Documentation
Projet gus05

Titre : Ateliers de formation – partie 1/2

Auteur : Augustin Delale

Version : 19/02/2010

ATELIERS DE FORMATION
PARTIE 1/2

Version du 19/02/2010

Table des matières

Table des matières

Avant propos

Ce document constitue la première partie d'une suite d'ateliers pratiques pour découvrir le projet gus05 qui est un projet de développement collaboratif en langage Java autour du framework gus05. Aucune connaissance préalable sur ce nouveau framework n'est requise pour suivre ces ateliers néanmoins il vous est conseillé d'avoir des rudiments en langage Java. Chaque atelier débute par un descriptif détaillé des objectifs visés, lesquels sont récapitulés dans le tableau ci-dessous.

Vous allez donc à travers une succession d'exemples concrets apprendre petit à petit à utiliser le framework gus05 pour développer vos propres composants de programmation et concevoir des applications. Vous découvrirez aussi de quelle manière ces composants peuvent être réutilisés pour nous amener vers un travail collaboratif entre développeurs dont le but final est la mise en place d'une communauté autour du framework gus05.

Je vous souhaite à tous une excellente lecture en espérant pouvoir collaborer prochainement avec vous sur de passionnants projets de développement d'applications. Bien amicalement !

Augustin

Contacts

Adresse du site web dédié à projet gus05
http://www.gus05.com

Forum dédié au projet gus05 :
http://gus05.forumactif.com

Pour plus de renseignements, n'hésitez pas à me contater à mon adresse mail perso :
adelale@hotmail.com

Copyright

Légende

Zones de texte :

Définitions : L'ensemble des définitions de base qu'il vous faudra assimiler

Remarques : Certains points sur lesquels je souhaite attirer votre attention

Attention ! : Les mises en gardes importantes

Instructions : Ce que vous devez accomplir sur votre ordinateur pour suivre ce tutorial

Code source : les classes java que vous devez recopier (ou copier-coler...) dans l'éditeur d'Eclipse

Les zones de texte non coloriées concernent quant à elles les explications du tutorial.

Description des ateliers

ATELIER 1 **Installation et premier lancement d'Eclipse**	Cet atelier vous propose d'installer sur votre ordinateur la JVM Java et Eclipse (logiciel permettant de développer en langage Java). Les développeurs Java habitués de l'IDE Eclipse pourront ignorer cet atelier.
ATELIER 2 **Installation et lancement d'une application de test**	Dans un premier temps, nous allons télécharger et installer les sources du framework et du gestionnaire Kassia puis effectuer un premier lancement de l'application test de Kassia.
ATELIER 3 **Ma première entité**	Pour débuter, développons une première entité graphique très simple : l'affichage d'un panneau de couleur verte. A travers cet exemple, on se familiarisera avec la notion d'entité et on apprendra le paramètrage de base.
ATELIER 4 **Les ressources : icônes et sons**	Nous allons ajouter des ressources à notre application et développer des entités pour les visualiser. Nous verrons aussi comment ajouter du paramétrage pour personnaliser l'apparence de l'application.
ATELIER 5 **Le multi-affichage d'entités**	Nous montrons ici comment il est possible d'afficher ensemble plusieurs entités graphiques. Cet atelier permettra d'introduire les notions de service et de mapping.
ATELIER 6 **Déployer son application**	Cet atelier explique comment créer un fichier JAR sous Eclipse pour obtenir une application Java directement utilisable. On appelle cette opération le déploiement.
ATELIER 7 **Le paramètrage extérieur : configurer un répertoire root**	Lorsqu'on dispose d'une application déployée, il est possible de lui ajouter du paramétrage en spécifiant un répertoire root dans lequel elle va chercher des fichiers.
ATELIER 8 **Paramètrer l'accès aux ressources extérieures**	L'accès aux fichiers et répertoires extérieurs à l'application se paramètre au moyen des pathIDs. Nous créerons une entité graphique pour visualiser ce paramètrage.
ATELIER 9 **Les entités de transformation**	Après les entités graphiques, nous nous attaquons à un nouveau type d'entité : les entités de transformation. Nous prendrons comme exemple plusieurs transformations textuelles.
ATELIER 10 **Faire évoluer ses entités**	Nous allons montrer comment on peut faire évoluer nos entités en reprenant le code source de l'entité développée à l'atelier 8. L'idée est de rendre personnalisable les icônes des fichiers
ATELIER 11 **Les caractéristiques et types de l'entité**	Il existe 10 caractéristiques possibles pour l'entité. Jusqu'à présent, nous n'avons vu que les entités graphiques et les entités de transformation. Il reste 8 caractéristiques à voir, en particulier la caractéristique Support.
ATELIER 12 **Recevoir des évenements**	Nous allons voir comment une entité peut recevoir des évènements extérieurs. Comme exemple, nous allons créer une entité avec des éditeurs de texte synchronisés et une autre pour jouer un son au démarrage et à l'arrêt de l'application.

ATELIER 13 **Afficher le statut de la connexion Internet**	Nous attaquons avec cet atelier notre première fonctionnalité complexe... Il s'agit de mettre en place un système pour réagir en temps réel aux connexions et déconnexions de votre ordinateur à Internet et changer en conséquence l'icône d'affichage du statut de connexion.
ATELIER 14 **Les actions et les menus**	Cet atelier montre comment le gestionnaire Kassia permet de créer simplement des actions et de les ajouter à des menus. Ce sera aussi l'occasion d'approfondir le concept de multiplicité.
ATELIER 15 (en cours) **Le multilinguisme dynamique**	Le gestionnaire Kassia permet aussi de changer dynamiquement la langue de vos interfaces graphiques. Nous illustrerons ce mécanisme avec un exemple simple de traduction de menu en anglais, français et japonais.
ATELIER 16 (en cours) **Récapitulatif sur le framework et Kassia**	Voici un récapitulatif de ce que nous avons appris sur le framework gus05 lui-même et sur le gestionnaire Kassia. Cela sera l'occasion de réflechir sur leurs rôles respectifs et sur la portée des connaissances que nous avons acquises avant d'aborder ce qu'il reste à étudier.

Récapitulatif des téléchargements

Atelier 1
- http://java.sun.com/javase/downloads/index.jsp
- www.eclipse.org

Atelier 2
- framework.zip
- kassia.zip

Atelier 4
- ressources.zip

Atelier 7
- entity.zip

Atelier 9
- entity2.zip

Atelier 10
- icones2.zip

Atelier 12
- entity3.zip
- Start_Exit.zip

Atelier 13
- Online_Offline.zip

Atelier 14
- ResourcesAtelier14.zip
- gus.app.action.about2a.jar

Atelier 15
- LingResources.zip

ATELIER 1 : Installation et premier lancement d'Eclipse

Objectifs : Cet atelier vous propose d'installer sur votre ordinateur la JVM Java et Eclipse (logiciel permettant de développer en langage Java). Les développeurs Java habitués de l'IDE Eclipse pourront ignorer cet atelier.

Etapes :

1. Vérifier la présence de Java sous Windows
2. Télécharger et installer la machine virtuelle de Java
3. Télécharger et installer l'application Eclipse
4. Premier lancement d'Eclipse

Ce premier atelier concerne principalement les débutants en Java qui partent de rien et doivent donc installer sur leur ordinateur les logiciels nécessaires pour faire de la programmation Java. Il s'adresse aussi aux développeurs Java qui n'ont pas l'habitude de développer avec l'outil Eclipse et qui souhaitent l'installer pour éxecuter les ateliers dans des conditions similaires aux miennes (Vous l'aurez deviné, je développe avec Eclipse !)

Etape 1 : Vérifier la présence de Java sous Windows

Avant toute chose, il est utile de savoir si votre ordinateur est équipé de Java (on parle de machine virtuelle Java : JVM) et le cas échéant, quelle version est installée. Il est en effet nécessaire d'avoir une JVM récente (version 1.5 au minimum) et dans le cas contraire, vous devrez réinstaller une version plus récente.

Pour ceux qui sont sous Windows : allez dans le menu Démarrer / Executer... puis tapez la commande "cmd" afin de lancer la console DOS. Ensuite tapez la ligne de commande suivante : "java -version".

Trois cas de figures se présentent :

- Si vous obtenez un message du genre "Commande inconnue java", alors Java n'a pas encore été installé sur votre ordinateur et l'étape suivante est donc pour vous.
- Si vous obtenez une version de Java inférieure à 1.5 alors l'étape suivante est aussi pour vous
- Si vous obtenez une version de Java supérieur ou égale à 1.5, alors vous pouvez passer directement à l'étape 3

Etape 2 : Télécharger et installer la machine virtuelle de Java

Vous devez d'abord aller à l'adresse suivante pour télécharger la dernière version en ligne du "Java SE Runtime Environment" : http://java.sun.com/javase/downloads/index.jsp

À l'heure où j'écris cet atelier (le 29 août 2009), la dernière version en ligne est "JRE 6 Update 16" qui correspond au fichier "jre-6u16-windows-i586.exe" mais pour vous, il s'agira sûrement d'une version plus avancée (Java 7 si vous êtes en 2010).

Note : à partir des versions 1.5 de java, il a été décidé de parler de Java 5, Java 6, Java 7

Une fois le fichier téléchargé, il vous suffit le lancer l'exécutable pour installer Java. Il s'agit d'une installation classique de logiciel pour laquelle il vous suffit de cliquer sur les boutons "suivant" jusqu'au bouton final "terminer".

Voilà, Java est installé sur votre ordinateur ! Pour vous en convaincre, vous pouvez refaire l'étape 1.

Etape 3 : Télécharger et installer l'application Eclipse

Occupons-nous à présent d'Eclipse. Vous trouverez ce logiciel téléchargeable gratuitement sur le site web www.eclipse.org. Parmi les nombreuses variantes que propose le site, je vous conseille de choisir "Eclipse Classic" qui à l'heure où j'écris en est rendu à la version 3.5.0 et fait 162 MO.

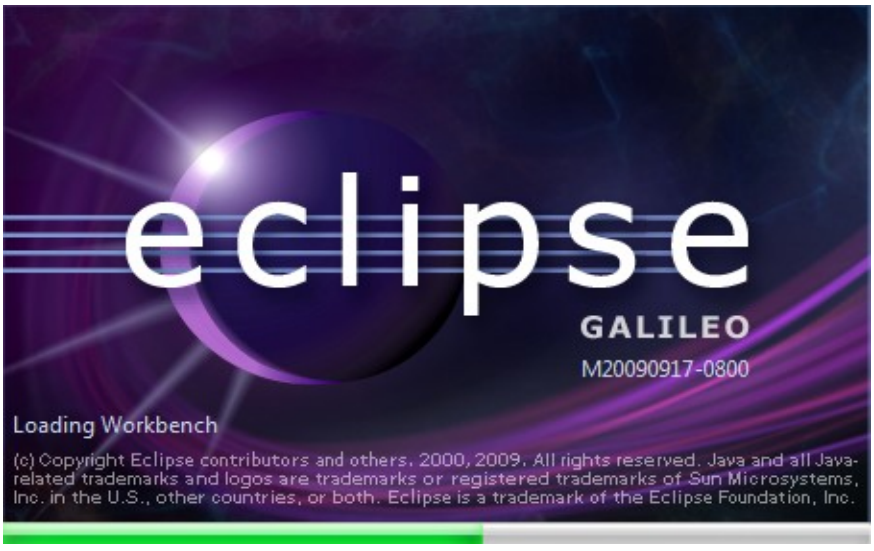
Une fois l'archive zip téléchargée, il vous suffit de la dézipper dans l'emplacement de votre choix (personnellement, je la place toujours sur le C). Il s'agit en effet d'une application directement executable sans installation, en lançant le fichier "eclipse.exe"

Etape 4 : Premier lancement d'Eclipse

Pour sauvegarder votre travail Java, Eclipse va avoir besoin d'un répertoire de travail appelé Workspace. Vous êtes libre de choisir l'emplacement qui vous convient, dans lequel se retrouveront toutes vos classes Java (code source et compilé).

Au premier lancement d'Eclipse, il vous est demandé de choisir ce répertoire. Si vous ne désirez plus en changer par la suite (a priori...) vous pouvez cocher la case à cet effet. Et voilà, Eclipse s'ouvre devant vous !

Si vous êtes novice sous Eclipse, les indications que je vous donnerai dans les prochains ateliers devront vous permettre de connaître le strict minimum pour vous débrouiller, mais bien sûr, vous pouvez faire du zèle en parcourant les nombreux tutoriaux disponibles sur le net.



ATELIER 2 : Installation et lancement d'une application de test

Objectifs : Dans un premier temps, nous allons télécharger et installer les sources du framework et du gestionnaire Kassia puis effectuer un premier lancement de l'application test de Kassia.

Etapes :

1. Créer un projet sous Eclipse
2. Ajouter 4 répertoires de source
3. Télécharger le code source
4. Installer et compiler le code source
5. Premier lancement de l'application
6. Lancement avec affichage des messages de log

Ce deuxième atelier devrait vous permettre de vous familiariser un peu avec Eclipse. Nous allons aussi récupérer le code source correspondant au framework gus05 et au gestionnaire Kassia, les compiler et lancer l'application de test de Kassia.

Mais tout d'abord, voici quelques explications d'ordre général. Le framework gus05 permet essentiellement de définir deux types de composants de programmation qu'on appelle "entité" et "gestionnaire", lesquels vont vous permettre de bâtir des applications. Pour vous aider à comprendre les rôles que jouent ces composants, disons simplement que si l'application était un orchestre, alors le gestionnaire serait le chef d'orchestre et les entités les musiciens. Autrement dit, le gestionnaire est le noyau central de votre application et les entités des sortes de briques fonctionnelles permettant de l'enrichir de nouvelles fonctionnalités.

Une application Java basée sur le framework gus05 se compose en fait de 4 parties distinctes (qui fourniront l'organisation de notre projet Eclipse) :

1. Le framework
2. Un gestionnaire
3. Un ensemble d'entités
4. Des ressources internes

L'application ayant impérativement besoin d'un gestionnaire pour se lancer et sachant qu'un gestionnaire est particulièrement complexe à développer (je vous rassure, vous allez apprendre en développant des entités), je vous ai fourni en plus du framework le gestionnaire Kassia qui est à l'heure actuelle le plus abouti de mes gestionnaires. Cet atelier permet ainsi de mettre en place une situation initiale qui ne comporte encore aucune entité mais dès l'atelier suivant, nous commencerons à en créer et à les ajouter à notre application.

Etape 1 : Créer un projet sous Eclipse

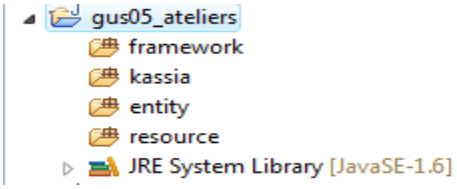
L'application Eclipse permet de créer des projets de développement correspondants à des sous-répertoires du répertoire Workspace que vous avez choisi. Pour vos développements gus05, vous allez avoir besoin de créer un projet que vous pourrez appeler par exemple "gus05_ateliers".

Sous Eclipse, allez dans le menu : *File / New / Java Project*, saisissez le nom du projet et cliquez sur le bouton *Finish*.

Etape 2 : Ajouter 4 répertoires de source

Vous allez ensuite créer 4 répertoires de sources dans le projet, afin d'y ranger séparément le code source du framework, du gestionnaire et des entités, ainsi que les ressources internes de votre application.

Faites un clic droit sur l'icône de votre projet () et choisissez dans le menu contextuel : *New / Source Folder*, enfin saisissez le nom du répertoire et cliquez sur le bouton *Finish*. Vous prendrez les noms suivants : "framework", "kassia", "entity" et "resource"



Remarque :

- Il se peut qu'un répertoire de source "src" soit ajouté par défaut au moment de la création du projet, vous pouvez le cas échéant renommer ce répertoire grâce au menu contextuel Refactor ou bien le supprimer grâce au menu contextuel Delete.

Etape 3: Télécharger le code source

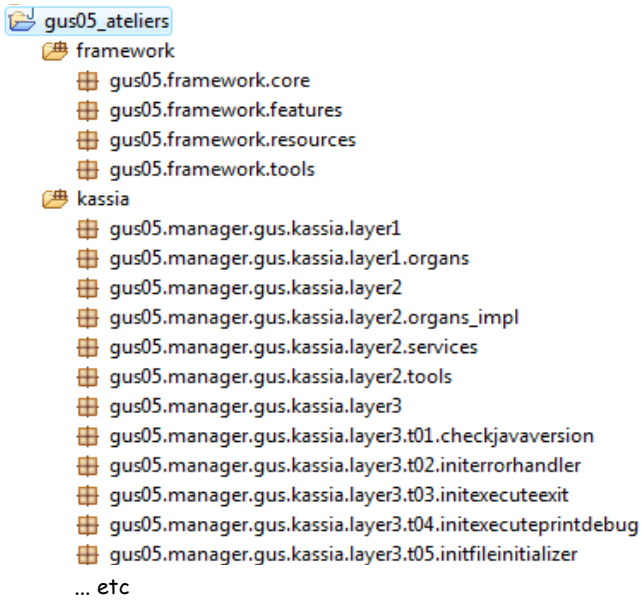
Vous pouvez télécharger le code source du framework gus05 et du gestionnaire Kassia en cliquant sur les liens de téléchargement ci-dessous :

framework.zip **kassia.zip**

Etape 4 : Installer et compiler le code source

En premier lieu, dézippez les deux archives zip que vous venez de télécharger : *framework.zip* et *kassia.zip*, puis copiez leurs contenus respectifs (sous-arborescence commençant par gus05/...) dans les répertoires de source () *framework* et *kassia* que vous retrouverez en vous rendant dans le répertoire projet *gus05_ateliers* lui-même contenu dans le répertoire Worskpace que vous avez choisi pour Eclipse.

Ensuite, il vous faut encore rafraîchir l'interface d'Eclipse pour voir effectivement apparaître les sources dans votre projet. Ceci est fait en sélectionnant le projet et en appuyant sur F5. Vous devriez alors obtenir quelque chose qui ressemble à ceci :



Et voilà, les répertoires *framework* et *kassia* contiennent désormais un ensemble de packages et de classes Java. Vous pouvez naturellement farfouiller à loisir dans le code source si vous êtes curieux de savoir comment le framework et le gestionnaire sont faits. Je vous fournirai des indications au fur et à mesure des ateliers pour vous aider à mieux comprendre tout ceci.

La compilation se fait en choisissant le menu : *Project / Build all*, ou plus simplement par la commande CTRL-B.

Pour votre information, les fichiers .class générés lors de cette compilation sont stockés dans un répertoire bin automatiquement créé dans le répertoire projet mais qui n'apparaît pas sous Eclipse.

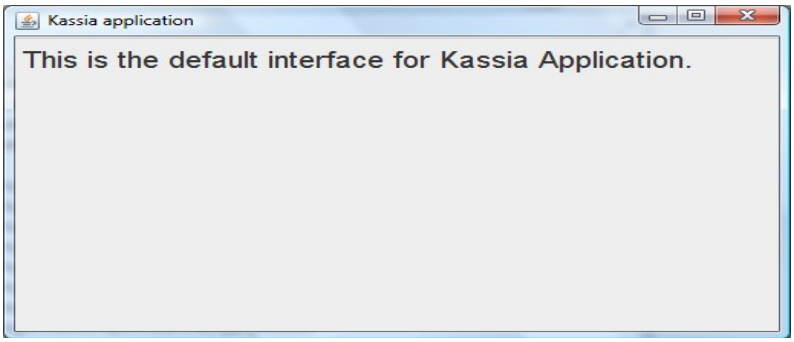
Etape 5 : Premier lancement de l'application

Le code source s'est compilé sans erreur. L'application du gestionnaire Kassia est donc prête à être lancée. Le code source ajouté dans votre projet ne comporte qu'une seule classe "main" possèdant une méthode *public static void main(String[] args)*. Il s'agit de la classe suivante :

gus05.manager.gus.kassia.layer2.KassiaMain

Ouvrez le répertoire de source kassia, puis le package gus05.manager.gus.kassia.layer2 et dedans, faites un clic droit sur l'icône de la classe *KassiaMain.* Choisissez dans le menu contextuel : *Run As / Java Application.*

L'application se lance et laisse apparaître la fenêtre suivante :



Vous venez de lancer l'application pour la première fois. Bravo ! Vous pouvez fermer la fenêtre pour quitter l'application.

Etape 6 : Lancement avec affichage des messages de log

Il est possible de configurer sous Eclipse les lancements d'application afin de garder en mémoire la classe qu'il faut lancer mais aussi de spécifier éventuellement des arguments à passer au démarrage. Nous allons donc créer une configuration de lancement sur notre classe *KassiaMain*.

Faites un clic droit sur l'icône de la classe *KassiaMain* et choisissez dans le menu contextuel : *Run As / Run configurations...* Prenez par exemple comme nom pour la configuration : "RunKassiaMain" puis dans le deuxième onglet (Arguments), ajouter "sysout" dans la zone texte "Program arguments", enfin cliquez sur le bouton *Apply* et sur le bouton *Run*.

L'application se lance de nouveau, mais cette fois l'argument "sysout" permet de faire afficher par le gestionnaire Kassia les messages de log de l'execution du programme dans la console. Vous pouvez quitter l'application et jeter un coup d'oeil à ce log.

Je vous expliquerai petit à petit comment interpréter tous ces messages particulièrement utiles pour comprendre comment s'est déroulée l'exécution du programme et débugger lorsque tout ne se passe pas comme prévu. Pour l'heure, voici néanmoins quelques explications de base :

Chaque ligne de message se structure en une succession d'informations comme suit :

1. le numéro du message
2. l'instant d'apparition dans le log (au format aaaammjj_hhmmss)
3. le numéro de la phase d'execution
4. le nombre total d'erreurs survenues jusque là
5. le type de message
6. l'origine du message
7. le contenu du message

Chaque information est séparée de la suivante par un espace, à l'exception du contenu du message qui vient après une tabulation.

Les différents types de messages possibles sont les suivants :

- [M] : message informatif de base
- [W] : message d'avertissement
- [E] : message d'erreur
- [D] : message de debuggage

Pour finir, le log se divise en 6 phases d'exécution distinctes (7 si on compte la phase initiale), correspondant à des tâches spécifiques dans le déroulement de l'application. Nous aurons l'occasion de revenir sur tout cela par la suite.

ATELIER 3 : Ma première entité

Objectifs : Pour débuter, développons une première entité graphique très simple : l'affichage d'un panneau de couleur verte. A travers cet exemple, on se familiarisera avec la notion d'entité et on apprendra le paramètrage de base.

Etapes :

1. Définition d'une entité graphique
2. Conventions de nommage et pseudo développeur
3. Ajouter un package et une classe au répertoire entity
4. Coder et compiler la classe
5. Premières notions de paramètrage
6. Les fichiers "prop" et "load"
7. Paramétrer l'entité
8. Tester l'entité

Etape 1 : Définition d'une entité graphique

Avant de développer une entité graphique, il est nécessaire de bien comprendre la définition générale d'une entité d'une part, et la définition d'une entité graphique d'autre part.

Une entité se définit au sens strict selon les trois règles suivantes :

1. **Il s'agit d'une ou plusieurs classes Java regroupées dans un unique package.**
2. **L'une des classes doit implémenter l'interface *Entity* définie par le framework gus05**
3. **Le nom du package est défini comme tel : <package-name> = gus05.entity.<entity-name>**

Une entité graphique se définit en ajoutant la règle ci-dessous :

1. **La classe principale qui implémente l'interface *Entity* doit aussi implémenter l'interface *Graphic* définie de même par le framework gus05**

Comme vous avez sans doute pu le constater, le framework gus05 comporte une grande majorité d'Interfaces (18 interfaces et seulement 2 classes) qui permettent notamment de définir les entités ainsi que de les caractériser. Dans le cas précis de l'entité graphique seulement deux interfaces nous intéressent :

gus05.framework.core.Entity (interface de définition des entités)
gus05.framework.feature.Graphic (interface de caractérisation graphique)

L'interface *Entity* définit 2 méthodes :

- *public String getName()* : fournit le nom de l'entité <entity-name>
- *public String getCreationDate()* : fournit la date de création de l'entité (format aaaa.mm.jj)

L'interface *Graphic* définit 1 méthode :

- *public JComponent gui()* : fournit l'objet Swing choisi pour représenter graphiquement l'entité

L'exemple de code source d'entité présenté à l'étape 4 vous fixera très certainement les idées sur la manière dont ces différentes méthodes peuvent être implémentées par la classe de l'entité. Mais avant cela, il nous reste à préciser certains points concernant le nommage des packages et des entités.

Etape 2 : Conventions de nommage et pseudo développeur

Vous avez dû remarquer qu'il existe des conventions de nommage générales pour les packages du projet gus05 suivant la catégorie de code auxquels ils appartiennent.

Catégorie de code	Le package commence par
Framework gus05	gus05.framework.
Gestionnaire	gus05.manager.
Entité	gus05.entity.
Ressources	gus05.resource.

Tous vos packages d'entité doivent donc impérativement débuter par gus05.entity.
Par ailleurs, pour reprendre la définition de l'entité, le nom du package est lié au nom de l'entité (nom renvoyé par la méthode getName() de l'interface Entity) de la manière suivante :
<package-name> = gus05.entity.<entity-name>

Dans la pratique, vous n'aurez donc qu'à vous soucier de la manière de nommer vos entités, ce qui nous amène aux conventions de nommage de celles-ci.

Conventions de nommage d'une entité :

Une entité se nomme en respectant les mêmes conventions qu'un package (succession de mots en minuscule séparés par des points) et afin d'éviter que deux développeurs ne créent des entités en les nommant de la même manière, le premier mot choisi devra impérativement être spécifique à chaque développeur, ce qu'on appelle le "pseudo développeur".

Mon pseudo développeur est "gus" et concernant les exemples de l'atelier, nous allons choisir le pseudo "charly". Quant à vous, dans la mesure où vous souhaiterez contribuer au projet gus05, vous allez devoir vous choisir votre pseudo et vous assurer que celui-ci n'a pas déjà été choisi par un autre développeur. Vous pouvez pour cela me contacter et me soumettre le pseudo que vous souhaitez adopter. Il est à noter que plusieurs mots-clés tels que "gus05", "entity", "manager", "package", "default" ou "kassia" ne peuvent être utilisés.

La suite du nom est laissée à votre soin et vous êtes donc libre de nommer vos entités comme vous le désirez, en gardant à l'esprit que votre manière de les nommer refletera l'organisation générale de votre travail et qu'elle mérite donc que vous y réfléchissiez sérieusement.

<entity-name> = <pseudo>.<what ever you want>

Etape 3 : Ajouter un package et une classe au répertoire entity

Il est temps de commencer à développer notre première classe d'entité ! Notre première entité consistant en l'affichage d'un panneau vert, nous allons la nommer :

charly.test.panel.green
correspondant à un nom de package: *gus05.entity.charly.test.panel.green*

Faites un clic droit sur l'icône du répertoire de source "entity" () et choisissez dans le menu contextuel : *New / Package*, enfin saisissez le nom du package et cliquez sur le bouton *Finish.*

Un package vide () est créé dans votre répertoire "entity". Nous allons y ajouter une classe Java que nous appelerons *GreenPanel* (le nom de la classe n'a pas vraiment d'importance). Faites un clic droit sur l'icône du package et choisissez dans le menu contextuel : *New / Class*, enfin saisissez le nom de la classe et cliquez sur le bouton *Finish.*

Une classe est créée () dans le package () et s'ouvre directement dans la partie de droite d'Eclipse. Cette classe est encore vide mais nous allons la compléter à l'étape suivante.

Remarque :

- **Si vous vous êtes trompé dans vos noms de package ou de classe et que vous souhaitez les changer, il vous faut savoir que les actions de renommage ou de déplacement doivent être éxecutées exclusivement à partir du menu contextuel *Refactor* d'Eclipse.**

Attention !

- **Ne modifiez pas les noms de classe ou de package directement dans le code source (même à partir de l'éditeur d'Eclipse).**
- **N'éditez pas les fichiers java correspondants, ne les renommez pas et ne les déplacez pas par d'autres moyens que le menu contextuel Refactor d'Eclipse**
- **Si vous vous trompez, des erreurs apparaitront dans votre projet Eclipse que vous ne pourrez corriger qu'en annulant vos actions.**

Etape 4 : Coder et compiler la classe

A présent, saisissez le code de votre classe comme ci-dessous :

```
package gus05.entity.charly.test.panel.green;

import java.awt.Color;
import javax.swing.JComponent;
import javax.swing.JPanel;
import gus05.framework.core.Entity;
import gus05.framework.features.Graphic;

public class GreenPanel implements Entity, Graphic {

	public String getName(){return "charly.test.panel.green";}
	public String getCreationDate(){return "2009.08.30";}
```

```
        private JPanel panel;

        public GreenPanel()
        {
                panel = new JPanel();
                panel.setBackground(Color.GREEN);
        }

        public JComponent gui()
        {return panel;}
}
```

Une fois que vous avez saisi le code source, sauvegardez les modifications dans l'éditeur en appuyant sur CTRL-S, puis compilez en appuyant sur CTRL-B.

