Documentation du projet de Conception de Logiciels Extensibles

Groupe 5 - CLEF

Table des matières

Installation

Importer le projet

1. Cloner le dépot https://github.com/ErrOrnAmE/CLEF.git ou désarchiver le fichier .zip
2. Importer le projet dans votre IDE
3. Configurer la version de Java pour utiliser la version 1.8
4. **Ajouter la librairie Gson** se situant dans le dossier “lib”
5. - Si une erreur “Impossible de trouver la classe principale ‘framework.Framework’” est indiqué, essayez de nettoyer votre workspace et de le réimporter
6. Compiler le projet

Lancement et utilisation de l’application

Une fois l’application lancée, vous vous retrouverez face à deux fenêtre: Le chat et le moniteur d’extensions.

La fenêtre de chat

Voici deux scénarios type d’utilisation de l’application: Vous pouvez indiquer un pseudonyme, l’adresse du chat et le port du chat. Vous pouvez ensuite choisir d’être serveur ou d’être client. Une fois que vous aurez cliqué sur le bouton “Connect”, vous pourrez envoyer et recevoir des messages sur ce chat. Vous pourrez enfin vous déconnecter du chat.

Si vous fermez la fenêtre, qui est une extension, elle demandera à quitter l’application.

Le moniteur d’extensions

La liste des extensions est affichée dans cette fenêtre avec 3 colonnes: “Type”, indiquant le type d’extension; “Name”, indiquant le nom de l’extension; et “Status” indiquant le statut de l’extension. 3 boutons sont disponibles en bas de la fenêtre: “Exit app” qui demande à quitter l’application; “Load” permettant de charger une extension; et “Kill” permettant d’arrêter une extension.

Si vous fermez la fenêtre, qui est aussi une extension, l’application continuera de s'exécuter.

Explications et notes

Au lancement de l’application, nous pouvons remarquer que toutes les extensions sont “Loaded”, excepté “ConsoleGUI”. En effet, “SimpleGUI”, “Logger” et “Monitor” sont toutes les trois “autorun”. De plus, “SimpleGUI”, à son démarrage, demande à “TCPClient” si l’application

est connecté à un serveur; et demande à “TCPServer” si l’application possède un serveur lancé. Cela explique donc pourquoi ces deux dernières extensions sont “Loaded”.

Vous pouvez tester le hotswap de composants avec le scénario suivant: Lancer deux applications (un serveur et un client). Sur une de ses applications, vous pouvez “kill” SimpleGUI et “load” ConsoleGUI. Vous pourrez voir que la connection entre les applications aura été maintenu, et vous pourrez continuez à discuter sur le chat.

Si vous souhaitez vous déconnecter depuis ConsoleGUI, vous pouvez écrire “/exit”.

La Javadoc complète du projet est disponible dans le dossier “doc”.

Le Framework

Les classes du Framework

Le package **framework** rassemble toutes les classes et interfaces permettant à la plateforme de s'exécuter.

La classe **Framework** est la pièce maîtresse de la plateforme: elle s'occupe de charger les fichiers de configuration, pré-charger les différentes extensions et proposer une API (grâce à un ensemble de méthodes statiques) pour accéder à ses différentes fonctionnalitées. Au lancement du programme, le Framework effectue les étapes suivantes:

1. Charge la configuration ("application.json")
2. Charge les dépendances; les extensions qui seront nécessaires
3. Exécute les extensions qui sont déclarés en "autorun" dans leurs configuration

La classe **Config** permet de gérer les fichiers de configuration au format json. Il s'agit simplement d'accesseur et mutateurs.

L'interface **IExtension**, implémenté par toutes les extensions, indique 3 méthodes à redéfinir si besoin: start(), stop() et handleEvent(). Les deux premières seront exécutés, respectivement, au chargement et à la terminaison de l'extension. La dernière permet de gérer un nouvel événement. (Plus d'informations sur les événements dans un chapitre suivant)

La classe **ExtensionContainer** est un handler permettant d'encapsuler une extension avec un Proxy. Ce handler gère le chargement et la terminaison de l'instance de l'extension.

La classe **Status** rassemble statiquement les différents états d'une extension.

L'interface **IExtensionActions** permet d'indiquer les méthodes appelées sur le proxy, mais gérées ExtensionsContainer, tel que load(), kill(), getStatus(), etc. Cette classe est "package protected" pour ne pas laisser la possibilité à des classes en dehors du Framework d'accéder à ces méthodes.

