



09/12/2019

Architecture logicielle

Projet d'architecture à plugin



BLANCHARD - GILLET - GONZALEZ - LOISEAU
M2 MIAGE - ISI

Table des matières

I.	Introduction	2
II.	Installation de l'application	3
III.	Procédure d'ajout d'un plugin	6
IV.	Architecture de l'application	10
V.	Conclusion	12

Lien vers le code source : https://github.com/aldvine/archi_logiciel

I. Introduction

Dans le cadre du module 'Architecture Logicielle' nous devons réaliser une architecture permettant de pouvoir importer des plugins au sein d'une application. L'objectif de ce projet étant de nous faire manipuler les différents concepts de JAVA tout en prenant en main les pattern strategy et MVC.

En effet, nous devons réaliser une application fonctionnant avec des plugins. Ces plugins peuvent permettre de personnaliser l'application ou encore d'y apporter des améliorations. Pour réaliser ce projet nous avons décidé de réaliser une application fonctionnant avec des personnages qui possèdent des attributs (santé, force etc...) Ces attributs peuvent être affichés et modifiés grâce aux plugins et les personnages peuvent se confronter à un combat. Afin de pouvoir travailler ensemble, nous avons mis en place l'outil de gestion de version git. Cet outil nous permet de suivre l'avancement du projet et de pouvoir travailler à distance.

Dans ce rapport, nous allons présenter la mise en place de notre application, son fonctionnement, ainsi que la procédure d'ajout d'un plugin. Puis nous allons expliquer nos choix concernant l'architecture de notre application, pour finir sur une conclusion de la réalisation de ce projet.

II. Installation de l'application

Pour initialiser notre projet, il faut d'abord importer le projet sur votre ordinateur. Pour cela, il faut ouvrir une console ou un terminal puis se placer dans le dossier où vous souhaitez importer le projet. Une fois que vous êtes situé dans le répertoire de votre choix, vous pouvez exécuter la commande suivante :

Git clone https://github.com/aldvine/archi_logiciel.git

Ensuite, pour exécuter notre projet, il faut posséder l'IDE Eclipse. Cet IDE va permettre de récupérer les fichiers sources de notre application. Pour cela, allez dans l'onglet 'File' puis sélectionnez 'Open Projects from File System...'. Vous arrivez sur la fenêtre suivante :