Je ne m'étendrai pas sur le code lui même qui est assez simple et d'un intérêt limité en lui-même (mais il faut bien commencer simplement). La compréhension du paramètrage qui va suivre à l'étape suivante est en revanche primordiale.

Etape 5 : Premières notions de paramétrage

Paramètrage intérieur et extérieur

Le gestionnaire Kassia peut prendre en compte des fichiers de paramétrage intérieurs comme extérieurs à l'application. Le paramètrage intérieur consiste en différents fichiers texte contenus à l'intérieur même du fichier JAR de l'application alors que le paramètrage extérieur consiste en des fichiers texte placés sur votre ordinateur. Ce deuxième paramétrage permet en fait d'altérer le premier paramétrage (on parle de paramétrage initial et de paramètrage secondaire) afin de modifier le comportement de l'application.

Pour le moment, nous ne nous intéresserons qu'au paramètrage intérieur. Les fichiers qui le composent sont considérées comme des ressources internes de l'application et doivent donc être placées dans le répertoire de source *resource*.

Resources internes pour le gestionnaire Kassia

Le gestionnaire Kassia est conçu pour rechercher toutes ses ressources internes dans le package *gus05.resource.gus.kassia*. Vous pouvez donc créer ce package dans le répertoire "resource" de la même manière que précédemment décrit (clic droit...)

Etape 6 : Les fichiers "prop" et "load"

Pour notre paramètrage, nous allons avoir besoin de 2 fichiers texte nommés "prop" et "load" (fichiers sans extension). Vous pouvez créer ces 2 fichiers dans le package *gus05.resource.gus.kassia* de la manière suivante : *clic droit / New / File*.

Le fichiers "prop" : il s'agit d'un fichier de type "properties" fournissant un ensemble de propriétés à l'application, pouvant être utilisées par le gestionnaire lui-même mais aussi par les entités.

Le fichier "load" : il s'agit d'un autre fichier de type "properties" qui indique au gestionnaire comme charger en mémoire les entités de l'application.

Pour ceux qui ne savent pas ce qu'est un fichier "properties", c'est un fichier texte ASCII muni a priori de l'extension "properties" et permettant de stocker un ensemble de propriétés texte sous la forme <key> = <value>. Ce format de fichier est très utilisé par le gestionnaire Kassia et de manière général dans les applications gus05. Il faut noter que pour des raisons d'encodage, tous les anti-slash apparaissant dans des clés ou valeurs doivent être dédoublés. "\" s'écrit donc "\\" dans les chemins d'accès notamment.

Etape 7 : Paramétrer l'entité

Paramétrer une entité consiste à mettre en place tout le paramètrage nécessaire pour permettre au gestionnaire de trouver et instancier la classe de l'entité sans encombre. Dans le cas le plus simple, il suffit de rajouter une ligne dans le fichier "load" pour indiquer au gestionnaire comment trouver la classe de cette entité.

Ouvrez le fichier *load* situé dans le package *gus05.resource.gus.kassia,* ajoutez-y la ligne suivante et sauvegardez les modifications (CTRL-S) :
charly.test.panel.green=GreenPanel*

Ce paramètrage explique au gestionnaire Kassia qu'il doit trouver une entité nommée *charly.test.panel.green* dans le package correspondant en choisissant la classe qui s'appelle *GreenPanel*. L'étoile ajoutée à la fin est une indication sur la manière dont l'entité doit être gérée, qui s'appelle la multiplicité de l'entité. Nous ne nous en occuperons pas pour le moment.

A présent, relancez l'application pour observer ce qu'il se passe. Puisque le "run configuration" a été crée à l'atelier précédent, il suffit de le lancer en cliquant sur l'icône en forme de triangle blanc dans un rond vert. Vous pouvez aussi pour cela, passer par le menu Run, ou plus simplement encore utiliser la commande clavier CTRL-F11.

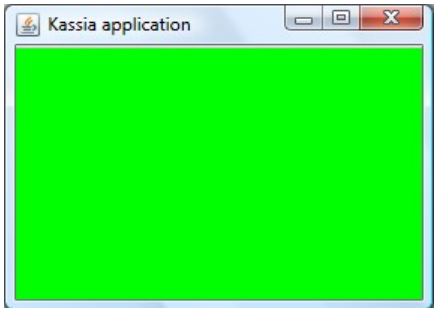
Bon, a priori rien n'a changé dans l'application. Mais en revanche, il y a un léger changement dans le log. A la ligne 39, le message indique "entity number = 1" alors qu'auparavant, il indiquait "No entity found". Cela signifie que notre entité est détectée et chargée en mémoire par le gestionnaire !

Etape 8 : Tester l'entité

Que reste-t-il à faire alors ? L'application a trouvé notre entité mais rien ne lui explique comme s'en servir. Pour la tester graphiquement, nous allons la prendre comme interface graphique principale de notre application.

Pour ce faire, ouvrez le fichier *prop* situé dans le package *gus05.resource.gus.kassia,* ajoutez-y la ligne suivante et sauvegardez les modifications (CTRL-S) :
app.maingui=charly.test.panel.green

Relancez une nouvelle fois l'application et si aucune erreur de paramétrage n'a été faite, vous devriez avoir votre entité affichée dans la fenêtre de l'application.



Félicitation, vous venez de tester votre première entité !!

ATELIER 4 : Les ressources : icônes et sons

Objectifs : Nous allons ajouter des ressources à notre application et développer des entités pour les visualiser. Nous verrons aussi comment ajouter du paramétrage pour personnaliser l'apparence de l'application.

Etapes :

1. Télécharger et installer des icônes et des sons
2. Changer l'icône, le titre et la taille de la fenêtre de votre application
3. Créer une nouvelle entité pour visualiser les icônes
4. Introduction à la classe Outside du framework
5. Une entité pour écouter les sons
6. Une entité pour faire la liste des propriétés

Dans cet atelier, nous allons commencer à utiliser des fichiers de ressource multimédia (icône et son), et progresser dans le paramétrage de notre application et dans la compréhension du framework (introduction de la classe *Outside*) pour développer de nouvelles entités plus intéressantes.

Etape 1 : Télécharger et installer des icônes et des sons

Nous allons récupérer 3 fichiers d'icône (.gif) et 2 fichiers de son (.wav) :

 **calc.gif**  **msn.gif**  **textEditor.gif**  **laughing.wav**  **meuh.wav**

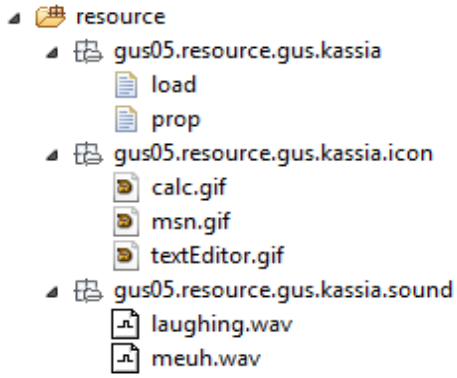
Ces fichiers sont stockés dans une archive zip que vous pouvez télécharger ci-dessous :

ressources.zip

Nous allons placer ces 5 ressources à l'intérieur de notre application, avec les fichiers de paramétrage "load" et "prop". Créez dans le répertoire *resource* les packages suivants :

- *gus05.resource.gus.kassia.icon*
- *gus05.resource.gus.kassia.sound*

Puis, en passant par l'explorateur Windows (si vous êtes sous Windows...), copiez les fichiers gif dans le répertoire de package "icon" et les fichiers wav dans le répertoire de package "sound". Enfin, revenez sous Eclipse et rafraichissez l'interface en appuyant sur F5. Vous devriez obtenir ceci :



Malheureusement ce n'est pas encore suffisant pour que le gestionnaire Kassia puisse trouver et charger ces ressources. Il a besoin dans chaque cas d'un fichier de listing qui lui spécifie les noms des fichiers présents dans les répertoires.

Vous allez donc créer dans chaque package un fichier "listing" dans lequel vous taperez les lignes suivantes :

Pour le fichier *listing* du package *gus05.resource.gus.kassia.icon* :
calc.gif
msn.gif
textEditor.gif

Pour le fichier *listing* du package *gus05.resource.gus.kassia.sound* :
laughing.wav
meuh.wav

Sauvegardez les modifications et relancez l'application.

La quatrième phase du log qui donne un récapitulatif de l'application s'est agrandie de 2 lignes supplémentaires. Je vous fournis ci-dessous l'extrait de mon log correspondant :

```
0041 20090831_081638 4 0 [M] KS4_start@presentApp()    . icons number = 3
0042 20090831_081638 4 0 [M] KS4_start@presentApp()    . sounds number = 2
```

Les 3 icônes et les 2 sons ont bien été chargés par l'application !

Etape 2 : Changer l'icône, le titre et la taille de la fenêtre de votre application

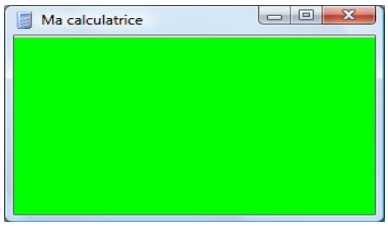
Nous avons dit au sujet du fichier de paramétrage *prop* qu'il fournit un ensemble de propriétés à l'application, pouvant être utilisées soit par le gestionnaire Kassia, soit par les entités. L'une des propriétés que Kassia prend en compte est *app.maingui*, qui permet de spécifier l'entité graphique responsable de l'interface graphique principale. Voyons à présent d'autres propriétés prises en compte par le gestionnaire.

Ouvrez le fichier *prop* et ajoutez-y les lignes suivantes :

```
app.icon=calc
app.title=Ma calculatrice
app.size=300 200
```

Sauvegardez les modifications et relancez l'application.

L'icône, le titre et la taille de la fenêtre ont changé conformément aux valeurs des propriétés ajoutées dans le fichier *prop*.



Au sujet de la propriété app.icon :

La valeur de la propriété correspond à la clé de stockage de la ressource d'icône dans l'application, c'est à dire le nom du fichier source sans l'extension ".gif". Le gestionnaire Kassia est capable de charger ainsi en mémoire des icônes à partir de fichiers gif ou de fichiers png.

Au sujet de la propriété app.title :

De la manière dont nous éditons actuellement le fichier prop, nous ne pouvons spécifier que des valeurs de propriété en texte simple (c'est à dire a priori en caractères latin éventuellement accentués mais sans caractères spéciaux, notamment le = et le /). Lorsque plus tard nous éditerons ce fichier avec un éditeur adapté aux fichiers properties, il sera possible de spécifier n'importe quel titre (y compris en écriture japonaise).

Au sujet de la propriété app.size :

Il existe plusieurs manières de spécifier la taille de la fenêtre de notre application. La manière que vous avons utilisé est la plus simple. Elle consiste à indiquer la largeur et la hauteur (en pixels) de la fenêtre (par exemple "app.size=700 600" pour une fenêtre de taille 700 x 600). Mais il existe aussi deux autres manières :

"app.size=pack" permet de compacter la fenêtre pour la mettre à la taille la plus petite possible acceptable par son contenu graphique (correspond à la méthode pack() dans la classe JFrame).

"app.size=screen" permet de mettre la fenêtre à une taille légèrement inférieure à votre écran (50 de moins en largeur et 100 de moins en hauteur).

Enfin, vous pouvez figer définitivement la taille en faisant précéder la valeur d'un astérisque. Dans ce cas, l'utilisateur ne pourra plus la redimensionner avec la souris. (par exemple "app.size=*200 200).

Etape 3 : Créer une nouvelle entité pour visualiser les icônes

Il est temps de créer une nouvelle entité qui va nous servir à visualiser l'ensemble des icônes qui sont chargées dans l'application : l'entité *charly.display.resource.icons*

Créez le package *gus05.entity.charly.display.resource.icons* dans le répertoire *entity* puis ajoutez-y une classe *DisplayIcons*.

Dans un premier temps, nous allons juste la doter d'un composant Swing JList comme interface graphique puis nous occuper de la paramétrer. Ensuite nous compléterons son code source pour faire afficher par le composant JList les icônes de l'application.

Tapez le code source de la classe comme ci-dessous puis sauvegardez et compilez (CTRL-S, CTRL-B)

```
package gus05.entity.charly.display.resource.icons;

import gus05.framework.core.Entity;
import gus05.framework.features.Graphic;

import javax.swing.JComponent;
import javax.swing.JList;

public class DisplayIcons implements Entity, Graphic {

	public String getName(){return "charly.display.resource.icons";}
	public String getCreationDate(){return "2009.08.31";}

	private JList list;

	public DisplayIcons()
	{
		list = new JList();
	}

	public JComponent gui()
	{return list;}
}
```

Après quoi, occupons-nous du paramétrage...

Ouvrez le fichier *load*, ajoutez-y la ligne suivante et sauvegardez les modifications (CTRL-S) :
charly.display.resource.icons=DisplayIcons*

Ouvrez le fichier *prop*, modifiez la valeur de la propriété *app.maingui* comme suit et sauvegardez les modifications (CTRL-S) :
app.maingui=charly.display.resource.icons

Remarque :

- **Un fichier** properties **ne peut pas contenir deux lignes avec la même clé de propriété néanmoins si vous souhaitez garder une trace de la ligne précédente, vous pouvez la mettre en commentaire plutôt que de la remplacer (faites-la commencer par un #).**

... et relançons l'application.

Le panneau vert a laissé la place à un panneau blanc (en fait, notre JList). Bien, voyons maintenant comment on peut améliorer ce résultat !

Etape 4 : Introduction à la classe Outside du framework

La classe *Outside* est la classe du framework utilisée par les entités pour communiquer avec le gestionnaire. Cette classe située dans le package *gus05.framework.core* offre aux entités 3 méthodes statiques qui sont :

- `public static Service service(Entity entity, String id) throws Exception`
- `public static Object resource(Entity entity, String id) throws Exception`
- `public static void err(Entity entity, String id, Exception e)`

Dans un premier temps, nous ne nous intéresserons qu'à la méthode *resource* qui permet à l'entité de récupérer un objet en provenance du gestionnaire. Les deux arguments passés à la méthode correspondent à l'entité elle-même et à une chaîne de caractères d'identification permettant à l'entité d'exprimer au gestionnaire ce qu'elle désire.

Dans le cas de l'entité *charly.display.resource.icons* nous sommes intéressés par l'objet Map contenant les icônes de l'application. Cet objet peut être récupéré grâce à l'id *app.gui.iconmap*, ce qui donne une utilisation de la classe *Outside* comme ceci :

```
Outside.resource(this,"app.gui.iconmap");
```

Cette méthode renvoie dans le cas présent un objet de type Map qu'il sera donc nécessaire de "caster" pour utiliser comme tel. Cet objet Map fournit pour chaque clé d'icône l'objet de type *javax.swing.Icon* correspondant.

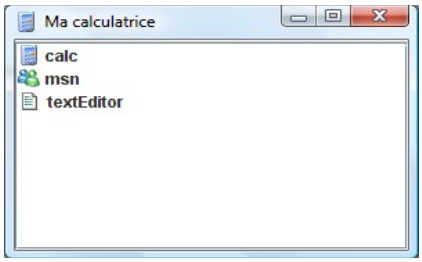
Après ces brèves explications, nous pouvons compléter la classe DisplayIcons pour lui donner son code source définitif présenté ci-dessous :

```
package gus05.entity.charly.display.resource.icons;

import java.awt.Component;
import java.util.Collections;
import java.util.Map;
import java.util.Vector;
import gus05.framework.core.Entity;
import gus05.framework.core.Outside;
import gus05.framework.features.Graphic;
import javax.swing.DefaultListCellRenderer;
import javax.swing.Icon;
import javax.swing.JComponent;
import javax.swing.JLabel;
import javax.swing.JList;

public class DisplayIcons implements Entity, Graphic {

      public String getName(){return "charly.display.resource.icons";}
```

```
	public String getCreationDate(){return "2009.08.31";}

	private JList list;
	private Map map;

	public DisplayIcons() throws Exception
	{
		map = (Map) Outside.resource(this,"app.gui.iconmap");

		Vector keys = new Vector(map.keySet());
		Collections.sort(keys);

		list = new JList(keys);
		list.setCellRenderer(new DefaultListCellRenderer(){
			public Component getListCellRendererComponent(JList list,Object
value,int index,boolean isSelected,boolean cellHasFocus)
			{
				JLabel comp =
(JLabel)super.getListCellRendererComponent(list,value,index,isSelected,cellHasFocus);
				comp.setIcon((Icon)map.get(value));
				return comp;
			}
		});
	}

	public JComponent gui()
	{return list;}
}
```

Les développeurs Java peu habitués à l'API Swing et notamment aux "renderers" seront sans doute un peu perdus. Je vous encourage à parcourir quelques tutoriaux sur Internet qui sauront vous expliquer mieux que moi les techniques de programmation graphique en Java.

Une fois la classe compilée, vous pouvez relancer l'application. Vous devriez obtenir ceci :



Etape 5 : Une entité pour écouter les sons

Notre troisième entité va nous permettre de faire la liste de l'ensemble des sons chargés dans l'application et de les écouter : *charly.display.resource.sounds*

De la même manière que précédemment, créez le package *gus05.entity.charly.display.resource.sounds* et la classe *DisplaySounds*, puis saisissez le code source ci-dessous :

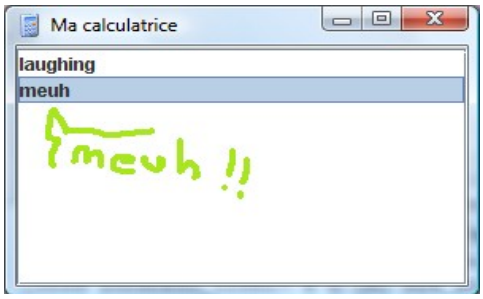
```
package gus05.entity.charly.display.resource.sounds;

import java.applet.AudioClip;
```

```
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.util.Collections;
import java.util.Map;
import java.util.Vector;
import gus05.framework.core.Entity;
import gus05.framework.core.Outside;
import gus05.framework.features.Graphic;
import javax.swing.JComponent;
import javax.swing.JList;

public class DisplaySounds implements Entity, Graphic {

	public String getName(){return "charly.display.resource.sounds";}
	public String getCreationDate(){return "2009.08.31";}

	private JList list;
	private Map map;

	public DisplaySounds() throws Exception
	{
		map = (Map) Outside.resource(this,"app.soundmap");

		Vector keys = new Vector(map.keySet());
		Collections.sort(keys);

		list = new JList(keys);
		list.addMouseListener(new MouseAdapter(){
			public void mousePressed(MouseEvent e){play();}
		});
	}

	public JComponent gui()
	{return list;}

	private void play()
	{
		if(list.isSelectionEmpty())return;
		String key = (String)list.getSelectedValue();
		AudioClip clip = (AudioClip)map.get(key);
		clip.play();
	}
}
```

Ajoutez la ligne suivante dans *load* :
charly.display.resource.sounds=DisplaySounds*

Et modifiez la propriété *app.maingui* :
app.maingui=charly.display.resource.sounds

Enfin, relancez l'application :



Etape 6 : Une entité pour faire la liste des propriétés

Pour la quatrième entité, je ne vous donnerai que le code source. Vous devriez normalement savoir quoi faire ! ^^

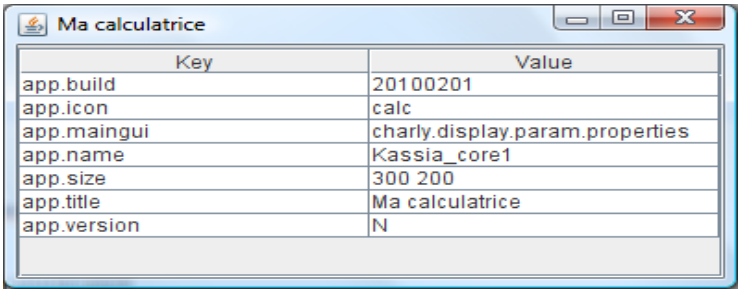
```
package gus05.entity.charly.display.param.properties;

import gus05.framework.core.Entity;
import gus05.framework.core.Outside;
import gus05.framework.features.Graphic;

import java.util.Collections;
import java.util.Map;
import java.util.Vector;
import javax.swing.JComponent;
import javax.swing.JTable;
import javax.swing.table.DefaultTableModel;

public class DisplayProperties implements Entity, Graphic {

	public String getName(){return "charly.display.param.properties";}
	public String getCreationDate(){return "2009.08.31";}

	private JTable table;
	private Map map;

	public DisplayProperties() throws Exception
	{
		map = (Map) Outside.resource(this,"internal.propmap");

		Vector keys = new Vector(map.keySet());
		Collections.sort(keys);

		String[][] content = new String[map.size()][2];
		for(int i=0;i<map.size();i++)
		{
			String key = (String) keys.get(i);
			String value = (String) map.get(key);

			content[i][0] = key;
			content[i][1] = value;
		}
		String[] columns = new String[]{"Key","Value"};

		table = new JTable(new DefaultTableModel(content,columns));
	}

	public JComponent gui()
	{return table;}
}
```

Remarque :

- **Le seul code source d'une entité est suffisant pour vous permettre de créer, coder, paramétrer et tester (graphiquement) l'entité sans erreur. Si vous obtenez une erreur au lancement de votre application, c'est que vous n'avez pas bien fait quelque chose parmi :**

1. **Le choix du nom de package pour l'entité**
2. **La recopie du code source**
3. **Le paramétrage dans le fichier load**
4. **La modification de la propriété app.maingui**

Voici le résultat auquel vous devez vous attendre pour cette quatrième entité.



L'entité *charly.display.param.properties* nous fournit les clés et valeurs de propriété de l'application. Nous pouvons constater que l'application contient 7 propriétés parmi lesquels 3 semblent venir de nulle part : *app.build*, *app.name* et *app.version*.

En fait, le gestionnaire Kassia initialise par défaut 4 propriétés qui sont :

app.name	Kassia_core1
app.title	Kassia application
app.version	N
app.build	20100201

Il est possible de modifier les valeurs de chacune des ces propriétés en les ajoutant dans le fichier *prop* mais on ne peut pas les supprimer. Il s'agit des propriétés de base de n'importe quelle application basée sur le gestionnaire Kassia, permettant notamment de l'identifier. Nous allons les commenter brièvement.

app.name : fournit le nom de l'application, considéré comme un identifiant unique plus qu'un nom d'usage.

app.title : fournit le titre de l'application tel qu'il apparaît dans le haut de sa fenêtre, utilisé aussi comme nom d'usage

app.version : fournit la version de l'application. Cette information est une indication parlante pour l'utilisateur afin de connaître l'état d'avancement de l'application mais ne peut pas être utilisé pour l'identifier formellement. La valeur par défaut "N" indique que l'application constituée uniquement du gestionnaire Kassia n'est pas versionnable.

app.build : fournit la date du jour (au format aaaammjj) où l'application a été déployée. Contrairement à *app.version*, cette information permet d'identifier formellement la version "technique" du fichier JAR. La valeur par défaut correspond à la date de dernière modification dans le gestionnaire Kassia. (01 février 2010 à l'heure ou j'écris)

ATELIER 5 : Le multi-affichage d'entités

Objectifs : Nous montrons ici comment il est possible d'afficher ensemble plusieurs entités graphiques. Cet atelier permettra d'introduire les notions de service et de mapping.

Etapes :

1. Qu'est-ce qu'un service ?
2. Créer et tester une entité multi-affichage
3. Réfléchir sur les appels des entités
4. La notion de mapping
5. Mise en pratique

Nous avons créé jusqu'à présent 4 entités graphiques que nous avons fait afficher successivement en changeant à chaque fois la valeur de la propriété *app.maingui*. Le but de ce 5ème atelier est de faire afficher simultanément les interfaces des 4 entités précédentes, disposées en damier 2 x 2. Pour cela, une entité doit pouvoir être capable de manipuler d'autres entités, et c'est ce qui va nous amener à introduire les services.

Une fois notre entité "multi-affichage" créée et testée, nous approfondirons la notion d'appel des entités vers le gestionnaire et la manière dont ces appels peuvent être paramétrés, grâce à ce qu'on appelle le mapping.

Etape 1 : Qu'est-ce qu'un service ?

Un service est un objet de type *Service* (interface *gus05.framework.core.Service*) qui est utilisé par les entités pour accèder aux caractéristiques d'autres entités. A ce titre, un service peut être notamment considéré comme un objet de type *Graphic* et donc fournir une interface par la méthode *public JComponent gui()*. Nous allons nous en servir dans cet atelier pour faire afficher par une entité les interfaces graphiques d'autres entités.

Un objet Service s'obtient par la classe *Outside* grâce à la méthode publique *service* et la manière la plus simple d'obtenir le service issu d'une autre entité <entity-name> est :

```
Outside.service(this,"<entity-name>");
```

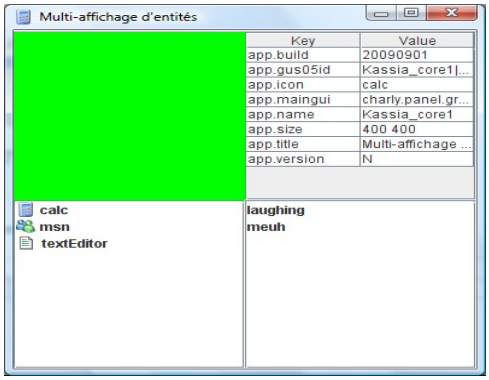
Etape 2 : Créer et tester une entité multi-affichage

Créez, paramétrez et testez l'entité *charly.panel.grid22*. Dans cet exemple, nous prendrons comme taille pour la fenêtre de notre application 400 x 400 et comme titre "Multi-affichage d'entités"

```
package gus05.entity.charly.panel.grid22;

import gus05.framework.core.Entity;
import gus05.framework.core.Outside;
import gus05.framework.core.Service;
import gus05.framework.features.Graphic;
```

```
import java.awt.GridLayout;
import javax.swing.JComponent;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.Scrollable;

public class PanelGrid22 implements Entity, Graphic {

	public String getName(){return "charly.panel.grid22";}
	public String getCreationDate(){return "2009.08.31";}

	private Service gui1;
	private Service gui2;
	private Service gui3;
	private Service gui4;

	private JPanel panel;

	public PanelGrid22() throws Exception
	{
		gui1 = Outside.service(this,"charly.test.panel.green");
		gui2 = Outside.service(this,"charly.display.param.properties");
		gui3 = Outside.service(this,"charly.display.resource.icons");
		gui4 = Outside.service(this,"charly.display.resource.sounds");

		panel = new JPanel(new GridLayout(2,2));
		panel.add(buildComp(gui1));
		panel.add(buildComp(gui2));
		panel.add(buildComp(gui3));
		panel.add(buildComp(gui4));
	}

	public JComponent gui()
	{return panel;}

	private JComponent buildComp(Service s)
	{
		JComponent comp = s.gui();
		if(comp instanceof Scrollable)
			return new JScrollPane(comp);
		return comp;
	}
}
```

Le résultat graphique reprend les interfaces de nos 4 entités précédentes :



Etape 3 : Réfléchir sur les appels d'entité

Excepté la première entité, toutes les autres entités que nous avons développé font des appels vers le gestionnaire par la classe *Outside*. Nous récapitulons ci-dessous tous ces appels :

Les appels de ressources :

- `Outside.resource(this,"app.gui.iconmap");`
- `Outside.resource(this,"app.soundmap");`
- `Outside.resource(this,"internal.propmap");`

Les appels de services :

- `Outside.service(this,"charly.test.panel.green");`
- `Outside.service(this,"charly.display.param.properties");`
- `Outside.service(this,"charly.display.resource.icons");`
- `Outside.service(this,"charly.display.resource.sounds");`

Les objets renvoyés par les méthodes *resource* et *service* de *Outside* dépendent dans ces exemples des identifiants passés en paramètre qui doivent être connus du gestionnaire. Dans le cas contraire, une exception est génerée, qui entraine l'échec de la création de l'entité appelante.

En fait, les méthodes *resource* et *service* de *Outside* mettent en œuvre les mêmes mécanismes de génération d'objet. La seule différence réside dans le fait que dans le cas de l'appel service, l'objet généré est préalablement "enveloppé" dans un objet de type Service avant d'être renvoyé. En pratique l'objet qui se retrouve caché à l'interieur d'un service est presque toujours une instance d'entité.

Comment se fait la génération de l'objet, qu'il soit renvoyé tel quel (dans le cas d'un appel ressource) ou qu'il soit renvoyé sous la forme d'un service (dans le cas d'un appel service) ? Le gestionnaire Kassia reçoit une règle de génération qu'il tente d'interpréter. A première vue cette règle de génération correspond à l'identifiant provenant de l'entité appelante mais en réalité c'est plus complexe que ça. Nous allons voir qu'il est possible d'introduire un mapping entre les identifiants d'appel et les règles de génération, rendant possible le paramétrage des appels qui dès lors ne sont plus fixés en dur dans le code source des entités.

Etape 4 : La notion de mapping

Le mapping est une map particulière qui donne la possibilité d'associer une règle de génération spécifique à chaque appel d'entité. Voyons concrètement ce que cela donne avant de poursuivre les explications.