La classe **Event** définit les événements qui sont utilisés pas les extensions et le framework pour échanger des informations. (Plus d'informations sur les événements dans un chapitre suivant)

Les extensions

Une extension peut être défini comme répondant à un besoin, défini par une interface (Ex: "IMonitoring"). Ces interfaces sont réunis dans le package "interfaces", et il est possible d'en rajouter de nouvelles. Plusieurs extensions peuvent répondre au même besoin.

Chaque extension est situé dans son propre package. Dans ce package, se situent: une classe implémentant une de ces interfaces et un fichier de configuration (config.json).

Chaque extension est proxifié pour permettre un chargement paresseux de son instance.

Une extension peut avoir l'attribut "autorun", lui permettant d'être chargé dès le lancement du programme. (Plus d'explication dans un chapitre suivant)

Cycle de vie d'une extension

Au chargement d'une extension, la méthode start() sera appelée pour permettre à une extension de démarrer tous ses composants internes.

A la terminaison d'une extension, la méthode stop() sera appelée pour permettre l'arrêt "gracieux" de cette extension. (Ex: fermer les sockets, fermer les fenêtres)

Quand une extension possède les statuts "KILLED" ou "ERROR", et qu'une autre extension essaye de l'utiliser, le framework essaye de re-charger l'extension. Nous aurions aussi pu faire le choix de lever une exception, mais, après une longue réflexion, nous pensions qu'il était préférable de tenter le rechargement pour essayer de garantir l'intégrité de l'application.

Liste des interfaces

Voici la liste des interfaces que nous avons utilisé en exemple pour notre application de chat. La javadoc est disponible pour une documentation plus détaillée de leurs méthodes.

- **IMonitoring**: ces extensions permettent de monitorer l'application
- **INetworkServer**: ces extensions doivent pouvoir créer un serveur
- **INetworkClient**: ces extensions doivent pouvoir se connecter à un serveur
- **IGUI**: ces extensions propose une interface utilisateur graphique

Liste des extensions

Voici la liste des extensions que nous avons créées en exemple pour notre application de chat. La javadoc est disponible pour une documentation plus détaillée de leurs méthodes.

- **Logger** (IMonitoring): Log dans la console tous les événements de l'application
- **Monitor** (IMonitoring): Permet de load et kill les extensions
- **TCPServer** (INetworkServer): Implémentation TCP d'un serveur de chat
- **TCPClient** (INetworkClient): Implémentation TCP d'un client de chat
- **SimpleGUI** (IGUI): Interface de base du chat
- **ConsoleGUI** (IGUI): Interface de chat sans boutons

L'application

Avec ce framework, une application est un ensemble d'extensions, dont certaines en autorun. Cette application peut être configuré avec le fichier "application.json" à la racine du projet avec la configuration suivante:

```
{
      "name": "Ma super appli",            //Nom de l'application
      "description": "Lorem ipsum...",     //Description de l'application
      "extensions": [                      //Liste des extensions nécessaires
            "monitor",
            "extensionA",                  //Noms des packages
            "extensionB"
      ]
}
```

Les événements

Le framework propose un système d'événement pour gérer les échanges entre différentes extensions. Un événement est l'association d'un nom (String) et d'un payload (Object).

Créer un événement

Pour "lancer" un événement, cela se fait en une seule ligne:

```
Framework.event("message.received", "Bonjour, comment vas-tu?");
```

Recevoir un événement

Pour recevoir un événement, l'extension qui veut souscrire à un type événement (exemple: "message.received"), doit ajouter la ligne suivante (dans la méthode start() par exemple):

```
Framework.subscribeEvent("message.received", this);
```

Il doit ensuite redéfinir la méthode "handleEvent" (exemple avec "message.received"):

```
@Override
public void handleEvent(Event event) {
    if(event.is("message.received")){
```

```
        System.out.println("Le message suivant a été
reçu:"+(String)event.getPayload());
    }
}
```

La méthode “is()” permet de tester quel est le type d’événement qui est reçu. Les événements peuvent avoir plusieurs “niveaux” de catégorie et il est possible d’utiliser un “wildcard” pour gérer plusieurs type d’événements.
Exemples:

```
Event event = new Event("network.client.connected", null);

event.is("network.client.connected");      // True
event.is("network.client.disconnected");   // False
event.is("network.client");                     // True
event.is("network.server");                     // False
event.is("network");                       // True
event.is("network.*");                     // True
event.is("network.*.connected");                // True
event.is("network.*.disconnected");        // False
```

Il est aussi possible d’utiliser les catégories et les wildcards pour souscrire à un événement.