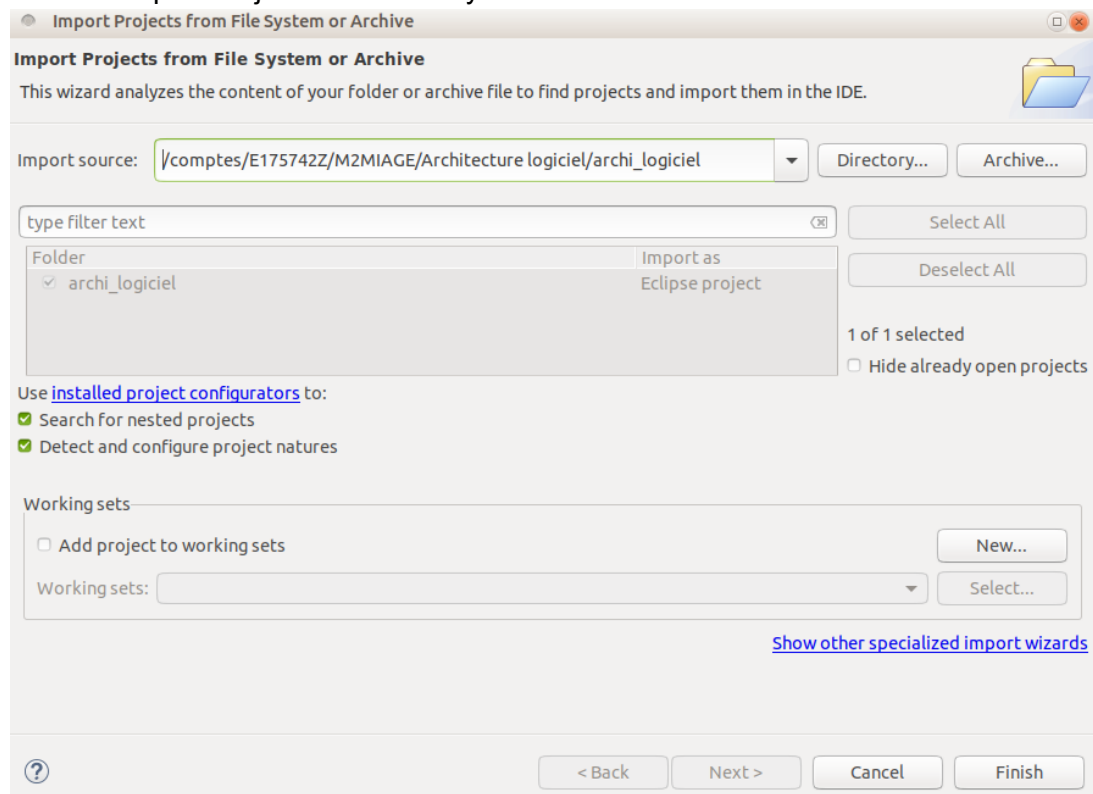


Figure 1 : Choix de l'import du projet

Sur cette fenêtre, choisissez le répertoire dans lequel vous avez importé le projet comme sur la figure ci-dessus. Ensuite, cliquez sur 'Finish'.

Notre projet est maintenant importé dans Eclipse, vous obtenez les sources suivantes :

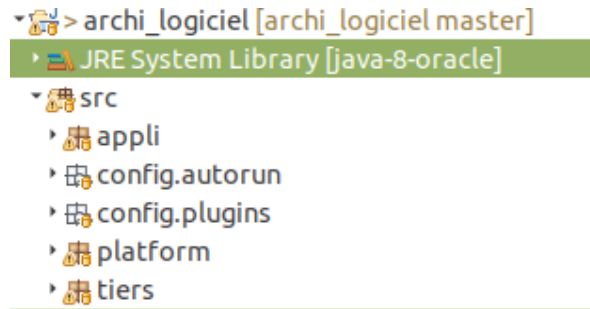


Figure 3 : Sources de l'application

Vous avez dès à présent importé notre projet, vous pouvez commencer par l'exécuter en sélectionnant la racine du projet, puis cliquez sur le bouton Run (CTRL + F11). Une fois que le projet est compilé, une fenêtre s'affiche. Elle est composée de 3 parties :

- **Action sur les personnages :** Notre application utilise 2 personnages que l'on peut confronter. Pour faire cette confrontation, l'utilisateur peut charger les deux personnages grâce à la partie située en haut de la fenêtre (voir Figure 4). Ainsi, l'utilisateur peut charger un personnage, voir le détail de ce personnage (santé, force, ...), et modifier le personnage. Il y a également deux autres boutons qui permettent de confronter les deux personnages et de voir le détail des deux personnages. Les parties concernant l'affichage des personnages les modifications des personnages et la confrontation sont gérées par les plugins qui sont chargés par l'application.

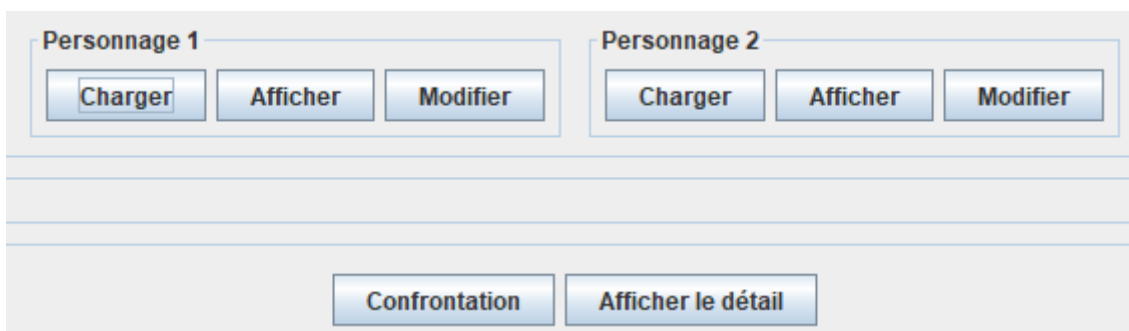


Figure 4 : Utilisation des personnage

- **Suivi des actions de l'application** : Pour comprendre le fonctionnement de l'application et savoir quel plugin est utilisé, nous avons mis en place un espace information qui permet d'avoir une trace des différentes actions qui sont exécutées lors de l'utilisation de notre application. (Chargement de plugin, activation de plugin, erreurs, etc...)