Nous allons tout d'abord remplacer dans le code source de l'entité *charly.panel.grid22* les identifiants d'appels par des mots plus quelconques : "gui1", "gui2", "gui3", "gui4". De cette manière, nous n'aurons plus l'impression que les 4 objets service sont fixés à l'avance.

```
gui1 = Outside.service(this,"gui1");
gui2 = Outside.service(this,"gui2");
gui3 = Outside.service(this,"gui3");
gui4 = Outside.service(this,"gui4");
```

Bien sûr, si à ce stade vous tentez de lancer l'application, le gestionnaire va générer une exception dès l'appel de service "gui1" et tenter de vous expliquer dans le log qu'il ignore ce que signifie ce terme.

Créez un fichier "mapping" dans le package *gus05.resource.gus.kassia*, dans lequel vous saisirez le texte suivant :
charly.panel.grid22@gui1=charly.test.panel.green
charly.panel.grid22@gui2=charly.display.param.properties
charly.panel.grid22@gui3=charly.display.resource.icons
charly.panel.grid22@gui4=charly.display.resource.sounds

Maintenant vous pouvez relancer l'application et vérifier qu'elle fonctionne bien.

Le mapping est donc stocké dans un fichier properties du nom de "mapping", permettant de spécifier des relations entre appels d'entité et règles de génération.

De manière simple, un appel se note de la manière suivante : <entity-name>@<call-ID>. Quant aux règles de génération, nous retiendrons pour le moment qu'il peut s'agir de noms d'entité (dans le cas d'appels de service) et de mots-clés connus du gestionnaire Kassia (dans le cas d'appels de ressources) même si nous découvrirons plus tard, qu'elles peuvent devenir beaucoup plus complexes.

Etape 5 : Mise en pratique

Vous pouvez à loisir manipuler le mapping en mettant à chaque valeur n'importe laquelle des 4 entités graphiques disponibles et tester les modifications. Vous pourrez ainsi changer les positions dans la grille et même afficher plusieurs fois la même entité.

ATELIER 6 : Déployer son application

Objectifs : Cet atelier explique comment créer un fichier JAR sous Eclipse pour obtenir une application Java directement utilisable. On appelle cette opération le déploiement.

Etapes :

1. Exporter le projet Eclipse vers un fichier JAR
2. Lancer le fichier JAR
3. Tester le fichier de déploiement deploy.jardesc

Cet atelier très simple va consister en une succession de manipulations sous Eclipse pour créer un fichier JAR à partir de notre projet et enregistrer la configuration de création de JAR dans un fichier spécifique à Eclipse afin de répéter par la suite facilement l'opération à chaque fois qu'on souhaitera redéployer notre application.

Etape 1 : Exporter le projet Eclipse vers un fichier JAR

1. Faites un clic droit dans l'arborescence d'Eclipse sur l'icône du projet *gus05_ateliers*, et dans le menu contextuel, choisissez : *Exporter...*
2. Dans l'arborescence de la boite de dialogue, choisissez JAR file (dans la catégorie Java) et cliquez sur le bouton *Next*.
3. Au dialogue suivant, choisissez juste l'emplacement et le nom (par exemple *KassiaAtelier.jar*) du futur fichier JAR et cliquez sur le bouton *Next*.
4. Au dialogue suivant, cochez la case "Save the description of this JAR in the workspace et saisissez comme "Description file" : */gus05_ateliers/deploy.jardesc*. Cliquez sur le bouton *Next*.
5. Enfin, au dialogue suivant, spécifiez comme "Main class", la classe du gestionnaire Kassia qui contient la méthode main : *gus05.manager.gus.kassia.layer2.KassiaMain*, et cliquez sur le bouton *Finish*.

Eclipse génère alors le fichier JAR à l'emplacement que vous avez spécifié et le fichier deploy.jardesc à la racine du projet.

Etape 2 : Lancer le fichier JAR

Rendez-vous à l'emplacement ou le fichier JAR est sencé avoir été généré, vérifiez qu'il est bien là et lancez le en double-cliquant dessus. Normalement l'application doit se lancer normalement.

Remarque :

- **Il se peut que Windows configure bizarrement le programme par défaut qui lance les fichiers JAR (par exemple un programme d'ouverture de fichiers ZIP) alors que ceux-ci sont normalement pris en charge par la machine virtuelle de Java.**
- **Dans ce cas, faites un clic droit sur le JAR et choisissez : Ouvrir avec / Choisir le**

programme par défaut ... Alors, dans la liste des programmes recommandés, doit apparaître "Java(TM) Plateform SE binary". Selectionnez le, n'oubliez surtout pas de cocher la case "Toujours utiliser le programme sélectionné pour ouvrir ce type de fichier" et cliquez sur le bouton "OK".

Etape 3 : Tester le fichier de déploiement deploy.jardesc

L'intérêt du fichier *deploy.jardesc* est de refaire toute l'opération de création de JAR en un seul clic (3 en fait si on compte bien).

Pour cela, faites un clic droit sur le fichier et choisissez dans le menu contextuel : *Create JAR*. Si le fichier JAR existe déjà, Eclipse vous demande de confirmer que vous le remplacez et vous répondez oui.

ATELIER 7 : Le paramètrage extérieur : configurer un répertoire root

Objectifs : Lorsqu'on dispose d'une application déployée, il est possible de lui ajouter du paramétrage en spécifiant un répertoire root dans lequel elle va chercher des fichiers.

Etapes :

1. Créer un répertoire root et un script le lancement
2. La base du paramétrage exterieur : le fichier app.properties
3. Donner accès à d'autres répertoires et fichiers
4. Les séquences de chargement
5. Charger des icônes ou des sons supplémentaires
6. Charger du mapping supplémentaire
7. Charger des entités supplémentaires
8. Réorganisation de notre paramétrage

A l'étape 5 de l'atelier 3, nous avions rapidement évoqué le paramétrage extérieur d'une application comme pouvant altérer son paramètrage intérieur. A présent que nous disposons d'une application sous forme de fichier JAR, nous allons pouvoir étudier comment mettre en place un tel paramètrage et notamment comment altérer les ressources ou le mapping de l'application, ou même comment ajouter et paramétrer de nouvelles entités.

Etape 1 : Créer un répertoire root et un script le lancement

Tout d'abord, il est nécessaire de préciser au démarrage de l'application un répertoire root dans lequel celle-ci pourra rechercher son paramétrage extérieur. Pour cela, il va vous falloir créer un script de lancement qui passe en argument la valeur suivante : "root=<chemin-d'accès-de-votre-root>"

A coté du fichier *KassiaAtelier.jar* créez un fichier texte que vous nommerez *launch.bat*. Ouvrez ce fichier avec notepad et ajoutez-y le texte ci-dessous (en adaptant la valeur à votre chemin d'accès):

java -jar KassiaAtelier.jar root=C:\Mon Java\root1
Dans cet exemple, le répertoire root choisi est : "C:\Mon Java\root1"

A propos des arguments passés au démarrage de l'application, le gestionnaire Kassia prend par convention le point virgule comme délimiteur d'arguments (conserver l'espace qui est le délimiteur de la JVM imposait des restrictions sur les chemins d'accès). Si par exemple vous souhaitez ajouter l'argument sysout qui permet d'imprimer les messages de log vers la sortie standard de l'application, le script deviendra alors :

java -jar KassiaAtelier.jar root=C:\Mon Java\root1;sysout

Etape 2 : La base du paramétrage exterieur : le fichier app.properties

Le répertoire root étant fixé, nous allons pouvoir y ajouter des fichiers de paramètrage. De lui même, le gestionnaire Kassia n'est capable d'y trouver qu'un fichier : le fichier *app.properties* qui permet d'altérer ses propriétés. Amusons-nous par exemple à modifier l'icône et le titre de l'application.

Créez un fichier texte *app.properties* dans le répertoire root et ouvrez-le avec notepad. Ajoutez-y ensuite le texte suivant :

```
app.title=Messenger MSN
app.icon=msn
```

Après avoir sauvergardé, executez le fichier JAR.

L'icône et le titre ont changé en conséquence.

Etape 3 : Donner accès à d'autres répertoires et fichiers

Avant d'aller plus loin, il est nécessaire de préciser la manière dont les notions de fichier, répertoire et chemin d'accès sont prise en compte par le gestionnaire Kassia. Un chemin d'accès (relatif ou absolu) est considéré comme un "emplacement" sur votre ordinateur qui peut correspondre le cas échéant à un fichier, à un répertoire (emplacement réalisé) ou à "rien du tout" (emplacement non réalisé). On dira qu'un emplacement non réalisé se réalisé lorsqu'il devient un fichier ou un répertoire. Bref, on emploiera par la suite les termes "chemin d'accès" ou "emplacement" pour décrire soit un fichier, soit un répertoire, soit un emplacement non réalisé succeptible de devenir un fichier ou un répertoire, ce qui correspond finalement à la définition d'un objet Java de type *java.io.File*.

Pour que le gestionnaire Kassia puisse accéder à des emplacements (pour quelque usage que ce soit : paramétrage, stockage de données, fichier de log, icônes...), il est nécessaire de lui ajouter de nouvelles propriétés qu'on appelle des pathIDs et dont la clé commence par "path.".

Il est possible de définir un pathID à partir d'un autre pathID, par exemple :
path.logfile=<path.rootdir>\\KassiaAtelier.log

Ainsi, lorsque le gestionnaire Kassia cherche à résoudre un pathID (trouver son emplacement), il doit d'abord résoudre le pathID qui intervient éventuellement dans sa valeur. Il se trouve que dans notre exemple, le pathID *path.rootdir* est connu par défaut puisqu'il correspond au répertoire root spécifié en argument dans notre script de lancement.

Donc la résolution du pathID *path.logfile* fournit l'emplacement *KassiaAtelier.log* à la racine du répertoire *root*, qui est un emplacement encore non réalisé. A quoi cela sert-il me direz-vous ? Je n'ai pas choisi cet exemple par hasard. Il se trouve que le gestionnaire Kassia considère par convention le pathID *path.logfile* comme l'emplacement pour son fichier de log lorsque celui-ci est défini.

Editez le fichier texte *app.properties* et ajoutez y la propriété :
path.logfile=<path.rootdir>\\KassiaAtelier.log

Relancez l'application.

Vous devriez voir apparaître un fichier *KassiaAtelier.log* à l'intérieur du répertoire root, fichier dans lequel l'application écrit le log au fur et à mesure que des messages sont émis. Ceci vous permettra de débugger l'application de manière plus confortable qu'avec la console DOS d'execution du script (sortie standard du programme qui affiche le log si vous spécifiez l'argument *sysout*).

Etape 4 : Les séquences de chargement

Il est possible d'altérer le chargement des ressources, des entités ou du mapping de l'application en ajoutant dans le fichier *app.properties* des propriétés de séquence de chargement. De telles propriétés commencent toujours par "sequence." et leurs valeurs se composent d'une suite de pathIDs séparés par des points virgule.

Au moment du chargement d'un certain type (icône, son, mapping..). Le gestionnaire Kassia commence toujours par charger les ressources internes au fichier JAR, ensuite il parcourt la séquence de chargement (si elle existe) dans l'ordre en traitant les données relatives à chaque pathID. En cas de conflit de clé de stockage, la donnée est toujours rechargée, remplaçant ainsi la précédente par la nouvelle.

Voyons quelques exemples de séquence de chargement ci-dessous.

Etape 5 : Charger des icônes ou des sons supplémentaires

Pour charger des icônes supplémentaires, créez par exemple un répertoire "icon" dans votre répertoire root et ajoutez-y vos propres fichiers gif ou png. Ensuite, ajoutez dans le fichier *app.properties* les propriétés suivantes :
path.icondir=<path.rootdir>\\icon
sequence.icon=path.icondir

De même pour charger des sons supplémentaires, créez par exemple un répertoire "sound" dans votre répertoire root et ajoutez-y vos propres fichiers wav. Ensuite, ajoutez dans le fichier *app.properties* les propriétés suivantes :
path.sounddir=<path.rootdir>\\sound
sequence.sound=path.sounddir

Etape 6: Charger du mapping supplémentaire

Pour charger du mapping supplémentaire, créez par exemple un fichier "mapping.properties" dans votre répertoire root dans lequel vous enregistrez vos nouvelles règles de mapping. Ensuite, ajoutez dans le fichier *app.properties* les propriétés suivantes :
path.mappingfile=<path.rootdir>\\mapping.properties
sequence.mapping=path.mappingfile

Voici ci-dessous un aperçu complet de la manière dont j'ai personnellement mis en pratique le début de cet atelier sur mon ordinateur. Cela vous permettra d'y voir plus clair dans le cas où vous seriez perdu, mais reste à titre d'exemple. Vous êtes bien sûr libre de prendre vos propres ressources et de fixer votre propre mapping.

Mon répertoire de travail

- ATELIER
  - KassiaAtelier.jar
  - launch.bat
  - root1
    - app.properties
    - KassiaAtelier.log
    - mapping.properties
    - icon
      - FLAG_fr.png
      - FLAG_ja.png
    - sound
      - cuckoo.wav

Fichier launch.bat

```
java -jar KassiaAtelier.jar root=root1;sysout
```

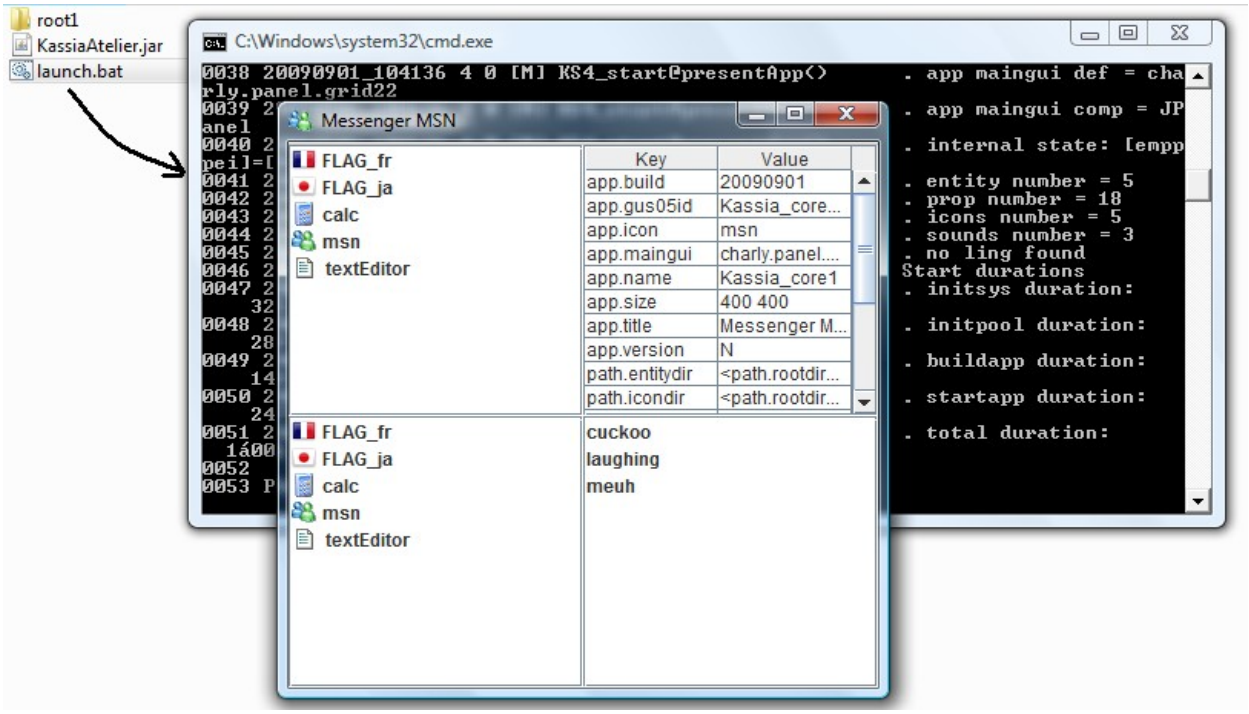
Fichier app.properties

```
app.title=Messenger MSN
app.icon=msn
path.logfile=<path.rootdir>\\KassiaAtelier.log

path.icondir=<path.rootdir>\\icon
sequence.icon=path.icondir

path.sounddir=<path.rootdir>\\sound
sequence.sound=path.sounddir

path.mappingfile=<path.rootdir>\\mapping.properties
sequence.mapping=path.mappingfile
```

Fichier mapping.properties

```
charly.panel.grid22@gui1=charly.display.resource.icons
```

Lancement de l'application



A REFAIRE (la propriété ap.gus05id a disparu)

Etape 7 : Charger des entités supplémentaires

De même que l'application a été capable de charger des icônes et des sons extérieurs à son fichier JAR, elle est aussi capable de charger des entités qui lui sont extérieures. Ce chargement à de plus l'avantage d'être dynamique, ce qui fait que le gestionnaire ne charge en mémoire que les entités dont il a besoin, au moment ou il en a besoin.

Une entité peut être déployée sous la forme d'un fichier JAR tout comme une application, ce qui rend cette entité directement utilisable par toute application basée sur le gestionnaire Kassia. Si vous souhaitez déployer vos propres entités, il vous suffit de faire un clic droit sur le package de celle-ci et de choisir dans le menu contextuel : *Export...* Vous choisissez alors d'exporter au format JAR et vous saisissez le nom du fichier, lequel doit impérativement être le même que le nom de l'entité. Enfin, vous cliquez sur le bouton *Finish*. Si par exemple vous déployez l'entité *charly.test.panel.green*, vous obtiendrez un fichier JAR : *charly.test.panel.green.jar*

Dans l'immédiat, il ne vous sera pas forcément utile de déployer les entités d'exemple de ces ateliers mais si vous êtes amené à développer vos propres entités et que vous souhaitez en faire profiter d'autres, alors déployer vos entités vous permettra de les échanger aisément. En revanche, je vais pouvoir vous fournir quelques entités que j'ai personnellement développé afin que vous puissiez les utiliser.

Téléchargez et dézippez l'archive zip accessible en ligne ci-dessous :

entity.zip

Cette archive contient 5 fichiers JAR correspondant à 5 entités que j'ai développé :

- *gus.graphic.panel.screen.animation.test2*
- *gus.management.summary.icon4*
- *gus.management.summary.systemenv2*
- *gus.time.support.clock.gui2*
- *gus.time.support.clock*

Pour charger ces entités supplémentaires, créez par exemple un répertoire "entity" dans votre répertoire root et ajoutez-y les 5 fichiers JAR. Ensuite, ajoutez dans le fichier *app.properties* les propriétés suivantes :

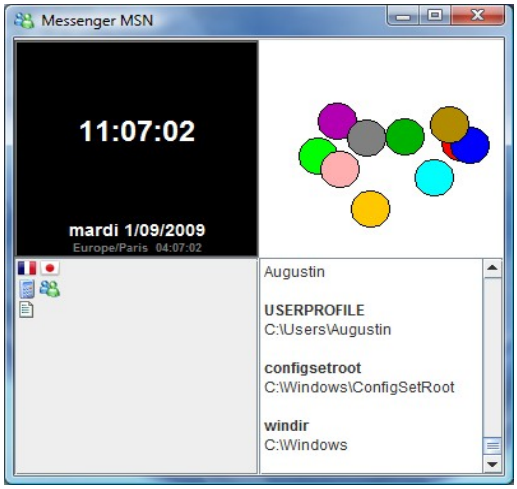
```
path.entitydir=<path.rootdir>\\entity
sequence.findentity=path.entitydir
```

Si vous relancez l'application maintenant, vous constaterez en lisant le fichier de log que le nombre d'entité chargé au démarrage est toujours 5, soit seulement les entités internes. Le chargement des entités externes, même s'il est correctement paramétré, ne se fera pas si l'application n'a pas besoin des entités, mais uniquement si celles-ci sont sollicitées.

A présent, éditez le fichier *mapping.properties* comme suit :

```
charly.panel.grid22@gui1=gus.time.support.clock.gui2
charly.panel.grid22@gui2=gus.graphic.panel.screen.animation.test2
charly.panel.grid22@gui3=gus.management.summary.icon4
charly.panel.grid22@gui4=gus.management.summary.systemenv2
```

Après avoir sauvegardé, relancez l'application et ...



C'est toujours la même entité principale qui s'affiche sous vos yeux (l'entité *charly.panel.grid22*) mais l'apparence graphique n'a rien à voir avec ce que vous avez développé dans les ateliers précédents. Il est temps que je vous présente mes entités.

gus.graphic.panel.screen.animation.test2
Une entité graphique fun, qui fait rebondir des ballons ^^

gus.management.summary.icon4
Cette entité a la même utilité que *charly.display.resource.icons* mais présente les icônes différemment.

gus.management.summary.systemenv2
Cette entité présente l'ensemble des propriétés de *System.getenv()*.

gus.time.support.clock.gui2
Cette entité est une horloge. Il faut noter que tout comme *charly.panel.grid22* cette entité est dépendante d'autres entités, en l'occurrence de *gus.time.support.clock*

gus.time.support.clock
Il s'agit d'une entité non graphique, dont la présence est nécessaire pour pouvoir utiliser l'horloge ci-dessus.

Etape 8 : Réorganisation de notre paramétrage

Le répertoire root de paramétrage externe est bien sûr utile lorsqu'on dispose d'une application déployée puisqu'il est le seul moyen d'en modifier le fonctionnement, mais il est aussi utile dans le cas d'une application en cours de développement, c'est à dire sous forme de projet Eclipse. Il permet d'ajouter des entités externes mais aussi d'ajouter des ressources (par exemple icônes ou sons) sans avoir à se soucier de maintenir le fichier de listing correspondant. Le plus efficace va être de combiner paramétrage interne et paramètrage externe.

Comme paramétrage interne, nous conserverons donc les fichiers *load*, *mapping* et *prop* du package *gus05.resource.gus.kassia* (ainsi que d'autres fichiers du même genre que nous n'avons pas encore introduit, je pense notamment au fichier *start* que vous découvrirez bientôt). Quant aux répertoires *icon et sound* contenant respectivement les fichiers d'icône et de son, nous allons les transférer dans un répertoire *root* qu'il va d'abord nous falloir créer au sein de notre projet Eclipse. Dans l'atelier 2, nous avions crée 4 répertoires de sources, mais à présent il s'agit de créer un répertoire de base. Nous l'appelerons "root".

Faîtes un clic droit sur le projet *gus05_ateliers* et dans le menu contextuel, choisissez : New / Folder. Une fois le répertoire "root" crée, rendez-vous avec votre explorateur Windows dans votre workspace puis dans votre répertoire projet et enfin dans votre répertoire resource. Dans son arborescence, déplacez les sous-répertoires *icon* et *sound* vers le répertoire *root* nouvellement crée. Sous Eclipse, vous pouvez à présent rafraichir votre projet avec la touche F5 et constater que "resource" ne contient plus qu'un seul package alors que "root" contient désormais "icon" et "sound". Vous pouvez d'ailleurs supprimer dans ces deux répertoires les deux fichiers listing qui ne nous seront plus d'aucune utilité. Ensuite, créez un sous-répertoire *entity* à coté de *icon* et *sound*, dans lequel vous pourrez copier l'ensemble des fichiers jar d'entité que vous avons utilisé à l'étape précédente.

Il nous reste maintenant à paramétrer le fichier *prop* et eventuellement ajouter des arguments à la configuration de lancement pour que les répertoires *root*, *icon*, *sound* et *entity* soient trouvés par l'application.

Sachant que le pathID *path.sys.user.dir* est reconnu par défaut comme étant le répertoire à partir duquel est lancé l'application (nous y reviendrons à l'atelier suivant consacré aux pathIDs), une manière efficace de définir l'emplacement *root* est :
path.rootdir=<path.sys.user.dir>\\root

L'emplacement *root* peut être indiqué dans les arguments de lancement de l'application de la même manière qu'avec une application déployée. Pour cela, vous pouvez éditer la configuration de lancement et fixer la valeur du champ "Program arguments" comme ceci :
sysout;root=<path.sys.user.dir>\root

Mais une manière encore plus simple est de l'indiquer directement dans le fichier *prop* de notre projet (ainsi d'ailleurs que tous les autres pathIDs). C'est ce que nous allons faire.

En reprenant les différentes propriétés que nous avons spécifié dans notre paramètrage externe tout à l'heure, voici ci-dessous ce que vous devriez obtenir pour votre fichier *prop*. Je vous affiche aussi les autres fichiers pour que vous puissiez vérifier que votre paramètrage interne est bien le même.

Fichier prop

```
app.maingui=charly.panel.grid22
app.icon=textEditor
app.title=Application test
app.size=900 600
path.rootdir=<path.sys.user.dir>\\root
path.icondir=<path.rootdir>\\icon
path.sounddir=<path.rootdir>\\sound
path.entitydir=<path.rootdir>\\entity
path.logfile=<path.rootdir>\\KassiaAtelier.log
sequence.icon=path.icondir
sequence.sound=path.sounddir
sequence.findentity=path.entitydir
```

Fichier load

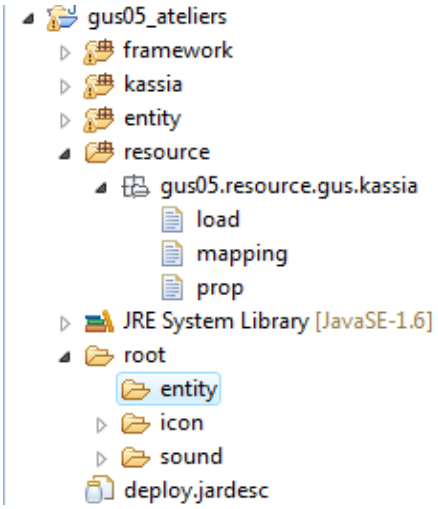
```
charly.test.panel.green=GreenPanel*
charly.display.resource.icons=DisplayIcons*
charly.display.resource.sounds=DisplaySounds*
charly.display.param.properties=DisplayProperties*
charly.panel.grid22=PanelGrid22*
```

Fichier mapping

```
charly.panel.grid22@gui1=charly.test.panel.green
charly.panel.grid22@gui2=charly.display.param.properties
charly.panel.grid22@gui3=charly.display.resource.icons
charly.panel.grid22@gui4=charly.display.resource.sounds
```

A priori, nous ne devrions plus nous resservir du répertoire root que nous avions crée à l'extérieur du projet Eclipse pour y tester notre fichier JAR. Vous pouvez donc supprimer ce répertoire ainsi que le script de lancement *launch.bat* et le fichier *KassiaAtelier.jar*.

Voici à quoi ressemble désormais notre projet Eclipse, qui va nous permettre de poursuivre notre découverte du framework gus05 dans les ateliers suivants.



ATELIER 8 : Paramètrer l'accès aux ressources extérieures

Objectif : L'accès aux fichiers et répertoires extérieurs à l'application se paramètre au moyen des pathIDs. Nous créerons une entité graphique pour visualiser ce paramètrage.

Etapes :

1. Une entité pour visualiser les pathIDs
2. Les pathIDs définis par le système
3. Les pathIDs définis par le paramétrage
4. Les mécanismes de résolution des pathIDs
5. La redirection de répertoire

Nous avons introduit les pathIDs à l'atelier 7. Ce 8ème atelier assez technique vise à faire le tour de la question en répondant aux questions suivantes :

- Comment sont gérés les pathIDs dans le gestionnaire ?
- Y a-t-il des pathIDs pré-existants ?
- Comment les paramétrer finement ?

Etape 1 : Une entité pour visualiser les pathIDs

Une fois résolus, les pathIDs sont stockés par le gestionnaire Kassia dans une map qui associe à chaque pathID l'objet File lui correspondant. Cette map peut être récupérée comme ressource grâce au mot-clé : *app.filemap*. Voici le code source d'une entité récupérant la ressource *app.filemap* et permettant d'afficher son contenu.