Arrêter de recevoir un événement

Pour arrêter de recevoir un événement, il faut appeler une autre méthode du Framework:

```
Framework.unsubscribeEvent("message.received", this);
```

Créer une extension

Pour créer une extension, il suffit de créer un package, une classe principale et un fichier de configuration. Pour l'utiliser dans une application, il ne faudra pas oublier d'ajouter le nom du package dans "application.json".

Classe principale

Voici un exemple de template pour votre classe principale:

```
public class MonExtension implements IMonInterface {

	public void start() {
		// Do stuff
	}

	public void stop() {
		// Do stuff
	}

	public void handleEvent(Event event) {
		// Do stuff
	}
}
```

Configuration

La configuration d'une extension se fait dans le fichier config.json de la façon suivante:

```
{
	"name": "MonExtension",
	"type": "IMonInterface",
	"description": "Cette extension fait des trucs",
	"autorun": false,
	"killable": false
}
```

Les paramètres sont les suivants:

- name: Le nom de la classe principale de l'extension
- type: Le nom de l'interface qui est implémenté
- description: Description de l'extension
- autorun: Indique si l'extension doit être lancé au lancement de l'application

- killable: Indique si l'extension peut être "killé" durant l'exécution de l'application

API du Framework

La classe **Framework** propose un ensemble de méthodes statiques (classées par "thèmes") pour accéder à des fonctionnalitées de la plateforme:

Application

void exit()

Quitte le programme
Essaye d'arrêter gracieusement chacune des extensions avant de quitter le programme

Config getConfig()

Récupère la configuration de l'application
Permet de récupérer l'objet Config de l'application ('application.json')

Extensions

getExtensions()

Récupère toutes les extensions

Signature:

```
public static Map<Class<?>,Map<Class<?>,IExtension>> getExtensions()
```

Exemple:

```
Map<Class<?>,Map<Class<?>,IExtension>> extensions = Framework.getExtensions()
```

getExtension(Class<T> extensionInterface)

Récupère une extension
Récupère la **première** extension disponible en fonction de l'interface donnée en paramètre

Signature:

```
public static <T extends IExtension> T getExtension(Class<T> extensionInterface)
```

Exemple:

```
IAffichage affichage = Framework.getExtension(IAffichage.class);
```

get(Class<T> extensionInterface)

Récupère une liste d'extension
Récupère la liste des extensions disponible en fonction de l'interface donnée en paramètre

Signature:

```
public static <T extends IExtension> List<T> get(Class<T> extensionInterface)
```

Exemple:

```
List<IAffichage> affichages = Framework.get(IAffichage.class);
IAffichage affichage = affichage.get(0);
```

getExtensionConfig(IExtension extension)

Récupère la configuration d'une extension

Signature:

```
public static Config getExtensionConfig(IExtension extension)
```

Exemple:

```
IAffichage affichage = Framework.getExtension(IAffichage.class);
Config affichageConfig = Framework.getExtensionConfig(affichage);
```

getExtensionStatus(IExtension extension)

Récupère le statut d'une extension
Voir la classe framework.Status

Signature:

```
public static String getExtensionStatus(IExtension extension)
```

Exemple:

```
IAffichage affichage = Framework.getExtension(IAffichage.class);

if (Framework.getExtensionStatus(affichage).equals(Status.KILLED)) {
    // Do something
}
```

loadExtension(IExtension extension)

Charge une extension

Signature:

```
public static boolean loadExtension(IExtension extension)
```

Exemple:

```
IAffichage affichage = Framework.getExtension(IAffichage.class);

Framework.loadExtension(affichage);
```

killExtension(IExtension extension)

Kill une extension

Signature:

```
public static boolean killExtension(IExtension extension)
```

Exemple:

```
IAffichage affichage = Framework.getExtension(IAffichage.class);

Framework.killExtension(affichage);
```

isKillable(IExtension extension)

Indique si une extension peut être killed

Signature:

```
public static boolean isKillable(IExtension extension)
```

Exemple:

```
IAffichage affichage = Framework.getExtension(IAffichage.class);

If (Framework.isKillable(affichage)) {
    Framework.killExtension(affichage);
}
```

Evénements

event(String name, Object payload)

Surcharge de event(Event event)

