lancement du runtime
Etape de résumé log

Etapes préparatoires

```
public static void start() throws Exception
{
	if(!errors.isEmpty())
```

```
                    throw new Exception("Error list already contains "+errors.size()+"
elements");

        putPool(POOL_MAP_SYSPROP,System.getProperties());
        putPool(POOL_MAP_SYSENV,System.getenv());
        putPool(POOL_PRINTSTREAM_SYSOUT,System.out);
        putPool(POOL_PRINTSTREAM_SYSERR,System.err);
        putPool(POOL_MAP_CONSTANTS,constants());
        putPool(POOL_SUPPORT_START,new DefaultSupport());
        putPool(POOL_SUPPORT_EXIT,new DefaultSupport());
        putPool(POOL_SUPPORT_EXIT2,new DefaultSupport());
        putPool(POOL_FILTER_TRUE,trueFilter);
        putPool(POOL_FILTER_FALSE,falseFilter);
        putPool(POOL_TRANSFORM_ID,idTransform);
        putPool(POOL_GRAPHIC_PANEL,panelGraphic);

        initPropMap();
        initPrintStreams();

        printM_("start()","printstreams initialized");
        printM_("start()","manager ID: ["+MANAGERID+"]");
        printM_("start()","internal id: ["+appGus05ID()+"]");
        printM_("start()","internal state: "+systemState());
        if(hasArgs()) printM_("start()","arguments line: ["+appArgsLine+"]");
```

Etape de remplissage de la pool

```
    private static void initPool() throws Exception
    {
      initpool_start = millis();

      lapse();
      T01.checkJavaVersion();t();
      T02.initErrorHandler();t();
      T03.initExecuteExit();t();
      T04.initExecutePrintDebug();t();
      T05.initFileInitializer();t();
      T06.initFileProvider();t();
      T07.initFileProvider_pool();t();
      T08.initUEncoderDecoder();t();

      // resource loading
      T09.loadProperties();t();
      T10.loadIcons();t();
      T11.loadSounds();t();
      T12.loadLings();t();
      T13.loadMapping();t();
      T14.loadEntityInfo();t();

      // entity generation
      T15.initEntityProviders();t();
      T16.initEntityGenerator();t();
      T17.initEntityClassHolder();t();
      T18.initEntityLoader();t();
      T19.initEntityClassLoader();t();
      T20.initEntityClassLoader_Bin();t();
      T21.initEntityClassLoader_Jar();t();
```

```
        // callers
        T22.initResourceCaller();t();
        T23.initServiceCaller();t();

        // data provider
        T24.initDataProvider();t();
        T25.initDataBuilder();t();
        T26.initDataSourceFinder();t();
        T27.initDataPersistence();t();
        T28.initDataBackup();t();

        // ling & action
        T29.initLingSupport();t();
        T30.initActionBuilder();t();
        T31.initActionHandler();t();
        T32.initActionUpdater();t();
        T33.initActionInfoBuilder();t();

        // resource provider
        T34.initResourceProvider();t();
        T35.initResourceProvider_basic();t();
        T36.initResourceProvider_output();t();
        T37.initResourceProvider_file();t();
        T38.initResourceProvider_file1();t();
        T39.initResourceProvider_dir();t();
        T40.initResourceProvider_entity();t();
        T41.initResourceProvider_entity1();t();
        T42.initResourceProvider_data();t();
        T43.initResourceProvider_data1();t();
        T44.initResourceProvider_process();t();
        T45.initResourceProvider_wrap();t();
        T46.initResourceProvider_combine();t();
        T47.initResourceProvider_lingProvider();t();
        T48.initResourceProvider_iconProvider();t();
        T49.initResourceProvider_actionBuilder();t();
        T50.initResourceProvider_putpool();t();

        // last pool objects
        T51.initMainFrame();t();
        T52.initMainGuiProvider();t();
        T53.initLogFile();t();

        initpool_end = millis();
    }
```