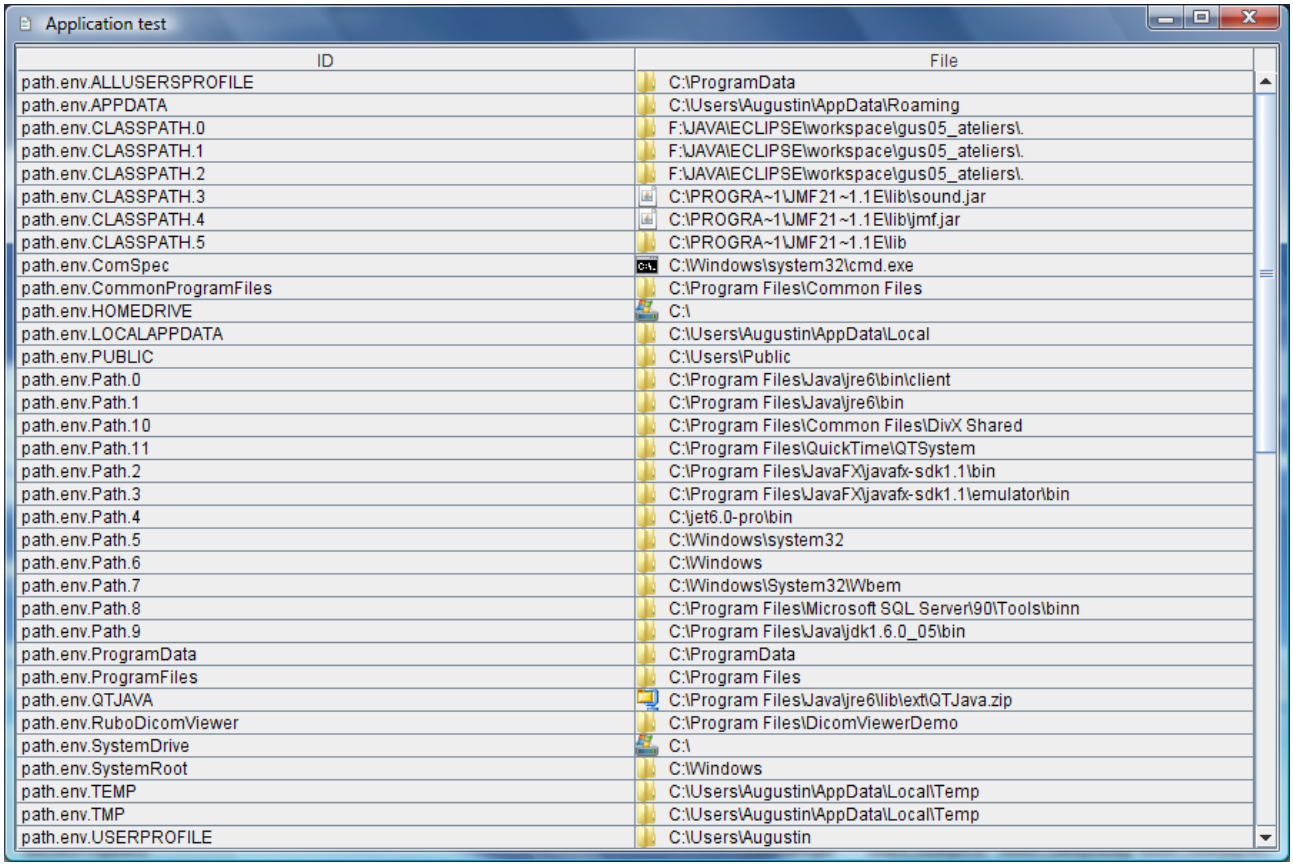
Vous pouvez créer et tester cette nouvelle entité.

```
package gus05.entity.charly.display.resource.pathids;

import java.awt.Color;
import java.awt.Component;
import java.awt.Font;
import java.io.File;
import java.util.Collections;
import java.util.Map;
import java.util.Vector;
import gus05.framework.core.Entity;
import gus05.framework.core.Outside;
import gus05.framework.features.Graphic;
import javax.swing.Icon;
import javax.swing.JComponent;
import javax.swing.JLabel;
import javax.swing.JTable;
import javax.swing.filechooser.FileSystemView;
import javax.swing.table.DefaultTableModel;
import javax.swing.table.TableCellRenderer;

public class DisplayPathIDs implements Entity, Graphic {

	public String getName(){return "charly.display.resource.pathids";}
	public String getCreationDate(){return "2009.09.01";}
```

```
	private JTable table;
	private Map map;

	public DisplayPathIDs() throws Exception
	{
		map = (Map) Outside.resource(this,"app.filemap");

		table = new JTable();
		table.setEnabled(false);

		Object[][] content = createTableContent();
		String[] columns = new String[]{"ID","File"};
		table.setModel(new DefaultTableModel(content,columns));
		table.setDefaultRenderer(Object.class,new CellRenderer0());
	}

	public JComponent gui()
	{return table;}

	private Object[][] createTableContent() throws Exception
	{
		Object[][] content = new Object[map.size()][2];
		Vector keys = new Vector(map.keySet());
		Collections.sort(keys);

		for(int i=0;i<map.size();i++)
		{
			Object key = keys.get(i);
			content[i][0] = key;
			content[i][1] = map.get(key);
		}
		return content;
	}

	private Icon findIcon(File f)
	{
		if(!f.exists()) return null;
		return FileSystemView.getFileSystemView().getSystemIcon(f);
	}

	private class CellRenderer0 extends JLabel implements TableCellRenderer
	{
		public CellRenderer0()
		{
			setOpaque(true);
			setFont(getFont().deriveFont(Font.PLAIN));
		}
		public Component getTableCellRendererComponent(JTable table, Object value,
boolean isSelected, boolean hasFocus, int row, int column)
		{
			setText("");
			setIcon(null);
			setForeground(Color.BLACK);

			if(value instanceof String)
				setText(" "+value);

			if(value instanceof File)
			{
				File f = (File)value;
				if(!f.exists()) setForeground(Color.LIGHT_GRAY);
				setText(" "+f.getAbsolutePath());
				setIcon(findIcon(f));
			}
```

```
                    return this;
                }
            }
}
```

Vous devriez obtenir ceci :



La colonne de gauche fournit le nom du pathID et la colonne de droit son emplacement. Si celui-ci est réalisé, alors l'entité affiche l'icône du fichier ou du répertoire tel que l'OS le représente, sinon il ne met pas d'icône et affiche la valeur en gris. Si vous parcourez l'ensemble du tableau, vous constaterez qu'il contient une seule valeur en gris : l'emplacement correspondant à *path.propfile*, et que par ailleurs il se partage entre pathIDs de type "path.env." et de type "path.sys."

Etape 2 : Les pathIDs définis par le système

L'entité précédente nous montre qu'il existe un bon nombre de pathIDs définit automatiquement par le gestionnaire Kassia. En fait, celui-ci exploite les informations trouvées dans les propriétés System de la JVM pour détecter des emplacements réalisés sur votre ordinateur.

Les pathIDs de type "path.sys." sont issus de la map *System.getProperties()* qui fournit des propriétés sur la machine virtuelle Java. Comme pathIDs intéressants, nous pouvons signaler :

- **path.sys.java.home** : le répertoire d'installation de Java
- **path.sys.user.dir** : le répertoire dans lequel est lancé l'application

- **path.sys.user.home** : le répertoire "Utilisateur" de votre ordinateur

Les pathIDs de type "path.env." sont issus de la map *System.getenv()* qui fournit des propriétés sur l'environnement système de votre ordinateur. Comme pathIDs intéressants, nous pouvons signaler :

- **path.env.programFiles** : le répertoire d'installation des programmes
- **path.env.SystemRoot** : le répertoire d'installation de l'OS
- **path.env.SystemDrive** : le driver d'installation de l'OS (C:\)

Parmi tous ces pathIDs, le plus utilisé est à mon sens *path.sys.user.dir* qui vous permet de retrouver le répertoire à partir duquel est lancé l'application, pathID que nous avons utilisé pour définir le pathID *path.rootdir*.

Etape 3 : Les pathIDs définis par le paramétrage

Les pathIDs définis par le paramètrage ou "pathIDs utilisateur" regroupent tous les pathIDs qui ne commencent ni par "path.sys." ni par "path.env.". Ces pathIDs sont tous initialisés et gérés de la même manière (que nous allons détailler plus bas) à deux exceptions près qui sont :

- *path.rootdir* : le chemin d'accès du répertoire root de l'application
- *path.propfile* : le chemin d'accès du fichier de propriétés extérieur (*app.properties* de l'atelier 7)

Normalement, à tout pathID utilisateur correspond une propriété du même nom. Lorsque le pathID est sollicité pour la première fois, la valeur de cette propriété est utilisée pour tenter une résolution qui produit en cas de succès un objet File, lequel est alors stocké dans la map *app.filemap* du gestionnaire. Concernant *path.rootdir* et *path.propfile*, l'initialisation suit un mécanisme différent.

path.rootdir : Dans le cas où une valeur d'argument du type "root=<path>" est trouvée, alors la valeur récupérée <path> est stockée comme propriété de l'application sous la forme "path.rootdir=<path>", écrasant le cas échéant une éventuelle propriété interne du même nom. Après le mécanisme général s'applique.

path.propfile : Ce pathID est toujours initialisé d'une manière ou d'une autre au démarrage de l'application, qu'il corresponde à un emplacement réalisé ou non. Dans le cas où aucune propriété *path.propfile* n'a été paramétrée en interne, le gestionnaire l'initialise comme ceci :

- Si le pathID *path.rootdir* est résolvable :
  path.propfile = <path.rootdir>\app.properties

- Sinon :
  path.propfile = <path.sys.user.dir>\app.properties

Etape 4 : Les mécanismes de résolution des pathIDs

Ce qui va suivre est un peu technique mais je préfère vous exposer une fois pour toute l'ensemble des règles de résolution quitte à ce que vous ne mémorisiez pas tout mais que vous puissiez y revenir au besoin. Tout d'abord, il est nécessaire de distinguer sollicitation et résolution de pathID.

Règles de sollicitation d'un pathID path.aaa :

Lorsqu'un pathID *path.aaa* est sollicité pour obtenir l'objet File correspondant, le gestionnaire vérifie si ce pathID est déjà enregistré dans la map *app.filemap*, auquel cas, il renvoie l'objet File correspondant. Sinon il essaie de le résoudre et si l'objet obtenu n'est pas l'objet null, alors la résolution a réussi et le fichier File nouvellement crée est aussitôt stocké dans la map avant d'être renvoyé (on dit que le pathID est résolvable). Dans le cas contraire, l'objet null est renvoyé (on dit que le path ID est non résolvable).

Règles de résolution d'un pathID path.aaa :

1. Si la propriété *path.aaa* n'existe pas, alors l'objet null est renvoyé.
2. Si la propriété existe, sa valeur est récupérée (on l'appelera V).
3. Si cette valeur V est du type "<path.bbb>\kkkk", le gestionnaire sollicite le pathID *path.bbb* :
    4. Si l'objet null est obtenu, l'objet null est renvoyé.
    5. Sinon, *path.aaa* est résolu et sa valeur se déduit de celle de *path.bbb* en y ajoutant kkkk
6. Si cette valeur V est du type "[N]path.ccc" (avec N un nombre entier positif ou négatif), le gestionnaire sollicite le path *path.ccc* :
    7. Si l'objet null est obtenu, l'objet null est renvoyé.
    8. Sinon, *path.aaa* est résolu et sa valeur se déduit de celle de *path.ccc* en remontant son arborescence de X niveaux (X = |N| si N<0 ou bien X = deep – N si N>=0, deep représentant le nombre total de niveaux de *path.ccc*).
9. Si cette valeur est du type "path.ddd", le gestionnaire sollicite le pathID path.ddd et renvoie la valeur retournée par path.ddd.
10. En dernier recours, le gestionnaire suppose que cette valeur V correspond litteralement à une valeur d'emplacement (réalisé ou non), *path.aaa* est alors résolu et sa valeur est V.

Remarque :

- **les règles de sollicitation s'appliquent en fait à n'importe quelle valeur de chaîne de caractères mais dans le cas d'une valeur ne débutant pas par "path.", le gestionnaire suppose que celle-ci correspond litteralement à une valeur d'emplacement plutôt que de lui appliquer le mécanisme de résolution qui ne concerne que les pathIDs. Donc l'étape 6 de la résolution reste valable si la valeur V est du type "[N]kkkk".**

Ouf !!!!! J'ai rarement écrit quelque chose d'aussi rhédibitoire que ça. Quelques exemples ne seront pas de trop pour vous aider à mieux saisir les mécanismes explicités ci-dessus.

path.rootdir=D:\\Mes dossiers\\Travail Java\\Mon root
la règle de résolution n° 9 s'applique
résolution-> File: "D:\Mes dossiers\Travail Java\Mon root"

path.logfile=<path.rootdir>\\log.txt
la règle de résolution n° 5 s'applique
résolution-> File: "D:\Mes dossiers\Travail Java\Mon root\log.txt"

path.javadir=[-1]<path.rootdir>
La règle de résolution n° 8 s'applique avec N = -1 (on remonte d'un niveau à partir de path.rootdir)
résolution-> File: "D:\Mes dossiers\Travail Java"

path.mesdossiers=[1]<path.rootdir>
La règle de résolution n° 8 s'applique avec N = 1 (on remonte de (3-1) niveaux à patir de path.rootdir)
résolution-> File: "D:\Mes dossiers"

path.racine=[0]<path.logfile>
La règle de résolution n° 8 s'applique avec N = 0 (on remonte de (4-0) niveaux à patir de path.logdir)
résolution-> File: "D:\"

path.ebookdir=<path.racine>\\Mes eBooks
la règle de résolution n° 5 s'applique
résolution-> File: "D:\Mes eBooks"

Etape 5 : La redirection de répertoire

Le gestionnaire Kassia met en oeuvre un dernier mécanisme qui vient modifier la résolution des pathIDs dans le cas ou celle-ci renvoie un emplacement correspondant à un répertoire. Dans le cas particulier ou le répertoire contient à sa racine un fichier texte nommé "guslink.txt", la valeur texte de ce fichier est lue puis passée dans le mécanisme de sollicitation pour obtenir un nouvel emplacement qui vient remplacer le précédent.

Ce dernier mécanisme a été rajouté pour garantir un moyen de contrôle sur les répertoires utilisés par les applications lorsque leurs accès sont paramétrés par des propriétés internes de l'application et qu'il est difficile de mettre en place un fichier de propriétés externe pour les alterer.

Pour conclure ...

Le paramètrage des pathIDs semble éprouvant mais vous vous rendrez compte avec l'habitude qu'il est en fait assez intuitif. Il est de toute manière absolument indispensable dans la plupart des applications Java pour leur permettre d'accéder à des fichiers et des répertoires et donc de sauvegarder et de charger des données sur votre disque dur.

ATELIER 9 : Les entités de transformation

Objectif : Après les entités graphiques, nous nous attaquons à un nouveau type d'entité : les entités de transformation. Nous prendrons comme exemple plusieurs transformations textuelles.

Etapes :

1. Définition d'une entité de transformation
2. Plusieurs exemples simples de transformations textuelles
3. Une entité graphique pour tester ces transformations
4. La notion de multiplicité d'une entité
5. Tester des transformations textuelles supplémentaires

Jusqu'à présent, nous n'avons crée et utilisé que des entités graphiques mais il existe d'autres types d'entité. Parmi les plus utiles, on trouve les entités de transformation qui permettent notamment d'effectuer un traitement sur une donnée. Dans cet atelier, nous allons étudier le cas simple des transformations textuelles avec lesquelles une chaîne de caractères passée en entrée renvoie une autre chaîne de caractères. L'introduction de ce nouveau type d'entité permettra aussi de préciser la notion primordiale de multiplicité d'une entité.

Etape 1 : Définition d'une entité de transformation

Une entité de transformation est une entité dont la classe implémente l'interface *Transform* du package *gus05.framework.feature*, qui contient la méthode suivante :

```
public Object transform(Object obj) throws Exception;
```

Rien de bien compliqué là dedans. Des exemples seront plus parlants que de longues explications.

Etape 2 : Plusieurs exemples simples de transformations textuelles

Nous allons illustrer ce nouveau type d'entité par 4 exemples de transformation textuelle :

- *charly.stringtransform.character.touppercase* — met toutes les lettres du texte en majuscule
- *charly.stringtransform.character.tolowercase* — met toutes les lettres du texte en minuscule
- *charly.stringtransform.character.invertchar* — inverse l'ordre des lettres du texte
- *charly.stringtransform.character.shufflechar* — mélange l'ordre des lettres du texte

Le code source de chacune de ces entités est donné ci-dessous. Vous pouvez les créer et les paramétrer (en mettant + dans le fichier *load*, à la fin des lignes à la place de *: j'expliquerai pourquoi à l'étape 4). Nous verrons ensuite comment tester ces entités.

charly.stringtransform.character.touppercase :

```
package gus05.entity.charly.stringtransform.character.touppercase;

import gus05.framework.core.Entity;
import gus05.framework.features.Transform;

public class StringTransform_toUppercase implements Entity, Transform {

	public String getName(){return "charly.stringtransform.character.touppercase";}
	public String getCreationDate(){return "2009.09.02";}

	public Object transform(Object obj) throws Exception
	{return obj.toString().toUpperCase();}
}
```

charly.stringtransform.character.tolowercase :

```
package gus05.entity.charly.stringtransform.character.tolowercase;

import gus05.framework.core.Entity;
import gus05.framework.features.Transform;

public class StringTransform_toLowercase implements Entity, Transform {

	public String getName(){return "charly.stringtransform.character.tolowercase";}
	public String getCreationDate(){return "2009.09.02";}

	public Object transform(Object obj) throws Exception
	{return obj.toString().toLowerCase();}
}
```

charly.stringtransform.character.invertchar :

```
package gus05.entity.charly.stringtransform.character.invertchar;

import gus05.framework.core.Entity;
import gus05.framework.features.Transform;

public class StringTransform_invertChar implements Entity, Transform {

	public String getName(){return "charly.stringtransform.character.invertchar";}
	public String getCreationDate(){return "2009.09.02";}

	public Object transform(Object obj) throws Exception
	{
		StringBuffer b = new StringBuffer(obj.toString());
		return b.reverse().toString();
	}
}
```

charly.stringtransform.character.shufflechar :

```
package gus05.entity.charly.stringtransform.character.shufflechar;

import gus05.framework.core.Entity;
import gus05.framework.features.Transform;

public class StringTransform_shuffleChar implements Entity, Transform {

	public String getName(){return "charly.stringtransform.character.shufflechar";}
	public String getCreationDate(){return "2009.09.02";}

	public Object transform(Object obj) throws Exception
	{
		StringBuffer in = new StringBuffer(obj.toString());
```

```
                StringBuffer b = new StringBuffer();
                while(in.length()>0)
                {
                int n = (int)(Math.random()*in.length());
                b.append(in.charAt(n));
                in.deleteCharAt(n);
                }
                return b.toString();
        }
}
```

Etape 3 : Une entité graphique pour tester ces transformations

Il nous faut maintenant mettre au point une entité graphique qui soit capable de tester ces transformations. Cette entité proposera une interface graphique avec deux zones texte, une pour saisir l'input et l'autre pour visualiser l'output, et enfin un bouton pour effectuer la transformation.

L'entité *charly.test.stringtransform.panel1* dont le code source est affiché ci-après est une entité graphique plus complexe que les précédentes, d'abord d'un point de vue programmation Java (pour ceux qui découvrent l'API Swing en même temps que le framework gus05) et d'autre part d'un point de vue utilisation du framework puisque la méthode *perform()* qui est appelée lorsque l'utilisateur appuie sur le bouton donne l'occasion d'utiliser la dernière des 3 méthodes de la classe *Outside*, la méthode :

```
public static void err(Entity entity, String id, Exception e)
```

En effet, lorsqu'il est nécessaire de gérer une exception dans le code source d'une entité (le fameux try... catch...), la technique habituelle consiste à avoir recours à la méthode *err* de la classe *Outside* en passant en paramètre : l'entité elle-même, un identifiant et l'exception dont il est question. L'identifiant n'est qu'une indication informelle sencée renseigner sur l'endroit où s'est produite l'exception à l'intérieur de l'entité, donc vous êtes libre de la fixer comme bon vous semble. Personnellement, j'ai pris l'habitude de prendre le nom de la méthode dans laquelle survient l'exception et je vous encourage à adopter la même convention.

Vous pouvez créer et paramétrer cette entité de la même manière que toute entité graphique, puis la tester avec l'une des 4 entités de transformation textuelle.

Pour ce faire, vous devrez d'abord fixer la valeur de propriété :
app.maingui=charly.test.stringtransform.panel1
et la valeur de mapping :
charly.test.stringtransform.panel1@transform.string=charly.stringtransform.character.shufflechar
(dans le cas ou vous souhaitez tester la transformation *charly.stringtransform.character.shufflechar*)

Vous pouvez bien sûr vous amuser à relancer l'application en changeant à chaque fois la valeur de mapping pour tester successivement chacune des transformations.

```
package gus05.entity.charly.test.stringtransform.panel1;

import java.awt.BorderLayout;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JComponent;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;

import gus05.framework.core.Entity;
import gus05.framework.core.Outside;
import gus05.framework.core.Service;
import gus05.framework.features.Graphic;

public class TestPanel1 implements Entity, Graphic {

	public String getName(){return "charly.test.stringtransform.panel1";}
	public String getCreationDate(){return "2009.09.02";}

	private Service transform;

	private JTextArea area_input;
	private JTextArea area_output;
	private JButton button;
	private JPanel panel;

	public TestPanel1() throws Exception
	{
		transform = Outside.service(this,"transform.string");

		area_input = new JTextArea();
		area_output = new JTextArea();
		area_output.setEditable(false);

		button = new JButton("Perform transformation");
		button.addActionListener(new ActionListener(){
			public void actionPerformed(ActionEvent e){perform();}
		});

		JPanel p_center = new JPanel(new GridLayout(1,2));
		p_center.add(new JScrollPane(area_input));
		p_center.add(new JScrollPane(area_output));

		panel = new JPanel(new BorderLayout());
		panel.add(p_center,BorderLayout.CENTER);
		panel.add(button,BorderLayout.SOUTH);
	}

	private void perform()
	{
		try
		{
			String input = area_input.getText();
			String output = (String) transform.transform(input);
			area_output.setText(output);
		}
		catch(Exception e)
		{Outside.err(this,"perform()",e);}
	}

	public JComponent gui()
	{return panel;}
}
```

Etape 4 : La notion de multiplicité d'une entité

Il est temps d'aborder la notion de multiplicité d'une entité. Lorsque vous paramétrez une entité dans le fichier *load* afin que celle-ci soit chargée par le gestionnaire, vous devez impérativement faire terminer la ligne soit par +, soit par *. Il s'agit en fait de l'indication de multiplicité, + signifiant que l'entité est de multiplicité "unique" et * qu'elle est de multiplicité "multiple".

Jusqu'à présent, vous avez toujours choisi * pour les entités graphiques et + pour les entités de transformation. Il ne s'agit pas d'une règle stricte mais seulement d'une généralité. Je vais vous donner la définition de la multiplicité d'une entité et nous discuterons ensuite de ses implications.

Chaque entité est dotée d'une multiplicité dont les deux valeurs possibles sont UNIQUE et MULTIPLE.

Une entité est dite de multiplicité UNIQUE si chaque utilisation de cette entité se traduit par l'envoi d'une instance identique de classe à chaque fois.

Une entité est dite de multiplicité MULTIPLE si chaque utilisation de cette l'entité se traduit par l'envoi d'une instance distincte de classe à chaque fois.

Dit autrement, les entités appelantes qui font appel à une même entité de multiplicité unique partagent toutes la même instance de cette entité, alors que celles qui font appel à une même entité de multiplicité multiple possèdent chacune une instance de cette entité qui leur est propre.

Même si, lorsqu'on voit le code source d'une entité on peut dans 95% des cas dire clairement si elle doit être de multiplicité unique ou multiple, il ne faut pas perdre de vue que la multiplicité n'est pas une caractéristique intrinsèque de l'entité mais seulement un paramètrage. Suivant le contexte dans lequel on utilise une entité, il pourra certaines fois être souhaitable de changer sa multiplicité. Par ailleurs, nous verrons dans des ateliers plus avancés que la multiplicité peut être en quelque sorte modifiée dynamiquement pendant le runtime, ou plus exactement substituée par un mécanisme plus précis qui la fixe au niveau même de l'appel.

Qu'est-ce qui permet de dire alors si une entité doit être paramétrée avec une multiplicité unique ou multiple ? En règle générale, si une entité possède une variable de classe transmise à l'extérieure de celle-ci qui ne doit pas être partagée par plusieurs entités appelantes, alors cette entité doit avoir une multiplicité multiple. Ceci vous semblera sans doute un peu confus pour le moment mais avec l'expérience, vous y verrez plus clair.

Pour le moment, vous pouvez retenir les règles empiriques suivantes :

les entités graphiques doivent être de multiplicité multiple (*)
les entités de transformation doivent être de multiplicité unique (+)

Etape 5 : Tester des transformations textuelles supplémentaires

Je vais maintenant vous fournir quelques entités de transformation textuelle que j'ai eu l'occasion de développer, que vous allez pouvoir tester avec l'entité *charly.test.stringtransform.panel1*.

Téléchargez et dézippez l'archive zip accessible en ligne ci-dessous, puis placez les fichiers JAR qu'elle contient dans le sous-répertoire "entity" du répertoire root de votre projet Eclipse :

entity2.zip

Vous avez ainsi ajouté à votre répertoire entity 6 nouveaux fichiers jar listés ci-dessous :

- *gus.japanese.hiragana.builder*
- *gus.japanese.hiragana.convertor*
- *gus.security.messagedigest.builder1*
- *gus.stringtransform.japanese.hiragana*
- *gus.stringtransform.word.frequency*
- *gus.math.expression.simplify*

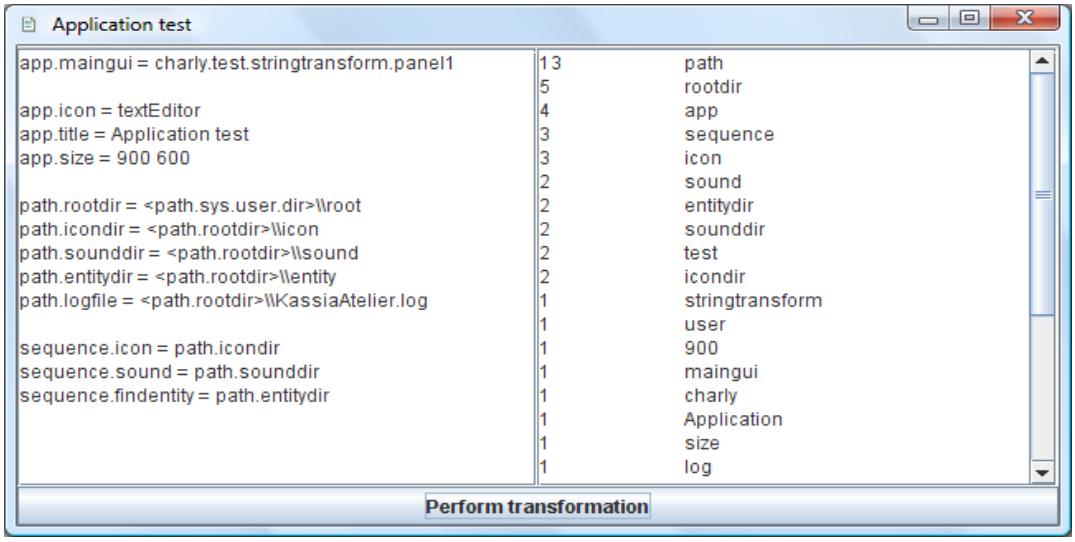
Nous allons présenter puis tester ses nouvelles entités en montrant à chaque fois un exemple d'utilisation.

L'entité *gus.stringtransform.word.frequency*

Il s'agit d'une entité qui calcule le nombre d'occurrences de chaque mot du texte passé en entrée et renvoie les valeurs sous forme de deux colonnes de texte, comme sur l'illustration ci-dessous.

Modifiez le mapping suivant :
charly.test.stringtransform.panel1@transform.string=gus.stringtransform.word.frequency
Puis lancez l'application.

Voici un exemple d'utilisation de cette entité :

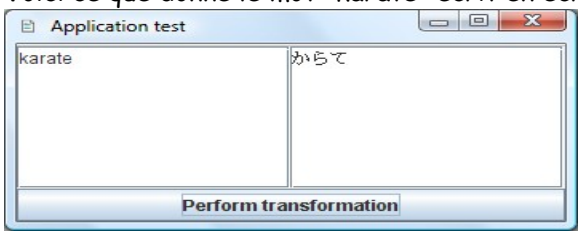


L'entité *gus.stringtransform.japanese.hiragana*

Il s'agit d'une entité qui transforme un texte japonais écrit en romaji (en caractères latins) en un texte écrit un hiragana (syllabaire japonais), et inversement ! Il faut noter que cette entité est dépendante des entités *gus.japanese.hiragana.builder* et *gus.japanese.hiragana.convertor*.

Modifiez le mapping suivant :
charly.test.stringtransform.panel1@transform.string=gus.stringtransform.japanese.hiragana
Puis lancez l'application.

Voici ce que donne le mot "karate" écrit en écriture japonaise hiragana :

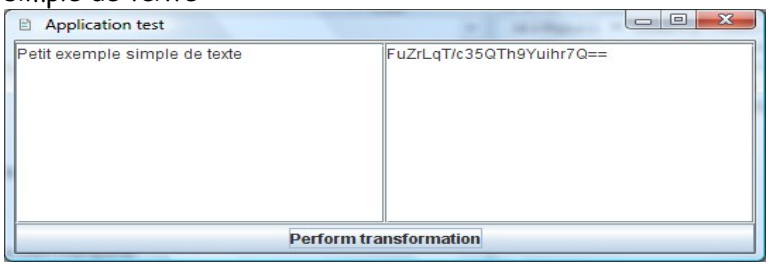


L'entité *gus.security.messagedigest.builder1*

Il s'agit d'une entité utile en cryptographie, qui génère le message digest MD5 d'une chaîne de caractères et affiche le nombre hexadecimal correspondant en encodage base64. Pour ceux qui connaissent le principe d'un checksum, le message digest suit le même principe, c'est à dire qu'il fournit une signature unique pour chaque séquence de donnée (fichier, texte, image...), ce qui permet d'en vérifier l'intégrité.

Modifiez le mapping suivant :
charly.test.stringtransform.panel1@transform.string=gus.security.messagedigest.builder1
Puis lancez l'application.