Signature:

```
public static void event(String name, Object payload)
```

Exemple:

```
Framework.event("message.received", "Hey, how are you?");
```

event(Event event)

Déclare un nouvel événement

Signature:

```
public static void event(Event event)
```

Exemple:

```
Event anEvent = new Event("message.received", "Hey, how are you?");
Framework.event(anEvent);
```

subscribeEvent(String name, IExtension handler)

Souscrit à un type d'événement
Demande à être notifié à chaque fois qu'un événement d'un certain type est déclaré

Signature:

```
public static void subscribeEvent(String name, IExtension handler)
```

Exemple:

```
Framework.subscribeEvent("message.received", this);
```

unsubscribeEvent(String name, IExtension handler)

Désouscrit à un type d'événement
Demande à ne plus être notifié à chaque fois qu'un événement d'un certain type est déclaré

Signature:

```
public static void unsubscribeEvent(String name, IExtension handler)
```

Exemple:

```
Framework.unsubscribeEvent("message.received", this);

Framework.unsubscribeEvent("*",this); // Se désouscrit de tous les événements
```

Annexe: Modifications apportées

Suite aux rapports d'analyse de projet effectués par nos camarades, nous avons fait les modifications suivantes. Ces modifications sont classés par thème. A chaque remarque est associée l'explication des modifications apportées.

Framework

Légère incompréhension des méthodes des interfaces pour les extensions

Changement des noms des méthodes + ajout de documentation

Le package contenant les interfaces des extensions ne devrait pas être dans le package du framework

Déplacement des interfaces dans le package "interfaces"

Besoin d'une meilleure gestion des exceptions et des erreurs

Ajout de nombreux try-catch, de conditions et d'events

Une extension "killed" continue de s'executer

Ajout de la méthode "stop()" dans IExtension qui permet de terminer gracieusement l'extension

Application

Ralentissement de la machine (memory leak?)

Cela venait d'une mauvaise utilisation d'un Scanner

Problèmes de déconnexion pas toujours effective / levant des exceptions

Envoi d'un message à la déconnexion du serveur pour permettre aux clients de fermer leurs sockets

Griser le champ "port" à la connexion

Refactoring de l'interface graphique

Besoin d'une gestion des erreurs

Refactoring de l'interface graphique, qui gère mieux les erreurs

Monitoring

Si MonitoringIHM est fermé, ne doit pas fermer l'application

A la fermeture du Monitor, sa propre extension est killed, mais l'application continue. (Au contraire à la fermeture de la fenêtre de l'interface graphique utilisateur, l'application sera fermée)

Le refresh du MonitoringIHM devrait être automatique

Fix du bug qui empêchait le rafraîchissement automatique du moniteur

Afficher les logs dans la console

Une nouvelle extension (Logger) fait son apparition et log tous les événements de l'application

Documentation

Quelques problèmes de maven et classpath: à régler

Suppression du pom.xml et ajout de la librairie Gson dans le repository. Il aurait fallu trouver une meilleure solution, mais le temps nous a manqué

Manque de documentation sur certaines classes

Javadoc complète de toutes les classes, même les non-publiques

Ajouts supplémentaires

Il s'agit des ajouts qui ne découlent pas de remarques trouvées dans les rapports d'analyse, mais que nous avons tout de même décidé d'installer.

Meilleure gestion du cycle de vie des extensions

Les extensions ont maintenant une méthode "start()" et une méthode "stop()"

Utilisation du mot-clé "default" pour rendre le code un peu plus propre

Permet de ne pas avoir besoin de redéfinir les méthodes de l'interface IExtension si il n'y a pas besoin (voir framework.IExtension)

Refactoring des extensions

Un code plus clean et qui suit plus la logique du Framework. Avec une utilisation plus avancée des événements.

Refactoring de la documentation utilisateur

Javadoc complète

Ajouter une hiérarchie des noms des events

Ajout des catégories et du "wildcard": voir la documentation des events.

Changement du nom du fichier config de base

Pour éviter la confusion entre les fichiers de configuration des événements (“config.json”) et celui de l’application (“config.json” aussi, maintenant: “application.json”)

Merge Monitoring et MonitoringIHM

L’extension Monitoring ne servait plus beaucoup après l’ajout de fonction dans l’api du Framework. La nouvelle extension “Monitor” est donc le merge des deux.

Enlever l’utilisation du cast avec l’api Framework (getExtension())

Utilisation de type générique dans les fonctions getExtension() (“<T extends IExtension> T”, voir framework.Framework)