Voici donc le message digest (écrit en base64) correpondant à la chaîne de caractères "Petit exemple simple de texte" :

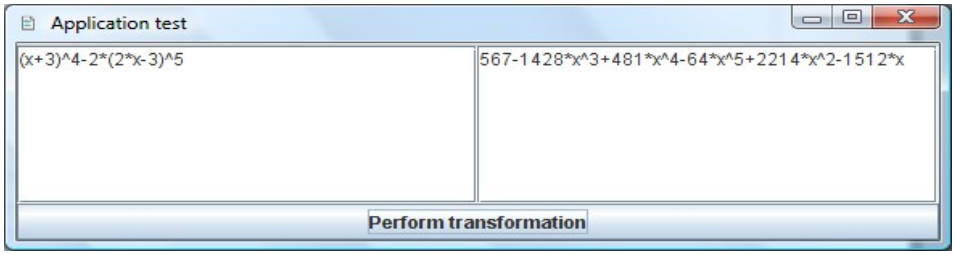


L'entité *gus.math.expression.simplify*

Voici pour finir un petit exemple de calcul formel (on est toutefois loin de concurrencer Maple) qui permet à partir d'une expression littérale d'obtenir la forme développée de cette expression.

Modifiez le mapping suivant :
charly.test.stringtransform.panel1@transform.string = gus.math.expression.simplify
Puis lancez l'application.

Voici donc la simplification de l'expression :
(x+3)^4-2*(2*x-3)^5 → 567-1428*x^3+481*x^4-64*x^5+2214*x^2-1512*x



ATELIER 10 : Faire évoluer ses entités

Objectif : Nous allons montrer comment on peut faire évoluer nos entités en reprenant le code source de l'entité développée à l'atelier 8. L'idée est de rendre personnalisable les icônes des fichiers

Etapes :

1. Des entités de transformation File -> Icon
2. Faire évoluer l'entité *charly.display.resource.pathids*
3. Tester l'entité dans différentes configurations
4. Réflexions sur les dépendances entre entités

Faire évoluer une entité consiste à modifier son code source afin d'en accroître les caractéristiques, d'étendre son mode de fonctionnement ou encore d'augmenter sa flexibilité d'utilisation tout en concervant (dans la mesure du possible) une compatibilité d'utilisation avec les versions précédentes. Nous allons dans ce cas reprendre l'entité *charly.display.resource.pathids* développée à l'atelier 8 pour en accroître la flexibilité graphique. Nous allons de fait permettre de personnaliser la manière dont les icônes des fichiers sont choisies.

Etape 1 : Des entités de transformation File -> Icon

Une entité de transformation File -> Icon est une entité dont la méthode *transform* prend en entrée un objet de type File et renvoie un objet de type Icon sensé représenter le fichier.

Un exemple très simple est donné ci-dessous :

```
package gus05.entity.charly.fileicon.icon1;

import java.io.File;

import javax.swing.Icon;
import javax.swing.filechooser.FileSystemView;

import gus05.framework.core.Entity;
import gus05.framework.features.Transform;

public class SystemIcon1 implements Entity, Transform {

	public String getName(){return "charly.fileicon.icon1";}
	public String getCreationDate(){return "2009.09.13";}

	public Object transform(Object obj) throws Exception
	{return findIcon((File)obj);}

	private Icon findIcon(File f)
	{
		if(!f.exists()) return null;
		return FileSystemView.getFileSystemView().getSystemIcon(f);
	}
}
```

Voici maintenant une deuxième entité qui reprend la première en la compliquant un peu.

```
package gus05.entity.charly.fileicon.icon2;

import java.io.File;
import java.util.Map;

import javax.swing.Icon;
import javax.swing.filechooser.FileSystemView;

import gus05.framework.core.Entity;
import gus05.framework.core.Outside;
import gus05.framework.features.Transform;

public class FileIcon2 implements Entity, Transform {

	public String getName(){return "charly.fileicon.icon2";}
	public String getCreationDate(){return "2009.09.14";}


	private Map map;


	public FileIcon2() throws Exception
	{map = (Map) Outside.resource(this,"app.gui.iconmap");}


	public Object transform(Object obj) throws Exception
	{return findIcon((File)obj);}


	private Icon findIcon(File f)
	{
		if(!f.exists()) return null;
		if(f.isFile())
		{
			String ext = getFileExtension(f);
			String iconID = "FILE_"+ext;

			if(map.containsKey(iconID))
			return (Icon) map.get(iconID);
		}
		return FileSystemView.getFileSystemView().getSystemIcon(f);
	}

	private String getFileExtension(File f)
	{
		String[] n = f.getName().split("\\.");
		if(n.length>1) return n[n.length-1].toLowerCase();
		return "";
	}
}
```

Cette deuxième entité nous permet de personnaliser les icônes associées à certains types de fichier en ajoutant dans notre répertoire d'icônes des fichiers gif nommés FILE_<file_extension>.gif. Si l'emplacement ne correspond pas à un fichier ou que son extension n'est pas prise en compte alors l'icône de l'OS est renvoyée. Il s'agit donc d'une amélioration de la première entité.

Vous pouvez créer et paramétrer ces deux entités en considérant qu'elles sont des entités de multiplicité unique. Nous allons ensuite voir comment nous pouvons les utiliser dans l'entité d'affichage des pathIDs développée à l'atelier 8.

Etape 2 : Faire évoluer l'entité charly.display.resource.pathids

Nous allons reprendre le code source de l'entité *charly.display.resource.pathids* et le modifier un peu. Les lignes ajoutés ou modifiées ont été surlignées en jaune ci-dessous.

```
package gus05.entity.charly.display.resource.pathids;

import java.awt.Color;
import java.awt.Component;
import java.awt.Font;
import java.io.File;
import java.util.Collections;
import java.util.Map;
import java.util.Vector;
import gus05.framework.core.Entity;
import gus05.framework.core.Outside;
import gus05.framework.core.Service;
import gus05.framework.features.Graphic;
import javax.swing.Icon;
import javax.swing.JComponent;
import javax.swing.JLabel;
import javax.swing.JTable;
import javax.swing.table.DefaultTableModel;
import javax.swing.table.TableCellRenderer;

public class DisplayPathIDs implements Entity, Graphic {

	public String getName(){return "charly.display.resource.pathids";}
	public String getCreationDate(){return "2009.09.01";}

	private JTable table;
	private Service fileIcon;
	private Map map;

	public DisplayPathIDs() throws Exception
	{
		fileIcon = Outside.service(this,"charly.fileicon.icon1");
		map = (Map) Outside.resource(this,"app.filemap");

		table = new JTable();
		table.setEnabled(false);

		Object[][] content = createTableContent();
		String[] columns = new String[]{"ID","File"};
		table.setModel(new DefaultTableModel(content,columns));
		table.setDefaultRenderer(Object.class,new CellRenderer0());
	}

	public JComponent gui()
	{return table;}

	private Object[][] createTableContent() throws Exception
	{
		int number = map.size();

		Object[][] content = new Object[number][2];
		Vector keys = new Vector(map.keySet());
		Collections.sort(keys);

		for(int i=0;i<number;i++)
		{
			Object key = keys.get(i);
			content[i][0] = key;
```

```
                content[i][1] = map.get(key);
            }
            return content;
        }

        private Icon findIcon(File f)
        {
            try{return (Icon) fileIcon.transform(f);}
            catch(Exception e){Outside.err(this,"findIcon(File)",e);}
            return null;
        }

        private class CellRenderer0 extends JLabel implements TableCellRenderer
        {
            public CellRenderer0()
            {
                setOpaque(true);
                setFont(getFont().deriveFont(Font.PLAIN));
            }
            public Component getTableCellRendererComponent(JTable table, Object value,
boolean isSelected, boolean hasFocus, int row, int column)
            {
                setText("");
                setIcon(null);
                setForeground(Color.BLACK);

                if(value instanceof String)
                    setText(" "+value);

                if(value instanceof File)
                {
                    File f = (File)value;
                    if(!f.exists()) setForeground(Color.LIGHT_GRAY);
                    setText(" "+f.getAbsolutePath());
                    setIcon(findIcon(f));
                }
                return this;
            }
        }
}
```

L'entité *charly.display.resource.pathids* est désormais dépendante d'une autre entité qui prend en charge la représentation des fichiers et répertoires sous forme d'icône. Une variable de classe de type Service est ajoutée et la méthode *findIcon* utilise cette variable pour récupérer les icônes des objets File d'une manière qui n'est plus déterminée par l'entité.

Etape 3 : Tester l'entité dans différentes configurations

Testez l'entité *charly.display.resource.pathids* pour vérifier qu'elle fournit bien la même interface graphique que précédemment. Cette entité fait appel par défaut à l'entité *charly.fileicon.icon1* sans qu'aucun mapping soit nécessaire.

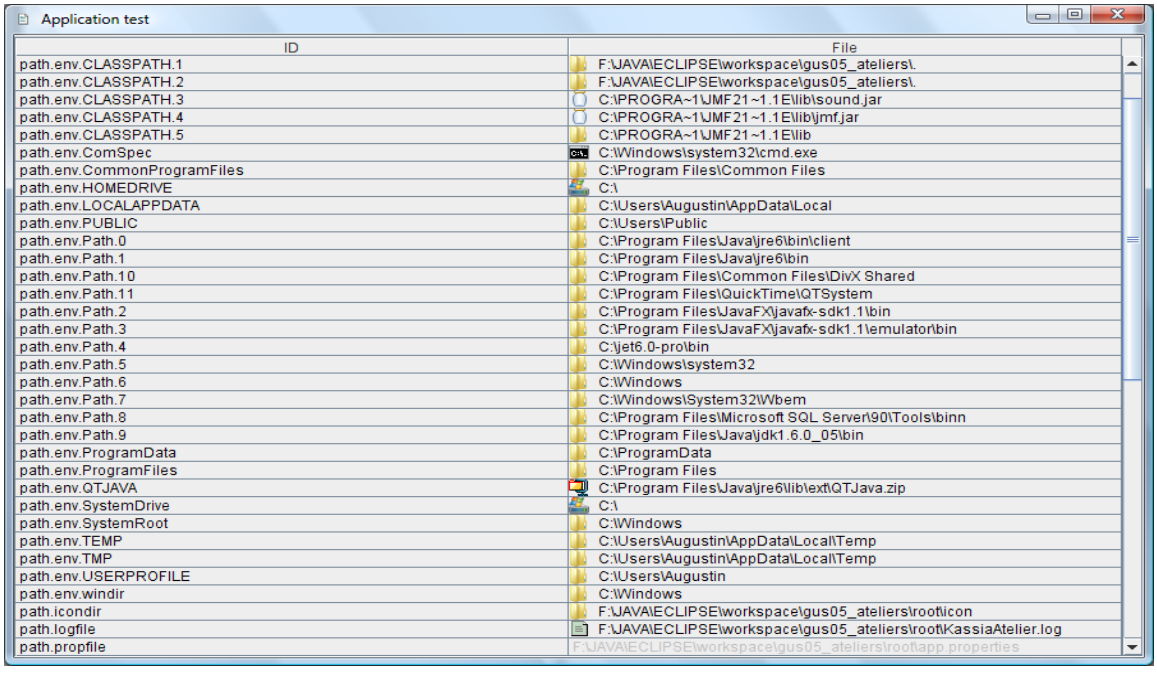
A présent ajoutez le mapping suivant et relancez l'application :
charly.display.resource.pathids@charly.fileicon.icon1 = charly.fileicon.icon2

On obtient toujours la même chose mais cette fois l'entité fait appel à charly.fileicon.icon2. Afin de profiter des fonctionnalités supplémentaires de cette deuxième entité de transformation File -> Icon, il nous faut récupérer et installer des nouvelles icônes.

Téléchargez le fichier zip ci-dessous et placez les 6 fichiers gif qu'il contient dans votre répertoire *icon*. Ces fichiers sont tous de la forme *FILE_<ext>.gif*

icones2.zip

Vous pouvez alors relancer l'application et vous devez observer quelques différences sur les icônes des fichiers de type jar, zip ou log.



La fonctionnalité qui consiste à personnaliser les icônes de fichiers en ajoutant des icônes *FILE_<ext>* apparaît ici pour ces quelques types mais sera surtout utile pour des fichiers dédiés à nos logiciels, dotés d'extensions que nous aurons choisi et qui sont a priori inconnues de l'OS.

Etape 4 : Réflexions sur les dépendances entre entités

Nous voyons apparaître clairement ici la notion de dépendance entre entités puisque dans ce cas (et dans les cas précédents d'entité graphiques) notre entité a recours aux services d'une autre entité par l'intermédiaire de l'appel à la méthode *service* de la classe *Outside*.

Dans le même temps, nous voyons que grâce au mécanisme du mapping, ces dépendances ne sont pas rigides. Par exemple, notre entité *charly.display.resource.pathids* a besoin d'une entité "dans le genre de" *charly.fileicon.icon1* mais pas nécessairement de cette entité précise. Elle a besoin d'une entité de transformation File -> Icon munie d'un mécanisme qui fournit pour chaque fichier l'icône correspondante sencée le représenter.

Pour gérer correctement ces dépendances et profiter pleinement de l'extrême flexibilité qu'elles offrent à nos développements, il est nécessaire de mettre en place une typologie des entités basées sur leurs structures et leurs modes de fonctionnement. Dans l'atelier suivant qui étudie les caractéristiques et introduit quelques types d'entité, nous parlerons du typage de l'entité. Mais il ne s'agira là que d'introduire le concept et non de l'étudier en profondeur, ce qui sera fait bien plus tard.

ATELIER 11 : Les caractéristiques et types de l'entité

Objectif : Il existe 10 caractéristiques possibles pour l'entité. Jusqu'à présent, nous n'avons vu que les entités graphiques et les entités de transformation. Il reste donc 8 caractéristiques à voir, en particulier la caractéristique Support.

Etapes :

1. Présentation rapide des 10 caractéristiques de l'entité
2. Etude de la caractéristique Support
3. Exemple d'entité multi-caractéristiques
4. Caractérisation et typage de l'entité
5. Signature d'une entité
6. Présentation de quelques types

Etape 1 : Présentation rapide des 10 caractéristiques de l'entité

Les 10 caractéristiques possibles pour l'entité correspondent aux 10 interfaces Java contenues dans le package *gus05.framework.features*. Comme lorsque nous avons introduit les caractéristiques *Graphic* et *Transform*, nous nous épargnerons de longues explications sur l'intérêt de chaque interface. Les exemples qui ne manqueront dans des prochains ateliers parleront d'eux même.

Caractéristique Execute

```
package gus05.framework.features;

public interface Execute {
    public void execute() throws Exception;
}
```

Caractéristique Filter

```
package gus05.framework.features;

public interface Filter {
    public boolean filter(Object obj) throws Exception;
}
```

Caractéristique Function

```
package gus05.framework.features;

public interface Function {
    public double function(double value) throws Exception;
}
```

Caractéristique Give

```
package gus05.framework.features;

public interface Give {
    public void give(Object obj) throws Exception;
}
```

Caractéristique Graphic

```
package gus05.framework.features;

import javax.swing.JComponent;

public interface Graphic {
    public JComponent gui();
}
```

Caractéristique Provide

```
package gus05.framework.features;

public interface Provide {
    public Object provide() throws Exception;
}
```

Caractéristique Register

```
package gus05.framework.features;

public interface Register {
    public void register(String key, Object obj) throws Exception;
}
```

Caractéristique Retrieve

```
package gus05.framework.features;

public interface Retrieve {
    public Object retrieve(String key) throws Exception;
}
```

Caractéristique Transform

```
package gus05.framework.features;

public interface Transform {
    public Object transform(Object obj) throws Exception;
}
```

Caractéristique Support

```
package gus05.framework.features;

import java.awt.event.ActionListener;
import java.util.List;

public interface Support {

	public void addActionListener(ActionListener listener);
	public void removeActionListener(ActionListener listener);
	public List listeners();
}
```

Parmi toutes ces caractéristiques, une seule s'avère plus complexe. Il s'agit de la caractéristique *Support* qui définit 3 méthodes alors que toutes les autres n'en proposent qu'une seule. Nous détaillerons cette caractéristique dans l'étape suivante.

Il faut faire remarquer qu'une classe d'entité pourra parfaitement implémenter librement une ou plusieurs (ou aucune) de ces interfaces sans perdre sa nature d'entité, la seule vraie contrainte étant qu'elle implémente l'interface *Entity*. Elle pourra bien entendu aussi implémenter n'importe quelle autre interface Java et hériter de n'importe quelle classe Java.

Les caractéristiques les plus utilisées sont sans conteste *Graphic*, *Transform*, *Give*, *Provide* et *Support*. Viennent ensuite *Execute*, *Retrieve*, *Register* et *Filter*. La caractéristique *Function* n'est quant à elle que très peu utilisée (seulement dans les cas d'entités qui se prennent pour des fonctions mathématiques…)

Etape 2 : Etude de la caractéristique Support

Pour bien comprendre l'intérêt de la caractéristique Support, il est préférable de connaître un peu le modèle de délégation d'évènements utilisé en Java dans les API graphiques AWT et SWING. Si vous n'êtes pas familier des "listeners" et des "adapters", vous pouvez par exemple lire le tutorial de Sun : Java AWT : Event Model Delegation

En guise d'introduction, je dirai que la caractéristique *Support* permet à l'entité de se comporter comme un bouton (classe *JButton*) sur lequel on clique et qui en réponse va envoyer un évènement vers un ensemble d'objets Java communément appelés listeners.

Tout comme la classe *JButton*, l'interface *Support* propose les méthodes *addActionListener* et *removeActionListener* qui permettent aux objets implémentant l'interface *ActionListener* (les fameux listeners) de s'enregistrer ou de se désenregistrer auprès de notre objet afin d'être avertis des évenements générés par celui-ci. La dernière méthode : *listeners*, permet quant à elle d'accéder à la liste des objets listener qui se sont enregistrés.

Pour tirer parti de la caractéristique *Support*, une entité doit a priori implémenter le mécanisme d'enregistrement des listeners qui se cache habituellement dernière les méthodes *addActionListener* et *removeActionListener*, notamment tenir à jour une liste de listeners et la parcourir à chaque fois qu'un évènement est généré pour le propager à chacun d'eux. Ce mécanisme étant répétitif et

fastidieux, le framework gus05 en propose un par défaut à travers la classe *DefaultSupport* située dans le package *gus05.framework.tools*. Pour tirer parti de la caractéristique *Support*, notre entité doit juste étendre la classe *DefaultSupport* et appeler la méthode *send* qu'elle comporte à chaque fois qu'elle souhaite générer un évènement :

```
public void send(Object source, String id);
```

Note : La méthode send est héritée de l'interface *Send* située dans le même package, ceci afin de la séparer clairement des méthodes de la caractéristique Support. Mais ceci n'a guère d'importance.

L'objet **source** correspond à la source de l'évènement, c'est à dire en pratique l'entité elle-même (on passera donc en paramètre *this*) et la chaîne de caractères **id** correspond à un identifiant de l'évènement permettant de le distinguer des autres évènements que l'entité peut éventuellement envoyer.

Nous allons dans cet atelier nous limiter à montrer à travers un exemple comment une entité peut générer des évènements grâce à la classe *DefaultSupport*. L'atelier suivant abordera la deuxième partie du mécanisme de gestion des évènements, c'est à dire la prise en compte des évenements par des implémentations de l'interface *ActionListener*.

Etape 3 : Exemple d'entité multi-caractéristiques

Je vous propose de développer une entité de multiplicité MULTIPLE munie des 4 caractéristiques : *Graphic*, *Give*, *Provide* et *Support*. Il s'agit d'une entité de type "éditeur de chaînes de caractères" qui agit comme un encapsuleur autour d'un composant graphique permettant à l'utilisateur d'éditer du texte. Voici ci-dessous le code source de l'entité *charly.dataeditor.string.editor1*.

Vous pouvez créer et paramétrer cette entité et nous commenterons ensuite divers aspects de son code.

```
package gus05.entity.charly.dataeditor.string.editor1;

import javax.swing.JComponent;
import javax.swing.JTextArea;
import javax.swing.event.DocumentEvent;
import javax.swing.event.DocumentListener;

import gus05.framework.core.Entity;
import gus05.framework.features.Give;
import gus05.framework.features.Graphic;
import gus05.framework.features.Provide;
import gus05.framework.tools.DefaultSupport;

public class StringEditor1 extends DefaultSupport implements Entity, Graphic, Give,
Provide {

	public String getName(){return "charly.dataeditor.string.editor1";}
	public String getCreationDate(){return "2009.09.23";}

	private JTextArea area;

	public StringEditor1()
	{
```

```
                area = new JTextArea();
                area.getDocument().addDocumentListener(new DocumentListener(){
                        public void changedUpdate(DocumentEvent e) {}
                        public void insertUpdate(DocumentEvent e) {dataModified();}
                        public void removeUpdate(DocumentEvent e) {dataModified();}
                });
        }

        private void dataModified()
        {send(this,"dataModified()");}

        public JComponent gui()
        {return area;}

        public void give(Object obj) throws Exception
        {area.setText((String)obj);}

        public Object provide() throws Exception
        {return area.getText();}
}
```

Une fois n'est pas coutûme, nous n'allons pas tester directement cette entité mais nous attendrons pour cela l'atelier suivant. Si vous décidiez de la mettre en entité graphique principale, vous n'auriez qu'une banale zone de texte sans grand intérêt. Cette entité est typiquement faite pour être utilisée par d'autres entités.

Attardons nous un peu sur son code source... Tout d'abord nous constatons que notre nouvelle entité implémente bien les interfaces *Graphic*, *Give* et *Provide* (et nous retrouvons d'ailleurs les méthodes associées à ces interfaces : *gui*, *give* et *provide*). Cette entité implémente aussi l'interface *Support* mais de manière indirecte, du fait de son héritage de la classe *DefaultSupport*. De fait, nous ne voyons pas dans le code source les 3 méthodes de l'interface *Support*, lesquelles sont déjà contenues dans *DefaultSupport*.

L'entité contient essentiellement une variable de classe de type *JTextArea* : *area*, qu'elle considère comme sa représentation graphique, et dont on peut accéder et modifier le contenu texte grâce aux caractéristiques *Give* et *Provide*. Par ailleurs, le bout de code suivant (qui vous déroutera peut être si c'est la première fois que vous le rencontrez) permet d'appeler la méthode *dataModified()* à chaque fois que le texte contenu dans *area* est modifié.

```
area.getDocument().addDocumentListener(new DocumentListener(){
                public void changedUpdate(DocumentEvent e) {}
                public void insertUpdate(DocumentEvent e) {dataModified();}
                public void removeUpdate(DocumentEvent e) {dataModified();}
        });
```

La méthode *dataModified()* se contente d'utiliser la méthode *send* de la classe *DefaultSupport* pour générer un évènement indiquant que le texte a été modifié.

```
private void dataModified()
{send(this,"dataModified()");}
```

Concernant le choix de l'identifiant passé en paramètre, vous remarquerez que j'adopte le même type de convention que pour l'identifiant passé en paramètre dans la méthode *err* de la classe *Outside*.

Sans entrer dans les détails, je conclurerai en disant que l'entité *charly.dataeditor.string.editor1* agit un peu comme un encapsuleur ou un "standardiseur" du composant graphique *JTextArea*. Tout ceci vous semblera plus clair lorsque nous montrerons dans l'atelier suivant comment utiliser cette entité.

Etape 4 : Caractérisation et typage de l'entité

La caractérisation d'une entité correspond à l'ensemble de ses caractéristiques, c'est à dire l'ensemble des interfaces qu'elle implémente parmi les 10 interfaces du package "feature". Le typage quant à lui regroupe les caractéristiques mais aussi des aspects d'utilisation de l'entité comme sa multiplicité ou encore la structure et le type des données attendues et renvoyées par ses méthodes. A un niveau plus élevé, on pourra aussi tenter de décrire son fonctionnement, la relation entre les données d'entrée et de sortie, ainsi que d'autres aspects qui rendent compte de ses mécanismes internes.

Dans ce sens, les dénominations "entité graphique" et "entité de transformation" renvoient avant tout à leurs caractérisations. Les types correspondants sont néanmoins definis comme suit :

Une entité de type graphique se définit comme une entité possédant une caractéristique *Graphic* et de multiplicité multiple.

Une entité de type transformation se définit comme une entité possédant une caractéristique *Transform* et de multiplicité unique.

La notion de typage (attribution d'un type aux entités) a déjà été introduite précédemment notamment lorsque nous avons parlé de transformation textuelle, de transformation "File -> Icon" ou d'éditeur de chaînes de caractères. A l'étape 4 de l'atelier précédent, nous avions conclu sur les dépendances d'entités en disant qu'il était nécessaire de mettre en place une typologie des entités basées sur leurs structures et leurs modes de fonctionnement. Qu'est-ce que cela implique exactement ?

En recherchant une définition de typologie sur Wikipedia, voici ce que nous trouvons :

Une typologie est une démarche, souvent scientifique mais fondée sur une étude, consistant à définir un certain nombre de types afin de faciliter l'analyse, la classification et l'étude de réalités complexes. Par extension, le terme typologie désigne parfois la liste des types propres à un domaine d'étude.

La typologie est une démarche progressive et complexe que nous menerons dans des ateliers plus avancés mais il s'agit pour l'heure de fournir quelques principes de base pour vous permettre d'appréhender la suite de notre étude. Nous nous appliquerons à définir petit à petit une hiérarchie de types du plus général au plus spécifique, basé sur des critères structurels et des critères fonctionnels. Bien évidemment les entités pourront cumuler plusieurs types, de même qu'elles peuvent cumuler plusieurs caractéristiques.

Etape 5 : Signature d'une entité

Le typage d'une entité se base essentiellement sur sa signature, qui est une sorte de description formalisée renseignant notamment sur les caractéristiques et la multiplicité de l'entité. En voici quelques exemples :

Signature pour le type graphique : Gra*
Signature pour le type transformation : Tra+
Signature pour le type transformation textuelle : Tra(String,String)+
Signature pour le type "icône de fichier" : Tra(File,Icon)+

Il faut noter que pour de nombreux types spécifiques, la signature est insuffisante pour les définir complétement et qu'une indication fonctionnelle est alors nécessaire. Par exemple, le type "icône de fichier" se définit précisément de la manière suivante :

Tra(File,Icon)+
Icon: l'icône de représentation du fichier File

Etape 6 : Présentation de quelques types

Voici pour finir quelques types d'entité que nous avons déjà utilisé ou que nous serons amené à utiliser prochainement.

Type graphique
Identifiant : graphic **Sous-type de :** **Signature :** Gra* **Indications :**

Type de transformation
Identifiant : transformation **Sous-type de :** **Signature :** Tra+ **Indications :**

Type "Transformation textuelle"
Identifiant : text transformation **Sous-type de :** transformation **Signature :** Tra(String,String)+ **Indications :**

Type "Icone de fichier"
Identifiant : file icon

Sous-type de : transformation
Signature : Tra(File,Icon)+
Indications : Icon = l'icône de représentation du fichier File

Type "conteneur de donnée"

Identifiant : data container
Sous-type de : graphic
Signature : Giv,Pro*
Indications : L'entité contient une donnée stockée.
Give et Provide permettent respectivement de changer et de récupérer la donnée.

Type "visualisateur de donnée"

Identifiant : data viewer
Sous-type de : data container
Signature : Giv,Pro,Gra*
Indications : entité graphique conteneur de donnée dont l'interface graphique permet à l'utilisateur de visualiser la donnée stockée.

Type "éditeur de donnée"

Identifiant : data editor
Sous-type de : data viewer
Signature : Giv,Pro,Gra,Sup*
Indications : entité visualisateur de donnée dont l'interface graphique permet à l'utilisateur d'éditer la donnée stockée.
Chaque action de modification de la donnée produit un évènement "dataModified()"

Type "éditeur de chaîne de caractères"

Identifiant : string editor
Sous-type de : data editor
Signature : Giv(String),Pro(String),Gra,Sup*
Indications : entité éditeur de donnée dont la donnée stockée est une chaîne de caractères

On peut évidemment imaginer plein d'autres types dans le même genre en variant le type de la donnée stockée (String, Date, Color, Map ...). Il ne s'agit là que de quelques exemples donnés à titre indicatif et qui n'ont qu'une valeur didactique. Leur pertinence sera même sans doute remise en cause lorsque nous aborderons véritablement la problèmatique de la typologie et les nombreux problèmes qu'elle soulève.

ATELIER 12 : Recevoir des évenements

Objectif : Nous allons voir comment une entité peut recevoir des évènements extérieurs. Comme exemple, nous allons créer une entité avec des éditeurs de texte synchronisés et une autre pour jouer un son au démarrage et à l'arrêt de l'application.

Etapes :

1. Synchronisation de deux zones texte
2. Synchronisation de quatre zones texte
3. Instancier des entités au démarrage de l'application
4. Une entité qui fait un "beep" au démarrage et à l'arrêt de l'application
5. Une entité qui joue un son au démarrage et à l'arrêt de l'application

Dans cet atelier, nous allons apprendre à nous servir de l'interface Java *javax.swing.ActionListener* pour recevoir des évènements. Cette interface définit une unique méthode prévue pour recevoir l'évènement sous la forme d'un objet de type ActionEvent :

```
public void actionPerformed(ActionEvent e)
```

Dans l'implémentation de cette méthode, il est possible à partir de l'objet *ActionEvent* de retrouver la source et l'identifiant qui sont à l'origine de l'évènement (passés en paramètre dans la méthode *send*) :

```
Object source = e.getSource();
String id = e.getActionCommand();
```

Cela sera utile pour distinguer les évènements dans les cas ou un seul objet *ActionListener* est utilisé pour écouter plusieurs objets *Support*, ou tout simplement lorsque l'objet *Support* est susceptible de générer plusieurs types d'évènement.

Etape 1 : Synchronisation de deux zones texte

L'idée est de prendre 2 éditeurs de texte, de les mettre cote à cote dans une même entité graphique et de faire en sorte que dés qu'on édite le texte de l'un, le texte de l'autre prenne automatiquement la même valeur.

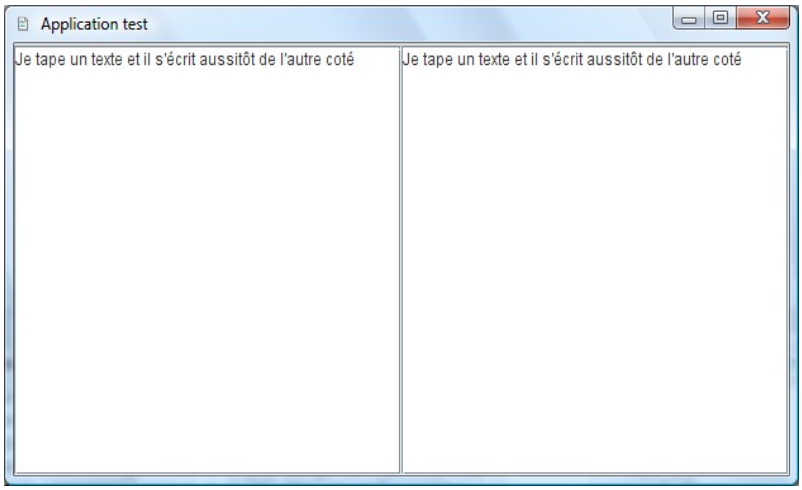
Voici dans un premier temps le code source d'une telle entité faisant appel à deux services *editor1* et *editor2* que l'on paramétra de la manière suivante :

```
charly.test.dataeditor.synchronize2texts@editor1=charly.dataeditor.string.editor1
charly.test.dataeditor.synchronize2texts@editor2=charly.dataeditor.string.editor1
```

Testez cette entité et observez bien ce qu'il se passe !

```
package gus05.entity.charly.test.dataeditor.synchronize2texts;

import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JComponent;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.Scrollable;

import gus05.framework.core.Entity;
import gus05.framework.core.Outside;
import gus05.framework.core.Service;
import gus05.framework.features.Graphic;

public class Synchronize2Texts implements Entity, Graphic {

	public String getName(){return "charly.test.dataeditor.synchronize2texts;}
	public String getCreationDate(){return "2009.09.23";}

	private Service editor1;
	private Service editor2;
	private JPanel panel;


	public Synchronize2Texts() throws Exception
	{
		editor1 = Outside.service(this,"editor1");
		editor2 = Outside.service(this,"editor2");

		editor1.addActionListener(new ActionListener() {
			public void actionPerformed(ActionEvent e)
			{transfert(editor1,editor2);}
		});
		editor2.addActionListener(new ActionListener() {
			public void actionPerformed(ActionEvent e)
			{transfert(editor2,editor1);}
		});

		panel = new JPanel(new GridLayout(1,2));
		panel.add(toComp(editor1));
		panel.add(toComp(editor2));
	}

	private JComponent toComp(Service s)
	{
		JComponent comp = s.gui();
		if(comp instanceof Scrollable)
			return new JScrollPane(comp);
		return comp;
	}

	private void transfert(Service in, Service out)
	{
		try
		{out.give(in.provide());}
		catch(Exception e)
		{Outside.err(this,"transfert(Service,Service)",e);}
	}

	public JComponent gui()
	{return panel;}
}
```

A première vue, l'entité fonctionne très bien :



à un détail près : les messages d'erreur qui se suivent dans le log au fur et à mesure que vous tapez votre texte. Quelque chose cloche, mais quoi donc ?

```
0052 PHASE 5: APPLICATION RUNTIME
0053 20090923_173248 5 1 [E] charly.test.dataeditor.synchronize2texts@transfert(Service,Service)
       java.lang.IllegalStateException
0054 > CAUSES:
0055 caused by: java.lang.IllegalStateException: Attempt to mutate in notification
(StringEditor1.java:39)
0056 20090923_173249 5 2 [E] charly.test.dataeditor.synchronize2texts@transfert(Service,Service)
       java.lang.IllegalStateException
0057 > CAUSES:
0058 caused by: java.lang.IllegalStateException: Attempt to mutate in notification
(StringEditor1.java:39)
0059 20090923_173249 5 3 [E] charly.test.dataeditor.synchronize2texts@transfert(Service,Service)
       java.lang.IllegalStateException
0060 > CAUSES: …
```

Ce qui cloche, c'est une erreur classique qui survient avec l'utilisation couplée de plusieurs objets Support. En effet, dans notre cas l'entité *charly.dataeditor.string.editor1* génère un évènement *dataModified()* aussi bien lorsque l'utilisateur édite le contenu du *JTextArea* que lorsque celui-ci est changé à travers la méthode *give*. On comprend vite ce qui se passe dans l'entité qui synchronise les deux textes : l'éditeur 1 étant modifié par l'utilisateur, celui-ci envoie un évènement qui copie le texte de cet éditeur vers l'éditeur 2. L'éditeur 2 réagit en envoyant un évènement qui copie le texte vers l'éditeur 1, et ainsi de suite. On est dans un cas de boucle infinie et on doit normalement s'attendre à voir notre application planter avec une interface graphique figée et une saturation de la mémoire.

Heureusement pour nous, la boucle est interrompue par une exception IllegalStateException sur le JTextArea puisqu'il est impossible de modifier son texte par la méthode setText lorsqu'il est en train d'être édité par l'utilisateur (problème d'accès concurrent par plusieurs Threads sur la même donnée)

Un moyen rapide de régler ce problème est d'interrompre volontairement la boucle en empêchant par exemple qu'un appel à la méthode transfert entraine un second appel à cette même méthode. Ceci peut être fait en ayant recours à un verrou bouléen qui indique si la méthode est actuellement utilisée ou non. Voici donc le nouveau le code de l'entité *charly.test.dataeditor.synchronize2texts* dans lequel les modifications on été surlignées en jaune.

```
package gus05.entity.charly.test.dataeditor.synchronize2texts;

import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JComponent;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.Scrollable;

import gus05.framework.core.Entity;
import gus05.framework.core.Outside;
import gus05.framework.core.Service;
import gus05.framework.features.Graphic;

public class Synchronize2Texts implements Entity, Graphic {

	public String getName(){return "charly.test.dataeditor.synchronize2texts";}
	public String getCreationDate(){return "2009.09.23";}

	private Service editor1;
	private Service editor2;

	private JPanel panel;
	private boolean transfering = false;

	public Synchronize2Texts() throws Exception
	{
		editor1 = Outside.service(this,"editor1");
		editor2 = Outside.service(this,"editor2");

		editor1.addActionListener(new ActionListener() {
			public void actionPerformed(ActionEvent e)
			{transfert(editor1,editor2);}
		});
		editor2.addActionListener(new ActionListener() {
			public void actionPerformed(ActionEvent e)
			{transfert(editor2,editor1);}
		});

		panel = new JPanel(new GridLayout(1,2));
		panel.add(toComp(editor1));
		panel.add(toComp(editor2));
	}


	private JComponent toComp(Service s)
	{
		JComponent comp = s.gui();
		if(comp instanceof Scrollable)
			return new JScrollPane(comp);
		return comp;
	}

	private void transfert(Service in, Service out)
	{
		try
		{
			if(transfering) return;
			transfering = true;
			out.give(in.provide());
			transfering = false;
		}
		catch(Exception e)
```

```
                {Outside.err(this,"transfert(Service,Service)",e);}
        }

        public JComponent gui()
        {return panel;}
}
```

Après avoir fait les corrections sur le code, vous pouvez relancer l'application et vérifier que la synchronisation des textes fonctionne sans message d'erreur dans le log.

Etape 2 : Synchronisation de quatre zones texte

L'entité suivante reprend le même principe mais avec cette fois 4 zones texte synchronisées. Nous retrouvons notamment le verrou booléen dans la méthode transfert pour empêcher que celle-ci ne soit appelée deux fois de suite.

Vous pouvez coder cette entité et la paramétrer en prenant le mapping suivant :

```
charly.test.dataeditor.synchronize4texts@editor1=charly.dataeditor.string.editor1
charly.test.dataeditor.synchronize4texts@editor2=charly.dataeditor.string.editor1
charly.test.dataeditor.synchronize4texts@editor3=charly.dataeditor.string.editor1
charly.test.dataeditor.synchronize4texts@editor4=charly.dataeditor.string.editor1
```

Voici le code source de l'entité :

```
package gus05.entity.charly.test.dataeditor.synchronize4texts;

import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JComponent;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.Scrollable;

import gus05.framework.core.Entity;
import gus05.framework.core.Outside;
import gus05.framework.core.Service;
import gus05.framework.features.Graphic;

public class Synchronize4Texts implements Entity, Graphic {

        public String getName(){return "charly.test.dataeditor.synchronize4texts";}
        public String getCreationDate(){return "2009.09.26";}

        private Service[] editors;

        private JPanel panel;
        private boolean transfering = false;

        public Synchronize4Texts() throws Exception
        {
                editors = new Service[4];

                editors[0] = Outside.service(this,"editor1");
                editors[1] = Outside.service(this,"editor2");
                editors[2] = Outside.service(this,"editor3");
```

```
		editors[3] = Outside.service(this,"editor4");

		panel = new JPanel(new GridLayout(2,2));
		panel.add(toComp(editors[0]));
		panel.add(toComp(editors[1]));
		panel.add(toComp(editors[2]));
		panel.add(toComp(editors[3]));

		editors[0].addActionListener(new ActionListener() {
			public void actionPerformed(ActionEvent e)
			{transfert(editors[0]);}
		});
		editors[1].addActionListener(new ActionListener() {
			public void actionPerformed(ActionEvent e)
			{transfert(editors[1]);}
		});
		editors[2].addActionListener(new ActionListener() {
			public void actionPerformed(ActionEvent e)
			{transfert(editors[2]);}
		});
		editors[3].addActionListener(new ActionListener() {
			public void actionPerformed(ActionEvent e)
			{transfert(editors[3]);}
		});
	}


	private JComponent toComp(Service s)
	{
		JComponent comp = s.gui();
		if(comp instanceof Scrollable)
			return new JScrollPane(comp);
		return comp;
	}

	private void transfert(Service in)
	{
		try
		{
			if(transfering) return;
			transfering = true;
			for(int i=0;i<4;i++)
			{
				if(editors[i]!=in)
				editors[i].give(in.provide());
			}
			transfering = false;
		}
		catch(Exception e)
		{Outside.err(this,"transfert(Service)",e);}
	}

	public JComponent gui()
	{return panel;}
}
```

Le résultat fournit sans surprise 4 zones texte synchronisées disposées en damier... A présent, nous allons tester avec cette entité les quelques éditeurs de texte suivants que j'ai crée pour l'occasion :

entity3.zip

Comme les fois précédentes, vous pouvez dézipper l'archive et placer les fichier JAR d'entité dans le répertoire entity de votre répertoire root, puis modifier le paramétrage de la manière suivante :

```
charly.test.dataeditor.synchronize4texts@editor1=gus.data.editor.string.editor_a1
charly.test.dataeditor.synchronize4texts@editor2=gus.data.editor.string.editor_a2
charly.test.dataeditor.synchronize4texts@editor3=gus.data.editor.string.editor_a3
charly.test.dataeditor.synchronize4texts@editor4=gus.data.editor.string.editor_a4
```

Vous pouvez alors relancer l'application et admirer le résultat !

En dehors de l'aspect esthétique et assez inutile de notre interface graphique, il s'agit avant tout de vous démontrer par des exemples simples qu'une entité qu'on a développé dans un certain contexte peut aisément être récombinée à d'autres entités pour fournir des résultats différents.

Etape 3 : Instancier des entités au démarrage de l'application

Après les synchronisations de texte, notre défi suivant dans cet atelier va être de mettre au point une entité capable de produire un son au démarrage et à l'arrêt de l'application. Notre entité ne sera pas graphique (en fait, elle ne disposera d'aucune caractéristique) et devra juste être instanciée au démarrage de l'application.

Après les fichiers de paramétrage *load*, *prop* et *mapping*, il existe un quatrième fichier : **start**, qui permet de spécifier une liste d'entités devant être initialisées au démarrage de l'application, avant même la création de l'interface graphique principale.

Créez dans le package *gus05.resource.gus.kassia* le fichier texte *start*. Il vous suffira par la suite d'ajouter dans ce fichier le listing des noms d'entité que vous souhaitez voir instanciées au démarrage de l'application.

Etape 4: Une entité qui fait un "beep" au démarrage et à l'arrêt de l'application

Une manière simple en Java de générer un "beep" est la méthode *beep()* de la classe *java.awt.Toolkit* :

```
Toolkit.getDefaultToolkit().beep();
```

Dans un premier temps, créez et paramétrez (multiplicité unique) l'entité très simple suivante, sans oublier d'ajouter son nom au fichier *start* :

```
package gus05.entity.charly.startexit.sound1;

import java.awt.Toolkit;
import gus05.framework.core.Entity;

public class StartExitSound1 implements Entity {

	public String getName(){return "charly.startexit.sound1";}
	public String getCreationDate(){return "2009.09.24";}

	public StartExitSound1()
	{Toolkit.getDefaultToolkit().beep();}
}
```

Si vous relancez l'application, vous devriez entendre un beep au démarrage de l'application.

En fait, le beep est généré au moment même ou l'entité *charly.startexit.sound1* est initialisée et il ne s'agit pas vraiment de l'instant de démarrage proprement dit. Cet instant correspond à la troisième phase visible dans le log, le moment ou l'application a terminé de se construire (c'est à dire que toutes les entités qui la composent ont été instanciées) et s'apprête à fournir un résumé de son état (quatrième phase). Plus précisément, ce moment dit de démarrage (ou start event) correspond à la ligne suivante dans le log :

```
0019 20090924_101213 3 0 [M] KS4_start@handleSupportStart(DefaultSupport) no task found
```

Les évènements indiquant le démarrage et l'arrêt de l'application sont accessibles grâce à deux objets *Support* du gestionnaire Kassia dont les identifiants sont les suivants :

```
Outside.service(this,"app.supportstart");
Outside.service(this,"app.supportexit");
```

Nous allons exploiter ces deux services en faisant surveiller leurs évènements par la classe de l'entité elle-même (qui devra donc implémenter l'interface *ActionListener*) afin de lui faire générer le beep en réponse aux évènements de démarrage et d'arrêt. Le code définitif de l'entité *charly.startexit.sound1* est donné ci-dessous :

```
package gus05.entity.charly.startexit.sound1;

import java.awt.Toolkit;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import gus05.framework.core.Entity;
import gus05.framework.core.Outside;
import gus05.framework.core.Service;
```

```
public class StartExitSound1 implements Entity, ActionListener {

	public String getName(){return "charly.startexit.sound1";}
	public String getCreationDate(){return "2009.09.24";}

	private Service startSupport;
	private Service exitSupport;

	public StartExitSound1() throws Exception
	{
		startSupport = Outside.service(this,"app.supportstart");
		exitSupport = Outside.service(this,"app.supportexit");

		startSupport.addActionListener(this);
		exitSupport.addActionListener(this);
	}

	public void actionPerformed(ActionEvent e)
	{Toolkit.getDefaultToolkit().beep();}
}
```

A titre indicatif, voici une variante plus condensée du code source :

```
package gus05.entity.charly.startexit.sound1;

import java.awt.Toolkit;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import gus05.framework.core.Entity;
import gus05.framework.core.Outside;

public class StartExitSound1 implements Entity, ActionListener {

	public String getName(){return "charly.sound.play";}
	public String getCreationDate(){return "2009.09.24";}

	public StartExitSound1() throws Exception
	{
		Outside.service(this,"app.supportstart").addActionListener(this);
		Outside.service(this,"app.supportexit").addActionListener(this);
	}

	public void actionPerformed(ActionEvent e)
	{Toolkit.getDefaultToolkit().beep();}
}
```

Apportez les modifications nécessaires à l'entité charly.sound.play. Vous pouvez relancer à présent l'application et constater que le beep se produit à son démarrage et à son arrêt.

Les messages de log respectifs qui attestent de cette surveillance des évèments de démarrage et d'arrêt sont les suivants :

```
0019 20090924_104528 3 0 [M] KS4_start@handleSupportStart(DefaultSupport) send:start (1 tasks found)

0057 20090924_104529 6 0 [M] T03@handleSupportExit(DefaultSupport)send:exit (1 tasks found)
```

Etape 5 : Une entité qui joue un son au démarrage et à l'arrêt de l'application

Souvenez vous de notre entité *charly.display.resource.sounds* qui affichait la liste des sons chargés au démarrage et était capable de les jouer. Nous allons créer une fonctionnalité similaire pour permettre à notre application de jouer un son spécifique à son démarrage et à son arrêt.

Voici une petite entité de multiplicité unique capable de jouer un son à partir de son identifiant. Vous pouvez la créer et la paramétrer :

```
package gus05.entity.charly.sound.play;

import java.applet.AudioClip;
import java.util.Map;

import gus05.framework.core.Entity;
import gus05.framework.core.Outside;
import gus05.framework.features.Give;

public class PlaySound implements Entity, Give {

	public String getName(){return "charly.sound.play";}
	public String getCreationDate(){return "2009.09.27";}

	private Map map;

	public PlaySound() throws Exception
	{map = (Map) Outside.resource(this,"app.soundmap");}

	public void give(Object obj) throws Exception
	{
		String soundID = (String) obj;
		if(!map.containsKey(soundID)) return;
		AudioClip clip = (AudioClip) map.get(soundID);
		clip.play();
	}
}
```

Il nous faut aussi choisir deux sons qui marqueront le début et la fin de notre application. Par convention, nous utiliserons les identifiants *start* et *exit* pour ces sons. Vous pouvez donc choisir deux fichier wav sur votre ordinateur (pas trop longs !) et les copier dans le répertoire *sound* de votre répertoire *root* en les renommant *start.wav* et *exit.wav*

Sinon, vous pouvez utiliser les deux fichiers wav contenus dans le zip ci-dessous :

Start_Exit.zip

Nous pouvons à présent nous attaquer à la dernière entité de l'atelier 12 :
charly.startexit.sound2

Le code source (que nous commenterons ensuite) est donnée ci-dessous. Vous pouvez la créer et la paramétrer (multiplicité unique) puis remplacer dans le listing start la précédente entité par celle-ci.

```
package gus05.entity.charly.startexit.sound2;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import gus05.framework.core.Entity;
import gus05.framework.core.Outside;
import gus05.framework.core.Service;

public class StartExitSound2 implements Entity, ActionListener {

	public String getName(){return "charly.startexit.sound2";}
	public String getCreationDate(){return "2009.09.27";}

	private Service playSound;
	private Service startSupport;
	private Service exitSupport;

	public StartExitSound2() throws Exception
	{
		playSound = Outside.service(this,"charly.sound.play");
		startSupport = Outside.service(this,"app.supportstart");
		exitSupport = Outside.service(this,"app.supportexit");

		startSupport.addActionListener(this);
		exitSupport.addActionListener(this);
	}

	public void actionPerformed(ActionEvent e)
	{
		String s = e.getActionCommand();
		if(s.equals("start")) play("start");
		else if(s.equals("exit")) play("exit");
	}

	private void play(String id)
	{
		try{playSound.give(id);}
		catch(Exception e)
		{Outside.err(this,"play(String)",e);}
	}
}
```

Dans ce dernier exemple, l'identifiant d'évènement est exploité pour distinguer parmi les deux évènements : start et exit. Dans chacun des cas, l'appel de la méthode *play* permet de jouer le son désiré, grâce au service *playSound*.

Vous pouvez relancer l'application pour constater que les sons de démarrage et d'arrêt sont joués à son démarrage et à son arrêt. N'hésitez pas par la suite à désactiver cette fonctionnalité en mettant la ligne du listing *start* en commentaire (faîtes la commencer par #) lorsque vous aspirerez à un peu de tranquilité pour tester nos futures entités !

ATELIER 13 : Afficher le statut de la connexion Internet

Objectif : Nous attaquons avec cet atelier notre première fonctionnalité complexe... Il s'agit de mettre en place un système pour réagir en temps réel aux connexions et déconnexions de votre ordinateur à Internet et changer en conséquence l'icône d'affichage du statut de connexion.

Etapes :

1. Une entité pour surveiller la connexion Internet
2. Une entité qui joue un son à l'interruption et au redémarrage de la connexion
3. Une entité label pour visualiser le statut de la connexion
4. Une nouvelle entité graphique principale

Etape 1 : Une entité pour surveiller la connexion Internet

Une technique parmi d'autres pour tester si votre ordinateur est bien connecté à Internet consiste à essayer d'ouvrir une connection sur une URL très connue (qui sera a priori toujours accessible en ligne), par exemple : http://www.google.com

Le code source permettant de faire un tel test est donné ci-dessous :

```
try
{
    URL urlTest = new URL("http://www.google.com");
    urlTest.openConnection().connect();
    System.out.println("Your computer is connected to Internet");
}
catch(IOException e)
{
    System.out.println("Your computer is not connected to Internet");
}
```

L'idée pour notre entité de surveillance de la connexion Internet va être d'utiliser un object Timer pour répéter le test à intervalle régulier (on choisira 1 seconde) et détecter les changements d'état de la connexion. Il s'agira d'une entité de multiplicité UNIQUE disposant d'une caractéristique *Filter* pour fournir l'état de la connexion et surtout d'une caractéristique *Support* pour informer en temps réel des changements d'état.

Caractéristique Filter :
L'object passé en paramètre est ignoré (on pourra passer null). La valeur de retour correspond à l'état de connexion : true = connecté, false = non connecté

Caractéristique Support :
L'entité génère 2 évenements qui sont :

- online() = passage de l'état déconnecté à connecté
- offline() = passage de l'état connecté à déconnecté

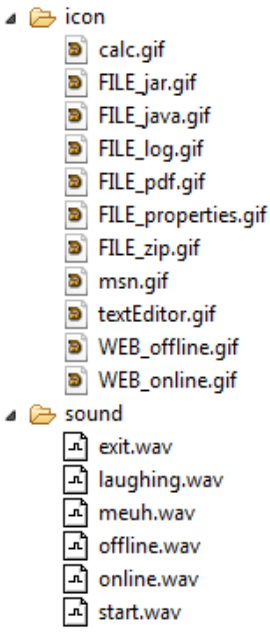
Voici le code source de l'entité :

```
package gus05.entity.charly.watch.internetconnection;

import java.io.IOException;
import java.net.URL;
import java.util.Date;
import java.util.Timer;
import java.util.TimerTask;

import gus05.framework.core.Entity;
import gus05.framework.features.Filter;
import gus05.framework.tools.DefaultSupport;

public class WatchInternetConnection extends DefaultSupport implements Entity, Filter {

    public String getName(){return "charly.watch.internetconnection";}
    public String getCreationDate(){return "2009.10.28";}

    public static final String URLTEST = "http://www.google.com";
    public static final long PERIOD = 1000;

    private Timer timer;
    private TimerTask task;
    private URL urlTest;
    private boolean isOnline = false;

    public WatchInternetConnection() throws Exception
    {
        urlTest = new URL(URLTEST);
        testConnection();

        task = new TimerTask(){public void run()
        {testConnection();}};

        timer = new Timer();
        timer.schedule(task,new Date(),PERIOD);
    }

    private void testConnection()
    {
        try
        {
            urlTest.openConnection().connect();
            if(isOnline) return;
            isOnline = true;
            online();
        }
        catch(IOException e)
        {
            if(!isOnline) return;
            isOnline = false;
            offline();
        }
    }

    public boolean filter(Object obj) throws Exception
    {return isOnline;}

    private void online()
    {send(this,"online()");}

    private void offline()
    {send(this,"offline()");}
}
```

Vous pouvez coder et paramétrer cette entité. Les étapes suivantes nous permettrons de tester qu'elle fonctionne correctement, mais tout d'abord, voici les ressources dont vous allez avoir besoin (2 icônes et 2 fichiers wav) pour les entités suivantes :

Online_Offline.zip

Téléchargez et dézippez cette archive puis placez respectivement les fichiers gif dans le répertoire "icon" et les fichiers wav dans le répertoire "sound". Rafraichissez votre projet sous Eclipse en appuyant sur F5 et voici ce que vous devriez avoir dans les deux répertoires :



Etape 2 : Une entité qui joue un son à l'interruption et au redémarrage de la connexion

Le moyen le plus simple d'exploiter notre entité de surveillance est d'émettre des alertes sonores lorsqu'une connexion ou une déconnexion à Internet est détectée. En reprenant le même principe que précédemment pour le démarrage et l'arrêt de l'application, on obtient l'entité suivante que vous pourrez coder, paramétrer (multiplicité UNIQUE) et ajouter au listing de start.

```
package gus05.entity.charly.watch.internetconnection.sound;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import gus05.framework.core.Entity;
import gus05.framework.core.Outside;
import gus05.framework.core.Service;

public class WatchInternetConnectionSound implements Entity, ActionListener {

	public String getName(){return "charly.watch.internetconnection.sound";}
	public String getCreationDate(){return "2009.12.01";}

	private Service play;
```

```
	private Service watch;

	public WatchInternetConnectionSound() throws Exception
	{
		play = Outside.service(this,"charly.sound.play");
		watch = Outside.service(this,"charly.watch.internetconnection");

		watch.addActionListener(this);
	}

	public void actionPerformed(ActionEvent e)
	{
		String s = e.getActionCommand();
		if(s.equals("online()")) play("online");
		else if(s.equals("offline()")) play("offline");
	}

	private void play(String id)
	{
		try{play.give(id);}
		catch(Exception e)
		{Outside.err(this,"play(String)",e);}
	}
}
```

Pour tester cette nouvelle fonctionnalité, vous pourrez lancer l'application puis essayer de débrancher et rebrancher le cable réseau de votre ordinateur (ou couper et remettre la connexion WiiFi ...). A chaque fois, vous devriez entendre les sons de verre brisé et de turbo-réacteur qu'on met en route, correspondant aux fichiers *offline.wav* et *online.wav* ...

Etape 3 : Une entité label pour visualiser le statut de la connexion

Un autre moyen moins intrusif d'informer l'utilisateur de l'état de la connexion Internet est d'ajouter à l'application un "petit voyant" allumé ou éteint, indiquant si la connexion est disponible ou non. Nous allons donc créer une entité graphique renvoyant un objet *JLabel* qui aura l'apparence suivante :

1.  offline (lorsque la connexion est indisponible)
2.  online (lorsque la connexion est disponible)

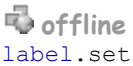
Les deux icônes correspondent aux fichiers *WEB_offline.gif* et *WEB_online.gif*. Pour les récupérer, nous pourrions utiliser le même objet map que dans l'entité *charly.display.resource.icons*, correspondant à l'id *app.gui.iconmap*, mais nous allons plutôt avoir recours à un objet de type *IconProvider*, définit dans le framework par l'interface *gus05.framework.resources.IconProvider*. Cet objet peut être obtenu grâce à l'id *app.iconprovider*, comme ceci :

```
iconProvider = (IconProvider) Outside.resource(this,"app.iconprovider");
```

Ensuite, l'objet iconProvider permet d'obtenir les icônes souhaitées simplement en appelant sa méthode *getIcon* et en précisant l'id de l'icône :

```
icon_online = iconProvider.getIcon("WEB_online");
icon_offline = iconProvider.getIcon("WEB_offline");
```

Nous allons donc créer une entité graphique renvoyant un objet *JLabel*, lequel sera mis à jour en temps réel pour correspondre à l'état de la connexion Internet, c'est à dire :




```
label.setIcon(icon_offline);
label.setText("offline");
label.setForeground(Color.GRAY);
```


```
label.setIcon(icon_online);
label.setText("online");
label.setForeground(Color.BLUE);
```

Le code source de l'entité est donné ci-dessous. Vous pouvez la coder et la paramétrer (multiplicité MULTIPLE), puis vous amuser à la tester provisoirement en la choisissant comme interface graphique principale de votre application (bien que, étant un JLabel, cette entité ne soit pas destinée au final à être une interface graphique principale…)

```
package gus05.entity.charly.watch.internetconnection.label;

import java.awt.Color;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.Icon;
import javax.swing.JComponent;
import javax.swing.JLabel;

import gus05.framework.core.Entity;
import gus05.framework.core.Outside;
import gus05.framework.core.Service;
import gus05.framework.features.Graphic;
import gus05.framework.resources.IconProvider;

public class WatchInternetConnectionLabel implements Entity, Graphic, ActionListener {

	public String getName(){return "charly.watch.internetconnection.label";}
	public String getCreationDate(){return "2009.12.01";}

	private Service watch;
	private IconProvider iconProvider;

	private JLabel label;
	private Icon icon_online;
	private Icon icon_offline;

	public WatchInternetConnectionLabel() throws Exception
	{
		watch = Outside.service(this,"charly.watch.internetconnection");
		iconProvider = (IconProvider) Outside.resource(this,"app.iconprovider");

		icon_online = iconProvider.getIcon("WEB_online");
		icon_offline = iconProvider.getIcon("WEB_offline");

		label = new JLabel(" ");
		updateLabel();
		watch.addActionListener(this);
	}

	public JComponent gui()
```

```
        {return label;}

        public void actionPerformed(ActionEvent e)
        {updateLabel();}


        private void updateLabel()
        {
          try
          {
              boolean isOnline = watch.filter(null);
              if(isOnline)
              {
                  label.setIcon(icon_online);
                  label.setText("online");
                  label.setForeground(Color.BLUE);
              }
              else
              {
                  label.setIcon(icon_offline);
                  label.setText("offline");
                  label.setForeground(Color.GRAY);
              }
          }
          catch(Exception e)
          {Outside.err(this,"updateLabel()",e);}
    }
}
```

Dans le cas de l'entité *charly.watch.internetconnection.sound*, on utilisait l'identifiant de l'évènement généré au changement d'état pour connaître le nouvel état de la connexion et émettre le son adéquate. Dans ce cas ci, nous procédons différemment : un appel à la méthode *filter* du service est effectué pour connaître l'état de la connexion (online ou offline) car nous souhaitons pouvoir mettre à jour le JLabel à la création de l'entité (c'est à dire, au démarrage de l'application) alors qu'aucun évènement de changement d'état n'est généré.

Etape 4 : Une nouvelle entité graphique principale

Nous souhaitons afficher notre JLabel d'état de connexion Internet dans une barre en bas de la fenêtre de notre application, dans le coin droit de la fenêtre. Pour ce faire, il est nécessaire de définir une entité graphique qui sera désormais prise comme interface graphique principale de l'application, laquelle se composera de notre barre du bas ainsi que d'un panneau central (correspondant à ce qui était précédemment notre interface graphique principale).

Le code source de cette entité est donné ci-dessous. Vous pouvez coder et paramétrer cette entité (multiplicité MULTIPLE), en prenant soin de ne pas oublier d'ajouter le mapping suivant :
charly.maingui.gui1@maingui = charly.test.dataeditor.synchronize4texts
et de changer la propriété *app.maingui* comme ceci :
app.maingui = charly.maingui.gui1

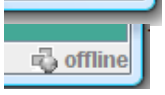
Dorénavant, lorsque vous souhaiterez mettre une nouvelle entité en interface graphique principale de votre application, vous pourrez changer la valeur de *charly.maingui.gui1@maingui* dans le fichier mapping plutôt que la valeur de la propriété *app.maingui* dans le fichier prop.

```
package gus05.entity.charly.maingui.gui1;

import gus05.framework.core.Entity;
import gus05.framework.core.Outside;
import gus05.framework.core.Service;
import gus05.framework.features.Graphic;

import java.awt.BorderLayout;
import javax.swing.JComponent;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.Scrollable;

public class MainGui1 implements Entity, Graphic {

	public String getName(){return "charly.maingui.gui1";}
	public String getCreationDate(){return "2009.12.03";}

	private Service webLabel;
	private Service maingui;

	private JPanel panel;


	public MainGui1() throws Exception
	{
		webLabel = Outside.service(this,"charly.watch.internetconnection.label");
		maingui = Outside.service(this,"maingui");

		JPanel p_bottom = new JPanel(new BorderLayout());
		p_bottom.add(webLabel.gui(),BorderLayout.EAST);

		panel = new JPanel(new BorderLayout());
		panel.add(buildComp(maingui),BorderLayout.CENTER);
		panel.add(p_bottom,BorderLayout.SOUTH);
	}

	public JComponent gui()
	{return panel;}


	private JComponent buildComp(Service s)
	{
		JComponent comp = s.gui();
		if(comp instanceof Scrollable)
			return new JScrollPane(comp);
		return comp;
	}
}
```

Vous pouvez relancer l'application et vous amuser à couper votre connexion Internet afin de vérifier que l'indicateur en bas à droite se met bien à jour comme prévu :

Lorsque la connexion Internet est disponible :

Lorsque la connexion Internet est indisponible :



ATELIER 14 : Les actions et les menus

Objectif : Cet atelier montre comment le gestionnaire Kassia permet de créer simplement des actions et de les ajouter à des menus. Ce sera aussi l'occasion d'approfondir le concept de multiplicité.

Etapes :

1. Une entité action pour terminer l'application
2. Une entité action pour afficher une boite de dialogue "A propos"
3. Une entité action pour charger un fichier texte vers un objet "String viewer"
4. Approfondissement du concept de multiplicité

Les actions (interface javax.swing.Action) sont des objets Java de l'API SWING qui héritent de l'interface ActionListener (précédemment introduite) et permettent d'associer des éléments graphiques (icône, label, description, shortcut...) à l'appel de la méthode actionPerform défini dans ActionListener. Ce sont des objets bien pratiques pour spécifier le comportement et l'apparence des composants graphiques "déclencheur d'action" tels que les boutons ou les menus.

Le framework gus05 offre un mécanisme de génération des objets Action à travers l'interface *ActionBuilder* définie dans le package *gus05.framework.resources*. Le gestionnaire Kassia fournit des objets de type ActionBuilder adaptés à chaque entité grâce à l'identifiant *actionbuilder*, comme ceci :

```
actionBuilder = (ActionBuilder) Outside.resource(this,"actionbuilder");
```

Ensuite, l'objet actionBuilder permet de générer les actions souhaitées en appelant sa méthode *buildAction* qui nécessite de préciser l'identifiant de l'action ainsi que l'objet *Execute* qui doit être éxecuté par l'action :

```
action = actionBuilder.buildAction("<actionID>",execute);
```

A quoi correspond donc l'identifiant passé en paramètre <actionID> ? Dans le cas le plus simple et sans autre paramétrage, il s'agit de l'identifiant de l'icône utilisée pour représenter l'action dans les boutons et les menus. Nous verrons juste après qu'il est possible de paramétrer un mapping qui fait correspondre des règles de représentation aux identifiants d'action, afin de personnaliser complément chaque action et même de la rendre multilingue (cf. atelier suivant).

Nous allons dans cet atelier développer un nouveau type d'entité appelé "Définisseur d'action" dont la signature est la suivante : Exe,Pro(Action)+

Ce type définit une d'entité de multiplicité UNIQUE contenant un objet Action associé à un objet Execute. L'objet Action est accessible par la caractéristique Provide et l'objet Execute transmet simplement sa caractéristique Execute à l'entité. Les exemples qui suivent vous permettront de vous fixer les idées.

Avant de réaliser les différentes étapes de cet atelier, vous devez d'abord récupérer et installer les ressources nécessaires aux entités, disponibles dans le fichier zip ci-dessous :

ResourcesAtelier14.zip

Etape 1 : Une entité action pour terminer l'application

Une action habituelle que l'on trouve dans les menus de logiciel est l'action de fermeture qui permet de quitter l'application. Cette première étape va nous donner l'occasion de développer une entité "Définisseur d'action" proposant une telle action.

Jusqu'à présent, le seul moyen dont vous disposiez pour quitter l'application était de cliquer sur la case de fermeture de la fenêtre principale en haut à droit. Mais l'application peut aussi être quittée programmatiquement grâce à un objet Execute contenu dans le gestionnaire Kassia, accessible comme ceci :

```
Outside.service(this,"app.executeexit");
```

Nous allons créer une entité *charly.appaction.exit* proposant une action dont l'identifiant *exit* correspond à l'ID de l'icône suivante :  , icône que vous avez trouvé dans l'archive zip ci-dessus.

Vous pouvez coder et paramétrer cette entité (multiplicité UNIQUE). Nous expliquerons ensuite comment l'utiliser.

```
package gus05.entity.charly.appaction.exit;

import javax.swing.Action;
import gus05.framework.core.Entity;
import gus05.framework.core.Outside;
import gus05.framework.core.Service;
import gus05.framework.features.Execute;
import gus05.framework.features.Provide;
import gus05.framework.resources.ActionBuilder;

public class ExitAction implements Entity, Provide, Execute {

	public String getName()		{return "charly.appaction.exit";}
	public String getCreationDate()	{return "2009.12.19";}

	public static final String ACTIONID = "exit";

	private Service executeExit;
	private ActionBuilder actionBuilder;
	private Action action;

	public ExitAction() throws Exception
	{
		executeExit = Outside.service(this,"app.executeexit");
		actionBuilder = (ActionBuilder) Outside.resource(this,"actionbuilder");
		action = actionBuilder.build(ACTIONID,executeExit);
	}

	public Object provide() throws Exception
	{return action;}

	public void execute() throws Exception
	{executeExit.execute();}
}
```

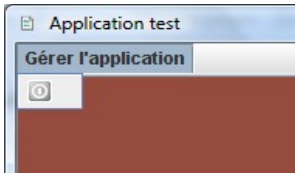
L'une des utilisations possibles de cette entité est l'ajout de l'objet Action dans un menu de l'application. Ceci peut être initialisé simplement par une entité de démarrage comme nous allons le voir. Afin d'ajouter un menu, nous devons disposer de l'objet JMenuBar correspondant à la barre de menu de notre application. Cet objet est accessible grâce à l'identifiant *app.gui.menubar* :

```
Outside.resource(this,"app.gui.menubar");
```

L'entité ci-dessous de multiplicité UNIQUE et prévue pour être lancée au démarrage récupère l'objet JMenuBar ainsi que le service associé à l'entité *charly.appaction.exit* pour initialiser le menu de notre application dont le titre est : "Gérer l'application"

```
package gus05.entity.charly.initmenu.menu1;

import javax.swing.Action;
import javax.swing.JMenu;
import javax.swing.JMenuBar;

import gus05.framework.core.Entity;
import gus05.framework.core.Outside;
import gus05.framework.core.Service;

public class InitMenu1 implements Entity {

	public String getName()          {return "charly.initmenu.menu1";}
	public String getCreationDate()  {return "2009.12.19";}

	private Service exitAction;
	private JMenuBar menuBar;

	public InitMenu1() throws Exception
	{
		exitAction = Outside.service(this,"charly.appaction.exit");
		menuBar = (JMenuBar) Outside.resource(this,"app.gui.menubar");

		Action action1 = (Action) exitAction.provide();
		JMenu menu1 = new JMenu("Gérer l'application");
		menu1.add(action1);

		menuBar.add(menu1);
	}
}
```

Vous pouvez coder et paramétrer cette entité puis relancer l'application. Vous devriez alors voir un menu "Gérer l'application" contenant un unique élément avec l'icône mais sans texte explicatif. Si vous cliquez sur cet élément, l'application se termine comme on pouvait s'y attendre.



Le résultat n'est pas parfait mais c'est déjà un début. Pour l'améliorer, il nous faut spécifier un mapping d'action permettant d'associer à chaque action une représentation graphique complète.

Pour cela, commencez par créer un fichier dans le package des ressources : *gus05.resource.gus.kassia*, que vous nommerez "actioninfo". Editez ensuite ce fichier de la manière suivante :

```
charly.appaction.exit@exit=exit#Quitter l'application
```

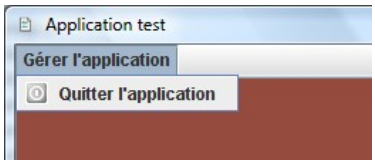
Vous aurez deviné qu'il s'agit d'un fichier properties, la clé de stockage correspondant à l'identifiant complet de l'action <entity-name>@<actionID> et la valeur correspondant à une représentation graphique d'action <iconID>#<display-text>.

Pour finir, il vous reste à indiquer que cette ressource doit être considérée comme le mapping pour les actions. Ajoutez la ligne suivante dans le fichier *prop* :

```
app.actioninfomap=data.map#inside.prop.actioninfo
```

A quoi peut bien correspondre cette valeur, vous demandez-vous certainement... Il s'agit d'un identifiant de Map dont la deuxième partie *inside.prop.actioninfo* précise la localisation de la source. Ce mécanisme du gestionnaire Kassia sera expliqué en détails dans des ateliers ultérieurs. Pour le moment, ne vous en préoccupez pas.

A présent, relancez l'application et vous pouvez vérifier que l'icône est restée la même mais que le texte affiché à coté correspond désormais au texte que nous avons paramétré dans le fichier *actioninfo*.



Etape 2 : Une entité action pour afficher une boite de dialogue "A propos"

Notre deuxième exemple d'action va être l'affichage de la boite de dialogue "A propos" que l'on retrouve dans tout logiciel qui se respecte. Nous allons procéder par étapes en codant, paramétrant et testant tout d'abord une entité "Définisseur d'action" qui permet d'afficher un message très simple comme "Ceci est une application de test !!" puis en affichant des informations spécifiques à l'application que nous irons chercher dans ses propriétés.

Une manière simple en Java d'afficher une boite de dialogue est de recourir à la classe JOptionPane. Le bout de code ci-dessous montre comment afficher un texte *message* dans une boite de dialogue standard dont le titre est *title*.

```
JOptionPane.showMessageDialog(null,message,title,JOptionPane.PLAIN_MESSAGE);
```

Nous allons créer une entité *charly.appaction.about* proposant une action dont l'identifiant *about* correspond à l'ID de l'icône suivante : . Vous pouvez coder et paramétrer cette entité (multiplicité UNIQUE).

```
package gus05.entity.charly.appaction.about;

import javax.swing.Action;
import javax.swing.JOptionPane;

import gus05.framework.core.Entity;
import gus05.framework.core.Outside;
import gus05.framework.features.Execute;
import gus05.framework.features.Provide;
import gus05.framework.resources.ActionBuilder;

public class AboutAction implements Entity, Provide, Execute {

	public String getName()          {return "charly.appaction.about";}
	public String getCreationDate()  {return "2009.12.23";}

	public static final String ACTIONID = "about";

	private ActionBuilder actionBuilder;
	private Action action;

	public AboutAction() throws Exception
	{
		actionBuilder = (ActionBuilder) Outside.resource(this,"actionbuilder");
		action = actionBuilder.build(ACTIONID,this);
	}

	public Object provide() throws Exception
	{return action;}

	public void execute() throws Exception
	{
		String message = "Ceci est une application de test !!";
		String title = "A propos";

	JOptionPane.showMessageDialog(null,message,title,JOptionPane.PLAIN_MESSAGE);
	}
}
```

Vous pouvez ensuite paramétrer l'action en ajoutant une deuxième ligne dans le fichier *actioninfo* :

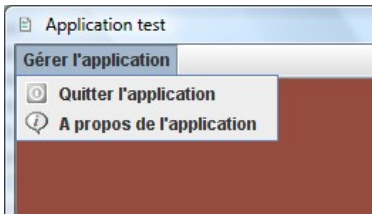
```
charly.appaction.about@about=about#A propos de l'application
```

Pour finir, il suffit d'ajouter cette action à notre menu "Gérer l'application" en modifiant le code source de l'entité *charly.initmenu.menu1* comme ci-dessous :

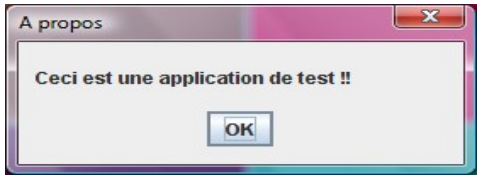
```
package gus05.entity.charly.initmenu.menu1;

import javax.swing.Action;
import javax.swing.JMenu;
import javax.swing.JMenuBar;

import gus05.framework.core.Entity;
import gus05.framework.core.Outside;
import gus05.framework.core.Service;

public class InitMenu1 implements Entity {

	public String getName()          {return "charly.initmenu.menu1";}
	public String getCreationDate()  {return "2009.12.19";}

	private Service exitAction;
	private Service aboutAction;
```

```
        private JMenuBar menuBar;

        public InitMenu1() throws Exception
        {
                exitAction = Outside.service(this,"charly.appaction.exit");
                aboutAction = Outside.service(this,"charly.appaction.about");
                menuBar = (JMenuBar) Outside.resource(this,"app.gui.menubar");

                Action action1 = (Action) exitAction.provide();
                Action action2 = (Action) aboutAction.provide();

                JMenu menu1 = new JMenu("Gérer l'application");
                menu1.add(action1);
                menu1.add(action2);

                menuBar.add(menu1);
        }
}
```

Nous pouvons à présent tester notre première mouture d'action "about" ...
Le menu s'est enrichi d'un nouvel élément :



Et lorsqu'on clique sur l'élément, notre boite de dialogue apparaît :



Bien, reprenons le code source de notre entité *charly.appaction.about* pour lui faire afficher dans la boite de dialogue les informations suivantes sur le logiciel :

- Le titre de l'application
- La version de l'application
- Le nom de l'auteur de l'application
- Le lien vers le forum gus05

Comment obtenir ces informations ? Grâce aux propriétés suivantes de l'application :

- app.title
- app.version
- app.author
- app.website

Définition des propriétés de base de l'application

Nous avions remarqué à l'atelier 4 que le gestionnaire Kassia initialise par défaut 4 propriétés : *app.name*, *app.title*, *app.version* et *app.build* dont jusque là seule *app.title* a été redéfinie dans le fichier *prop* parce-que cette propriété influence directement l'aspect graphique de l'application.

Ces 4 propriétés sont initialisées par défaut par le gestionnaire parce que ce sont les seules qu'il utilise de manière systèmatique. Elles permettent, notamment dans le log, d'identifier formellement quelle est l'application exécutée. Les valeurs par défaut indiquent qu'on est en présence de l'application test de Kassia, laquelle peut fonctionner sans aucun paramétrage. Pour n'importe quelle autre application, il est fortement conseillé de renseigner correctement ces 4 propriétés dans le fichier *prop* afin que celle-ci puisse être clairement identifiable.

Par ailleurs, plusieurs autres propriétes devront être ajoutées au paramétrage, en particulier celles qui sont exploitées par des entités de votre application. C'est le cas pour la seconde mouture de notre entité (dont le code source est ci-dessous) qui exploite les propriétés *app.author* et *app.website* encore non définies.

Nous allons donc commencer par personnaliser notre application en fixant les valeurs de propriétés suivantes dans le fichier de paramétrage *prop* :

```
app.name=testcharly
app.title=Application des ateliers de Charly
app.version=1.0
app.build=20091223
app.author=Charly
app.website=http://gus05.forumactif.com
```

Nous pouvons compléter le code source de l'entité *charly.appaction.about* pour obtenir sa version définitive ci-dessous :

```
package gus05.entity.charly.appaction.about;

import java.util.Map;
import javax.swing.Action;
import javax.swing.JOptionPane;

import gus05.framework.core.Entity;
import gus05.framework.core.Outside;
import gus05.framework.features.Execute;
import gus05.framework.features.Provide;
import gus05.framework.resources.ActionBuilder;

public class AboutAction implements Entity, Provide, Execute {

	public String getName()		{return "charly.appaction.about";}
	public String getCreationDate()	{return "2009.12.23";}

	public static final String ACTIONID = "about";

	public static final String PROPKEY_TITLE = "app.title";
	public static final String PROPKEY_VERSION = "app.version";
	public static final String PROPKEY_AUTHOR = "app.author";
```

```
	public static final String PROPKEY_WEBSITE = "app.website";


	private ActionBuilder actionBuilder;
	private Action action;
	private Map map;

	public AboutAction() throws Exception
	{
		actionBuilder = (ActionBuilder) Outside.resource(this,"actionbuilder");
		map = (Map) Outside.resource(this,"internal.propmap");
		action = actionBuilder.build(ACTIONID,this);
	}

	public Object provide() throws Exception
	{return action;}

	public void execute() throws Exception
	{
		JOptionPane.showMessageDialog(null,message(),"A
propos",JOptionPane.PLAIN_MESSAGE);
	}

	private String message()
	{
		String title = prop(PROPKEY_TITLE);
		String version = prop(PROPKEY_VERSION);
		String author = prop(PROPKEY_AUTHOR);
		String website = prop(PROPKEY_WEBSITE);

		return title+" version "+version+"\nRéalisé par "+author+"\n\nVisitez le
site web\n"+website;
	}

	private String prop(String key)
	{return map.containsKey(key)?(String) map.get(key):"?";}
}
```

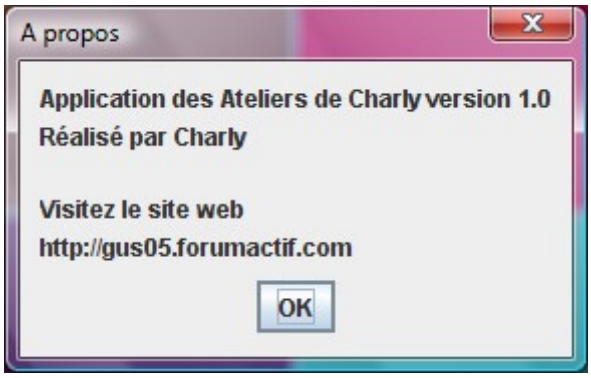
Je vais faire quelques remarques rapides sur le code ajouté :

1. Tout d'abord, l'entité fait appel à la ressource *internal.propmap* qui correspond à la map des propriétés de notre application (cf. entité de visualisation des propriétés de l'atelier 4).

2. Ensuite les noms des propriétés sont indiquées comme constante static final, ce qui accroit la lisibilité du code et rend leur valeur accessible par introspection de la classe.

3. Enfin, la méthode prop(String key) permettant d'accéder à une valeur de la propriété teste si la propriété existe avant de renvoyer sa valeur, faute de quoi elle renvoie "?". Ceci permet de garantir le fonctionnement de l'entité même dans le cas où les propriétés dont elle a besoin n'ont pas été définies.

Bien, il n'y a plus qu'à tester le résultat final. Le voici !



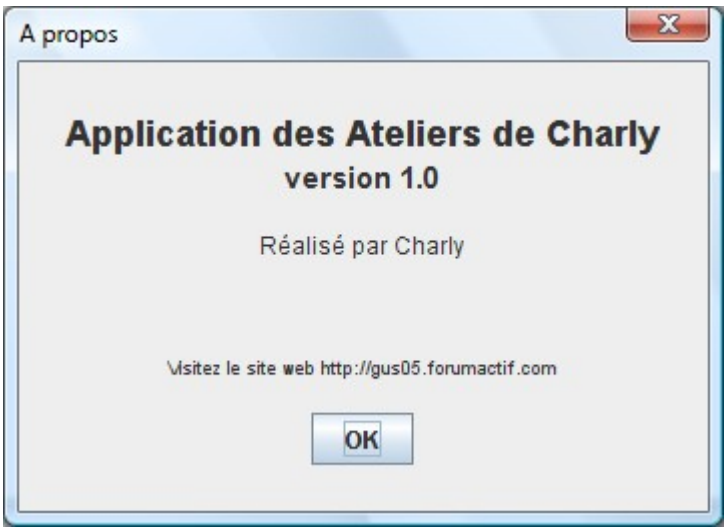
Toutes les informations sont là mais la mise en forme du message est sommaire. Pour obtenir un graphisme plus avenant, il faudrait passer du temps à créer un panneau en soignant la mise en forme, ce dont nous nous abstiendrons pour le moment, le but étant de montrer le principe d'affichage de la boite de dialogue "A propos"...

Voici une entité déployée que vous pouvez utiliser en remplacement de *charly.appaction.about* :

Téléchargez et Placez dans le répertoire entity :
gus.app.action.about2a.jar

Ajoutez dans le fichier mapping :
```
charly.initmenu.menu1@charly.appaction.about=gus.app.action.about2a
```
Ajoutez dans le fichier actioninfo :
```
gus.app.action.about2a@about=about#A propos de l'application
```

Relancez l'application et voici votre nouvelle boite de dialogue "A propos" !



Etape 3 : Une entité action pour charger un fichier texte vers un objet "String viewer"

Un objet de type "String viewer" est un objet doté d'une caractéristique Give prévue pour recevoir une chaîne de caractères et permettant de visualiser la valeur de cette chaîne grâce à une interface graphique. L'objectif de cette troisième étape va être de mettre en place une action pour récupérer le texte d'un fichier et l'envoyer vers le "String viewer" principal de l'application.

Nous allons créer une entité *charly.appaction.loadtextfile* proposant une action dont l'identifiant *open* correspond à l'ID de l'icône suivante :  , icône que vous avez trouvé dans l'archive zip ci-dessus.

Voici ci-dessous une première mouture de cette entité, dont l'action ne fait rien pour le moment (qui peut être considérée comme le canevas pour les entités de type "Définisseur d'actions"). Nous allons d'abord coder et paramétrer cette première mouture et ajouter notre nouvelle action au menu "Gérer l'application" comme nous l'avons fait pour les deux autres actions. Ensuite, nous nous concentrerons sur comment implémenter la nouvelle fonctionnalité.

```
package gus05.entity.charly.appaction.loadtextfile;

import javax.swing.Action;
import gus05.framework.core.Entity;
import gus05.framework.core.Outside;
import gus05.framework.features.Execute;
import gus05.framework.features.Provide;
import gus05.framework.resources.ActionBuilder;

public class LoadTextFileAction implements Entity, Provide, Execute {

	public String getName()		{return "charly.appaction.loadtextfile";}
	public String getCreationDate()	{return "2009.12.23";}

	public static final String ACTIONID = "open";

	private ActionBuilder actionBuilder;
	private Action action;

	public LoadTextFileAction() throws Exception
	{
		actionBuilder = (ActionBuilder) Outside.resource(this,"actionbuilder");
		action = actionBuilder.build(ACTIONID,this);
	}

	public Object provide() throws Exception
	{return action;}

	public void execute() throws Exception
	{

	}
}
```

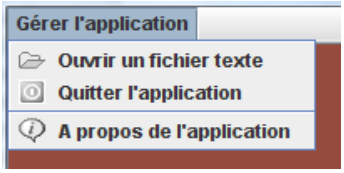
Vous pouvez préparer cette entité et ajouter son paramétrage comme suit :

Ligne à ajouter au fichier *actioninfo* :
```
charly.appaction.loadtextfile@open=open#Ouvrir un fichier texte
```
Ligne à ajouter au fichier *load* :
```
charly.appaction.loadtextfile=LoadTextFileAction+
```

Voici à présent les modifications à apporter à l'entité *charly.initmenu.menu1* :

```
package gus05.entity.charly.initmenu.menu1;

import javax.swing.Action;
import javax.swing.JMenu;
import javax.swing.JMenuBar;

import gus05.framework.core.Entity;
import gus05.framework.core.Outside;
import gus05.framework.core.Service;

public class InitMenu1 implements Entity {

	public String getName()		{return "charly.initmenu.menu1";}
	public String getCreationDate()	{return "2009.12.19";}

	private Service exitAction;
	private Service aboutAction;
	private Service loadAction;

	private JMenuBar menuBar;

	public InitMenu1() throws Exception
	{
		exitAction = Outside.service(this,"charly.appaction.exit");
		aboutAction = Outside.service(this,"charly.appaction.about");
		loadAction = Outside.service(this,"charly.appaction.loadtextfile");

		menuBar = (JMenuBar) Outside.resource(this,"app.gui.menubar");

		Action action1 = (Action) exitAction.provide();
		Action action2 = (Action) aboutAction.provide();
		Action action3 = (Action) loadAction.provide();

		JMenu menu1 = new JMenu("Gérer l'application");
		menu1.add(action3);
		menu1.add(action1);
		menu1.addSeparator();
		menu1.add(action2);

		menuBar.add(menu1);
	}
}
```

Si vous relancez l'application, voici le menu que vous devriez obtenir :



L'agencement du menu (que vous pouvez modifier à votre convenance) est dû au bout de code ci-dessous. Vous remarquerez que l'instruction addSeparator() permet d'ajouter des barres de séparation entre les actions du menu.

```
menu1.add(action3);
menu1.add(action1);
menu1.addSeparator();
menu1.add(action2);
```

Bien ! Revenons à l'entité *charly.appaction.loadtextfile* dont le but est de vous permettre de choisir un fichier sur votre ordinateur pour en récupérer le contenu texte et le transmettre au "String viewer principal" de votre application.

Voici ci-dessous le code source de cette entité que vous pouvez compléter.

```
package gus05.entity.charly.appaction.loadtextfile;

import java.io.File;
import java.io.FileReader;

import javax.swing.Action;
import javax.swing.JFileChooser;

import gus05.framework.core.Entity;
import gus05.framework.core.Outside;
import gus05.framework.core.Service;
import gus05.framework.features.Execute;
import gus05.framework.features.Provide;
import gus05.framework.resources.ActionBuilder;

public class LoadTextFileAction implements Entity, Provide, Execute {

	public String getName()		{return "charly.appaction.loadtextfile";}
	public String getCreationDate()	{return "2009.12.23";}

	public static final String ACTIONID = "open";


	private Service textViewer;
	private ActionBuilder actionBuilder;

	private Action action;
	private JFileChooser fc;

	public LoadTextFileAction() throws Exception
	{
		textViewer = Outside.service(this,"textviewer");
		actionBuilder = (ActionBuilder) Outside.resource(this,"actionbuilder");

		action = actionBuilder.build(ACTIONID,this);
		fc = new JFileChooser();
		fc.setFileSelectionMode(JFileChooser.FILES_ONLY);
	}

	public Object provide() throws Exception
	{return action;}

	public void execute() throws Exception
	{
		int v = fc.showOpenDialog(null);
		if(v!=JFileChooser.APPROVE_OPTION) return;

		File f = fc.getSelectedFile();
		String s = readText(f);
		textViewer.give(s);
	}
```

```
	private String readText(File f) throws Exception
	{
		FileReader fr = new FileReader(f);
		char[] a= new char[(int)f.length()];
		fr.read(a,0,(int)f.length());
		fr.close();
		return new String(a);
	}
}
```

Deux variables de classe ont été ajoutées : un service textViewer prévu pour recevoir le texte du fichier (le fameux "String viewer principal" de votre application, nous y reviendrons) et un objet JFileChooser qui va nous permettre de choisir un fichier.

Enfin, l'implémentation de la méthode execute (ce qu'il se passe lorsque notre action s'exécute) réalise les étapes suivantes :

1. Afficher un sélectionneur de fichier
2. Récupérer le fichier sélectionné (dans le cas ou le choix n'est pas validé par le bouton OK, on quitte la méthode)
3. Lire le texte contenu dans le fichier (grâce à la méthode *readText*)
4. Transmettre le texte à l'objet *textViewer*

Pour finir, il convient de se demander quel va être le String viewer principal de notre application, ceci afin de fixer correctement la valeur de mapping de l'appel *textviewer* de notre entité. Il s'agit bien sûr de l'entité *charly.test.dataeditor.synchronize4texts* qui a précédemment été paramétrée comme interface graphique centrale de notre application. Mais encore faut-il améliorer son code source pour la doter d'une caractéristique Give permettant de transmettre un texte simultanément aux 4 éditeurs texte qui la composent.

Les améliorations sont proposées ci-dessous, surlignées en jaune :

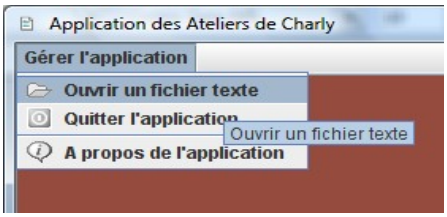
```
package gus05.entity.charly.test.dataeditor.synchronize4texts;

import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JComponent;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.Scrollable;
import gus05.framework.core.Entity;
import gus05.framework.core.Outside;
import gus05.framework.core.Service;
import gus05.framework.features.Give;
import gus05.framework.features.Graphic;

public class Synchronize4Texts implements Entity, Graphic, Give {

	public String getName(){return "charly.test.dataeditor.synchronize4texts";}
	public String getCreationDate(){return "2009.09.26";}

	private Service[] editors;
```

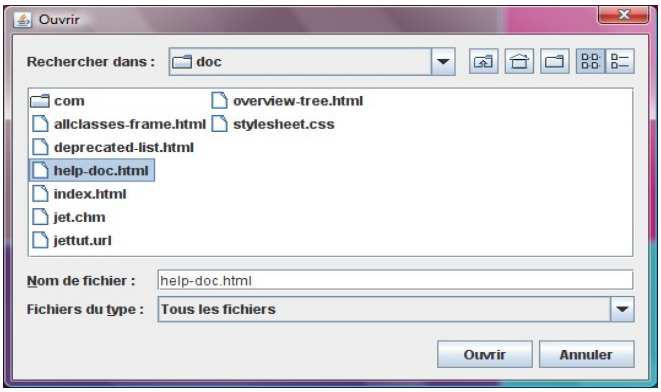
```
	private JPanel panel;
	private boolean transfering = false;

	public Synchronize4Texts() throws Exception
	{
		editors = new Service[4];

		editors[0] = Outside.service(this,"editor1");
		editors[1] = Outside.service(this,"editor2");
		editors[2] = Outside.service(this,"editor3");
		editors[3] = Outside.service(this,"editor4");

		panel = new JPanel(new GridLayout(2,2));
		panel.add(toComp(editors[0]));
		panel.add(toComp(editors[1]));
		panel.add(toComp(editors[2]));
		panel.add(toComp(editors[3]));

		editors[0].addActionListener(new ActionListener() {
			public void actionPerformed(ActionEvent e)
			{transfert(editors[0]);}
		});
		editors[1].addActionListener(new ActionListener() {
			public void actionPerformed(ActionEvent e)
			{transfert(editors[1]);}
		});
		editors[2].addActionListener(new ActionListener() {
			public void actionPerformed(ActionEvent e)
			{transfert(editors[2]);}
		});
		editors[3].addActionListener(new ActionListener() {
			public void actionPerformed(ActionEvent e)
			{transfert(editors[3]);}
		});
	}

	public void give(Object obj) throws Exception
	{
		transfering = true;
		for(int i=0;i<4;i++)
			editors[i].give(obj);
		transfering = false;
	}

	private JComponent toComp(Service s)
	{
		JComponent comp = s.gui();
		if(comp instanceof Scrollable)
			return new JScrollPane(comp);
		return comp;
	}

	private void transfert(Service in)
	{
		try
		{
			if(transfering) return;
			transfering = true;
			for(int i=0;i<4;i++)
			{
				if(editors[i]!=in)
				editors[i].give(in.provide());
			}
			transfering = false;
```

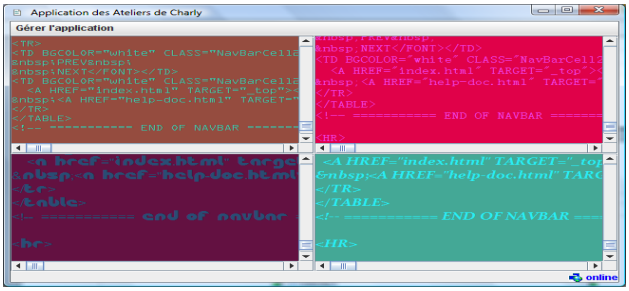
```
                }
                catch(Exception e)
                {Outside.err(this,"transfert(Service)",e);}
        }

        public JComponent gui()
        {return panel;}
}
```

Il nous reste à ajouter le mapping suivant :
`charly.appaction.loadtextfile@textviewer=charly.test.dataeditor.synchronize4texts`
et à tester notre nouvelle fonctionnalité ...

Lancez l'application, éxecutez l'action "Ouvrir un fichier texte" (1.), choisissez un fichier texte (2.), et visualisez son contenu texte dans les 4 éditeurs bariolés de notre application (3.). Ca fonctionne !

1. 

2. 

3. 

Etape 4 : Approfondissement du concept de multiplicité

Pour être vraiment honnête avec vous, je m'attendais à ce que notre précédent test ne fournisse pas le résultat escompté, c'est à dire que le texte du fichier ne s'affiche pas au final dans les 4 éditeurs. J'aurai alors entamé cette 4ème étape en vous expliquant que la raison de cet échec était un mauvais paramétrage de la multiplicité de nos diverses entités.

Mais voilà, tout s'est bien passé en apparence et j'ai fini par comprendre qu'on se trouvait dans une configuration encore plus tordue que ce que j'avais anticipé. Que dire sinon que par un heureux concours de circonstances la fonctionnalité fonctionne bien. Je débuterai cette 4ème étape en vous montrant la configuration que j'avais anticipé (qui foire...).

Souvenez vous de l'étape 2 de l'atelier 12 ou nous avions codé une entité qui permet la synchronisation de quatre zones texte : *charly.test.dataeditor.synchronize4texts*. Nous avions d'abord tester cette entité en prenant le mapping suivant :

```
charly.test.dataeditor.synchronize4texts@editor1=charly.dataeditor.string.editor1
charly.test.dataeditor.synchronize4texts@editor2=charly.dataeditor.string.editor1
charly.test.dataeditor.synchronize4texts@editor3=charly.dataeditor.string.editor1
charly.test.dataeditor.synchronize4texts@editor4=charly.dataeditor.string.editor1
```

avant d'adopter le mapping suivant :

```
charly.test.dataeditor.synchronize4texts@editor1=gus.data.editor.string.editor_a1
charly.test.dataeditor.synchronize4texts@editor2=gus.data.editor.string.editor_a2
charly.test.dataeditor.synchronize4texts@editor3=gus.data.editor.string.editor_a3
charly.test.dataeditor.synchronize4texts@editor4=gus.data.editor.string.editor_a4
```

dans lequel les entités exploitées sont des entités déployées sous forme de fichiers jar.

Je souhaite que vous reveniez au premier mapping pour lequel les 4 zones affichaient toutes des éditeurs "blancs" et que vous testiez de nouveau la fonctionnalité de chargement de texte à partir d'un fichier. Cette fois, ça ne fonctionne plus.

Pourquoi donc ?

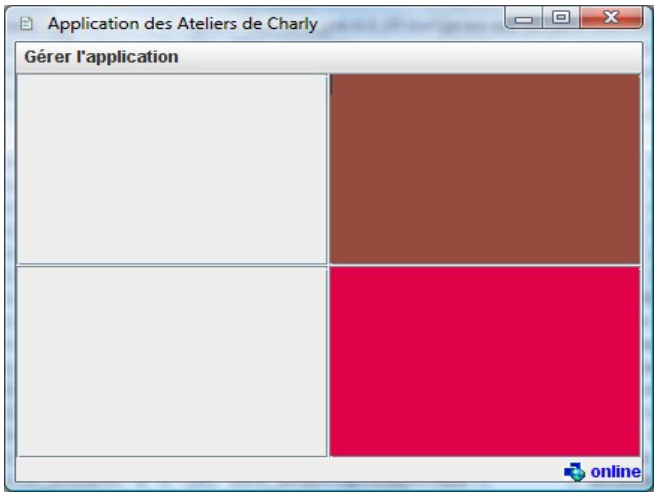
Cela tient en fait à une différence de multiplicité entre l'entité *charly.dataeditor.string.editor1* que vous avez développé et paramétré à l'étape 3 de l'atelier 11 (puis testé à l'atelier 12) et les entités déployées editor_a1, editor_a2, editor_a3, editor_a4. Si vous connaissez la multiplicité de votre éditeur (MULTIPLE), vous ne pouvez pas connaître la multiplicité des entités déployées puisque nous n'avons pas encore abordé ce sujet.

En fait, ces entités n'ont pas de multiplicité puisque nous n'en avons pour le moment paramétrer aucune. La multiplicité par défaut leur est donc appliquée : UNIQUE, ce qui n'est pas habituel pour des entités graphiques.

Quel est donc le problème soujacent avec des entités graphiques de multiplicité UNIQUE ? Si par exemple vous appliquez le mapping suivant, dans lequel les entités déployées sont utilisées à plusieurs endroits...

```
charly.test.dataeditor.synchronize4texts@editor1=gus.data.editor.string.editor_a1
charly.test.dataeditor.synchronize4texts@editor2=gus.data.editor.string.editor_a1
charly.test.dataeditor.synchronize4texts@editor3=gus.data.editor.string.editor_a2
charly.test.dataeditor.synchronize4texts@editor4=gus.data.editor.string.editor_a2
```

...voici ce qu'il advient de votre interface graphique :



Les deux cases de gauche sont vides !! Les entités editor_a1 et editor_a2 étant de multiplicité UNIQUE, elles ne sont instanciées qu'une seule fois chacune, ce qui signifie que leurs interfaces graphiques n'existent qu'en un exemplaire. L'entité appelante les positionne une première fois à un endroit puis à un endroit différent, ce qui annule le premier positionnement. Bref, on ne peut positionner simultanément à plusieurs endroits différents un même composant graphique. On retiendra que pour afficher plusieurs fois la même interface graphique, il est impératif que l'entité graphique soit de multiplicité MULTIPLE.

Cependant on a vu qu'en matière de multiplicité, il n'y a pas de règle stricte et il existe des situations où la multiplicité UNIQUE trouve son intérêt, notamment quand nous avons une entité graphique principale (affichée une seule fois) que nous souhaitons rendre accessible pour d'autres usages que l'affichage. C'est le cas de notre "String viewer principal" qui doit être accessible par l'entité *charly.appaction.loadtextfile* afin qu'il puisse lui transmettre du texte.

Un paramétrage normal voudrait que les entités editor_a1, editor_a2, editor_a3, editor_a4 soient de multiplicité MULTIPLE et l'entité *charly.test.dataeditor.synchronize4texts* de multiplicité UNIQUE. Et pour le moment, c'est exactement le contraire ! Actuellement, le gestionnaire crée deux instances distinctes de synchronize4texts, d'abord une pour le transfert de texte puis une autre pour l'affichage graphique. Mais ces deux instances utilisent les même instances d'éditeur et leur ordre de création fait que les composants graphiques des éditeurs sont correctement affichés par la deuxième instance et que ce sont ces mêmes éditeurs qui recevoient le texte à afficher.

Il nous reste à corriger les multiplicités des différentes entités pour parvenir à un paramétrage normal. Dans le fichier load, vous pouvez déjà effectuer la modification suivante :
charly.test.dataeditor.synchronize4texts=Synchronize4Texts+

Pour ce qui est des entités déployées, le paramétrage de la multiplicité peut se faire de la manière suivante :

Créez dans votre répertoire root un fichier *multiplicity.properties* et éditez le de la manière suivante :

```
gus.app.action.about2a=UNIQUE
gus.data.editor.string.editor_a1=MULTIPLE
gus.data.editor.string.editor_a2=MULTIPLE
gus.data.editor.string.editor_a3=MULTIPLE
gus.data.editor.string.editor_a4=MULTIPLE
gus.graphic.panel.screen.animation.test2=MULTIPLE
gus.japanese.hiragana.builder=UNIQUE
gus.japanese.hiragana.convertor=UNIQUE
gus.management.summary.icon4=MULTIPLE
gus.management.summary.systemenv2=MULTIPLE
gus.math.expression.simplify=UNIQUE
gus.security.messagedigest.builder1=UNIQUE
gus.stringtransform.japanese.hiragana=UNIQUE
gus.stringtransform.line.frequency=UNIQUE
gus.stringtransform.word.frequency=UNIQUE
gus.time.support.clock.gui2=MULTIPLE
gus.time.support.clock=UNIQUE
```

Ensuite, ajoutez au fichier *prop* les propriétés suivantes :

```
path.multiplicity=<path.rootdir>\\multiplicity.properties
sequence.entityinfo=path.multiplicity
```

Voilà, le paramétrage des multiplicités des entités déployées est terminé et lorsque vous ajouterez de nouvelles entités dans le répertoire *entity*, il vous suffira de rajouter les lignes correspondantes dans le fichier *multiplicity.properties*.

Vous devriez à présent un peu mieux saisir les implications de la multiplicité des entités dans la manière dont celles-ci interagissent les unes avec les autres. Mais si tout cela vous semble encore confus, ne vous inquiétez pas. Les ateliers suivants vous fourniront d'autres exemples pratiques qui permettront de vous habituer petit à petit à ces diverses notions de paramétrage.

ATELIER 15 : Le multilinguisme dynamique

Objectif : Le gestionnaire Kassia permet aussi de changer dynamiquement la langue de vos interfaces graphiques. Nous illustrerons ce mécanisme avec un exemple simple de traduction de menu en anglais, français et japonais.

Etapes :

1. Charger des ressources linguistiques
2. Visualiser les ressources linguistiques
3. Rendre les actions multilingues
4. Rendre le menu multilingue
5. Rendre le statut de connexion multilingue
6. Afficher et modifier la langue de l'application

Le multilinguisme dynamique appliqué à une fonctionnalité ou à une application toute entière signifie que cette fonctionnalité ou application est multilingue aussi bien dans son apparence graphique que dans les messages qu'elle transmet à l'utilisateur. L'aspect dynamique signifie quand à lui que l'utilisateur a la possibilité de changer la langue pendant l'execution même du programme sans avoir besoin de réinstaller ou redémarrer l'application.

Nous allons tout au long de cet atelier apprendre à charger et visualiser les resources linguistiques puis apprendre à les utiliser pour créer une interface graphique multilingue dont on pourra dynamiquement changer la langue.

Etape 1 : Charger des ressources linguistiques

Les données textuelles nécessaires à la traduction des éléments multilingues (interfaces graphiques, messages...) sont stockées sous la forme de fichiers texte munis de l'extension .ling (en fait des fichiers properties) appelés "ressources linguistiques". Comme toute autre ressource, celles-ci pourront soit être stockées en interne dans le jar, soit en externe sur votre ordinateur.

Vous pouvez déjà télécharger les quelques ressources linguistiques dont nous allons avoir besoin à cet atelier :

LingResources.zip

Cette archive zip contient les 3 fichiers suivants :

- CHARLY_fr.ling
- CHARLY_en.ling
- CHARLY_ja.ling

Les noms de fichier de ressource linguistique se conforment à la règle de nommage suivante :
<resource-id>_<language-code>.ling

<resource-id> correspond à l'identifiant de la ressource linguistique proprement dite.
<language-code> correspond au code ISO qui identifie la langue du fichier.

Nous disposons donc d'une resource linguistique *CHARLY* déclinée dans 3 langues différentes : le français, l'anglais et le japonais. Créez dans le répertoire *root* un sous répertoire *ling* dans lequel vous placerez ces fichiers.

Ensuite, ajoutez au fichier *prop* les deux propriétés suivantes :

```
path.lingdir=<path.rootdir>\\ling
sequence.ling=path.lingdir
```

Vous pouvez à présent relancer l'application et vérifier dans la log que votre ressource linguistique à bien été chargée au démarrage :

Etape 2 : Visualiser les ressources linguistiques

Etape 3 : Rendre les actions multilingues

```
charly.appaction.exit@exit = exit#EXIT
charly.appaction.about@about = about#ABOUT
gus.app.action.about2a@about = about#ABOUT
charly.appaction.loadtextfile@open = open#LOAD
```

Etape 4 : Rendre le menu multilingue

Etape 5 : Rendre le statut de connexion multilingue

Etape 6 : Afficher et modifier la langue de l'application

ATELIER 16 : Récapitulatif sur le framework et Kassia

Objectif : Voici un récapitulatif de ce que nous avons appris sur le framework gus05 lui-même et sur le gestionnaire Kassia. Cela sera l'occasion de réflechir sur leurs rôles respectifs et sur la portée des connaissances que nous avons acquises avant d'aborder ce qu'il reste à étudier.

Etapes :

1. Récapitulatif sur le framework gus05
2. Récapitulatif sur le gestionnaire Kassia
3. Ce que vous devez vraiment comprendre avant de continuer

Etape 1 : Récapitulatif sur le framework gus05

Le framework gus05 se compose de 20 classes que vous avez pu découvrir presque intégralement au cours des 15 premiers ateliers de ce tutorial (A une exception près). Voici ci-dessous un récapitulatif indiquant pour chaque classe l'atelier ou celle-ci a été introduite.

gus05.framework.core.Entity	vu à l'ATELIER 3
gus05.framework.core.Manager	(à voir)
gus05.framework.core.Outside	vu à l'ATELIER 4
gus05.framework.core.Service	vu à l'ATELIER 5
gus05.framework.features.Execute	vu à l'ATELIER 11
gus05.framework.features.Filter	vu à l'ATELIER 11
gus05.framework.features.Function	vu à l'ATELIER 11
gus05.framework.features.Give	vu à l'ATELIER 11
gus05.framework.features.Graphic	vu à l'ATELIER 3
gus05.framework.features.Provide	vu à l'ATELIER 11
gus05.framework.features.Register	vu à l'ATELIER 11
gus05.framework.features.Retrieve	vu à l'ATELIER 11
gus05.framework.features.Support	vu à l'ATELIER 11
gus05.framework.features.Transform	vu à l'ATELIER 9
gus05.framework.resources.ActionBuilder	vu à l'ATELIER 14
gus05.framework.resources.FileProvider	vu à l'ATELIER 14
gus05.framework.resources.IconProvider	vu à l'ATELIER 13
gus05.framework.resources.StringProvider	vu à l'ATELIER 15
gus05.framework.tools.DefaultSupport	vu à l'ATELIER 11
gus05.framework.tools.Send	vu à l'ATELIER 11

Seule l'interface *Manager* définie dans le package *gus05.framework.core* n'a pas été introduite pour la simple raison que c'est la seule interface du framework qui n'est a priori jamais utilisée directement par les entités. Nous allons néanmoins en toucher quelques mots juste après.

L'interface gus05.framework.core.Manager

L'interface *Manager* dont le code source est donné ci-dessous définit trois méthodes correspondant aux méthodes statiques de la classe *Outside* : service, resource et err. En fait, la classe *Outside* n'agit que comme un "conteneur statique" pour une implémentation de l'interface *Manager*, l'appel des méthodes service, resource et err renvoyant respectivement aux méthodes callService, callResource et sendError de l'objet contenu.

Voici le code source de l'interface Manager :

```
package gus05.framework.core;

public interface Manager {

    public Service callService(Entity entity, String id) throws Exception;
    public Object callResource(Entity entity, String id) throws Exception;
    public void sendError(Entity entity, String id, Exception e);
}
```

Voici à présent le code source de la classe Outside :

```
package gus05.framework.core;

public final class Outside {

    private static Manager manager;

    public static void setManager(Manager manager0)
    {if(manager==null)manager = manager0;}


    public static Service service(Entity entity, String id) throws Exception
    {return manager.callService(entity,id);}

    public static Object resource(Entity entity, String id) throws Exception
    {return manager.callResource(entity,id);}

    public static void err(Entity entity, String id, Exception e)
    {manager.sendError(entity,id,e);}
}
```

En tant que développeur d'entités, vous n'avez nul besoin de connaître l'envers du décors et vous pouvez donc parfaitement ignorer l'interface *Manager*. Cependant, si vous souhaitez vous intéresser un peu à la mécanique des gestionnaires (et éventuellement développer vos propres gestionnaires en temps voulu), vous pouvez déjà en apprendre un peu sur le rôle que doit tenir ce composant dans l'application : proposer une implémentation de l'interface *Manager* d'une part et transmettre cette implémentation à la classe *Outside* à travers la méthode *setManager* d'autre part.

Etape 2 : Récapitulatif sur le gestionnaire Kassia

Outre le framework gus05 proprement dit, ce premier tutorial nous a aussi permis d'aborder le gestionnaire Kassia, que ce soit au niveau du paramétrage des entités et des propriétés générales de l'application qu'au niveau des "objets fonctionnels" offerts aux entités. Nous allons ici récapituler globalement l'ensemble de ces éléments.

Paramétrage de base de l'application :

- Le fichier load — vu à l'ATELIER 3
- Le fichier prop — vu à l'ATELIER 3
- Le fichier mapping — vu à l'ATELIER 5
- Le fichier start — vu à l'ATELIER 12

Les propriétés prises en compte par le gestionnaire :

- app.icon — vu à l'ATELIER 4
- app.title — vu à l'ATELIER 4
- app.size — vu à l'ATELIER 4
- app.name — vu à l'ATELIER 4
- app.version — vu à l'ATELIER 4
- app.build — vu à l'ATELIER 4
- app.gus05id — vu à l'ATELIER 4
- app.maingui — vu à l'ATELIER 3
- app.actioninfomap — vu à l'ATELIER 14
- sequence.icon — vu à l'ATELIER 7
- sequence.sound — vu à l'ATELIER 7
- sequence.ling — vu à l'ATELIER 15
- sequence.mapping — vu à l'ATELIER 7
- sequence.findentity — vu à l'ATELIER 7
- sequence.entityinfo — vu à l'ATELIER 14
- sequence.start — vu à l'ATELIER 12
- path.rootdir — vu à l'ATELIER 7
- path.logfile — vu à l'ATELIER 7

Les objets fonctionnels du gestionnaire :

- internal.propmap
- app.gui.iconmap
- app.soundmap
- app.filemap
- app.supportstart

- app.supportexit
- app.iconprovider
- app.actionbuilder
- app.menubar

Etape 3 : Ce que vous devez vraiment comprendre avant de continuer

Avant de clore cette première partie de tutorial qui vous a permis à travers 16 ateliers d'apprendre progressivement les bases de la programmation gus05 et avant de poursuivre avec les 16 ateliers de la deuxième partie, il me semble important de faire le point sur tout ce que nous avons vu jusqu'à présent, ceci afin que vous ayez bien les idées claires sur les connaissances que vous êtes en train d'acquérir.

Car finalement, de quoi parle ce tutorial ? De la vingtaine de classe qui compose ce soit disant framework ? Du gestionnaire Kassia (dont le code est nettement plus complexe...) ? Des deux bien évidemment. Mais alors, pourquoi vous avoir fait croire que mon framework ne se résumait qu'à une vingtaine de classes quand il apparaît évident qu'elles ne font strictement rien et que le framework n'est "rien" sans le gestionnaire Kassia...

Mais je persiste et signe, le framework gus05 se résume bien à une vingtaine de classes extrêmement courtes dont la plupart sont des interfaces contenant une ou deux méthodes, le tout codé sans aucun commentaire et en java 1.4 qui plus est. Cette description n'est pas glorieuse j'en conviens et pourrait faire fuir n'importe quel développeur à la recherche de fonctionnalités intéressantes en Java, et pourtant de tout le code Java que j'ai écrit depuis maintenant 10 ans, si je ne devais garder qu'un seul truc, ce serait ce framework.

Vous devez bien saisir que le framework gus05 n'a qu'un rôle structurel au sein de l'application et peut être vu comme une sorte de base irréductible qui permet aux composants d'intéragir. Fonctionnellement parlant, ce sont les composants qui font tout le travail dans une application et le framework lui ne fait rien. Ceci m'amène à la conclusion suivante, ce n'est pas en vous présentant mon framework que j'aurai une chance de vous convaincre de son utilité mais en vous présentant les composants que j'ai crées à partir de celui-ci. Paradoxalement, les composants n'étant que des exemples d'utilisation du framework, leurs études respectives ne peuvent prétendre à un devenir un tutorial de connaissances générales sur le framework gus05 au caractère dogmatique et rassurant.

En particulier le gestionnaire Kassia n'est qu'un exemple d'utilisation du framework gus05 et j'aurai tout aussi bien pu rédiger ce tutorial en choisissant un autre gestionnaire avec des règles de fonctionnement différentes. Grâce au framework, les entités que nous avons développé ensemble auraient été parfaitement compatibles avec cet autre gestionnaire même si leur paramétrage se serait sans doute révélé différent. Tout cela pour dire que vous devez pouvoir distinguer dans tout ce que vous avez appris les connaissances liées au gestionnaire Kassia qui ne peuvent être prises "pour argent comptant" des connaissances liées au framework lui-même et au développement des entités qui sont quant à elles générales.

Mais alors, lorsqu'on développe une entité particulièrement complexe, comment s'assurer qu'elle sera compatible avec tous les gestionnaires ? Y a-t-il des précautions à prendre ? Aucune non... Ce sont aux gestionnaires à s'adapter aux entités et non l'inverse. Un bon gestionnaire doit être en mesure de proposer des mécanismes de paramétrage suffisamment flexibles pour s'adapter à n'importe quelle entité, et je veux croire que c'est le cas de mes dernières tentatives de gestionnaire : Endorf et Kassia. Contrairement au framework qui est définitivement figé depuis maintenant plus de 3 ans, le code source de mes gestionnaires évolue régulièrement pour y faire disparaître d'éventuels bugs et en améliorer les mécanismes. J'encourage d'ailleurs les développeurs les plus expérimentés à s'intéresser à ces problématiques et proposer des améliorations sur les gestionnaires existants ou même à en créer de nouveaux.

Ces 16 premiers ateliers nous ont permis de faire le tour du code source du framework gus05, de vous initier à la programmation des entités et d'éffleurer les possibilités du gestionnaire Kassia. Je dis bien "éffleurer" parce que vous n'avez encore rien vu... La deuxième partie du tutorial qui comporte les ateliers de 17 à 32 vous permettra de parcourir la quasi totalité des aspects de Kassia en faisant intéragir des entités de plus en plus complexes, vous permettant de réaliser des fonctionnalités de plus en plus riches et intéressantes. Nous continuerons naturellement à développer des entités avec le pseudo "charly" mais l'accent sera mis sur la réutilisation d'entités existantes, ceci afin de vous convaincre d'un aspect primordiale de la programmation gus05 : **la réutilisation massive des entités existantes**. Et bien évidemment, il est inutile de connaître ou maitrîser le code source des entités d'autres développeurs pour être capable de s'en servir et de les faire intéragir au sein d'une application. Sachant que j'ai développé plus de 3000 entités différentes, vous serez sans doute intéressé de pouvoir en tirer parti en temps voulu sans avoir à vous soucier des milliers de ligne de code que j'ai pu écrire pour en arriver là. ^^

L'objectif de la deuxième partie va être principalement de vous donner un niveau de maitrise suffisant du gestionnaire Kassia pour vous permettre de l'exploiter et de créer et tester toutes les entités que vous souhaitez. Au terme des 32 ateliers, vous devriez donc être en mesure de démarrer vos propres développements, d'exploiter ceux des autres et éventuellement de participer aux chantiers du projet gus05.