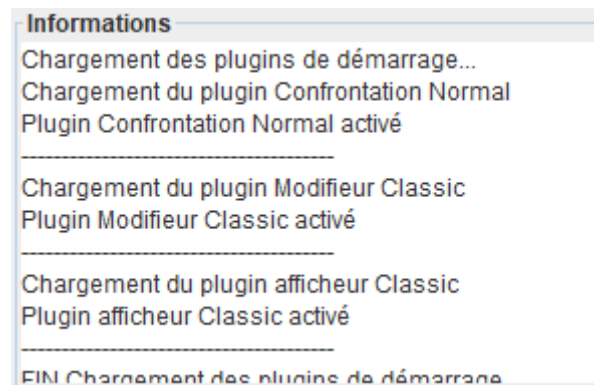


Figure 5 : Écran d'affichage des instructions

- **Chargement des plugins** : Pour charger des plugins, nous avons mis en place des boutons qui permettent de choisir les plugins que l'on souhaite utiliser. Pour le moment, nous avons créé que 2 plugins pour modifier les attributs des personnages, 2 plugins pour changer l'affichage des détails des personnages, et 1 plugin pour gérer la confrontation. L'ajout de boutons de plugin est fait automatiquement par l'application lorsqu'elle détecte des fichiers de configuration d'un plugin.



Figure 6 : Liste des plugins existant dans l'application

III. Procédure d'ajout d'un plugin

Nous avons ajouté des plugins de base, mais il est possible d'en ajouter des nouveaux permettant de faire évoluer les comportements de l'application. Pour cela, il faut que le plugin implémente une des interfaces suivantes :

Chaque interface a pour paramètre deux classes. Pour le fonctionnement de la classe JPanel, se référer à la documentation JAVA. Pour la classe Character, les attributs suivant sont accessible via des getters et setters:

class Character

```
String nom;  
Integer sante;  
Integer force;  
Integer intelligence;  
Integer agilite;  
String origin;  
String status;
```


- IAfficheur.java : l'interface permettant d'effectuer toutes tâches d'affichage
 - **void affiche(JPanel panel, Character c)** permet d'afficher les informations d'un personnage.
 - **void afficheDetail(JPanel panel, Character c1, Character c2)** permet d'afficher les informations de chaque personnage.

La partie affichage des personnages est intégralement gérée par le plugin d'affichage. Les plugins doivent utiliser l'objet JPanel pour l'affichage sur l'application. Cela permet aux plugins d'afficher des zones de texte, ou de créer un affichage plus évolué pour l'application.

- IBattle.java : l'interface permettant d'effectuer des tâches de combat
 - **void battle(Character c1, Character c2)** permet d'effectuer un combat entre les deux personnages. Il peut s'agir d'un affrontement ou d'un combat entier selon le choix du plugin.
- IModifier.java : l'interface permettant d'effectuer des modifications sur les personnages
 - **void modifier(Character c)** permet de modifier des attributs d'un personnage passer en paramètre. C'est le plugin sélectionné qui modifie les statistiques du personnage.

Ensuite, il faut ajouter le plugin dans le package *tiers* et la configuration dans le dossier *config/plugins*. Le fichier de configuration doit respecter le schéma suivant :

- **id** correspondant à l'identifiant unique du plugin (*exemple : Modifier2*)
- **className** correspondant au chemin vers la classe (*exemple : tiers.ModifierUpgraded*)
- **iface** correspondant à l'interface qu'il implémente (*exemple : appli.IModifier*)
- **name** est le nom du plugin qui sera affiché (*exemple : SOIN*)
- **description** détail ce que fait le plugin (*exemple : Augmente santé et force*)
- **dependencies** correspond à une liste d'identifiant de plugins nécessaires qui seront chargé avant. Les plugins sont séparés par des points-virgules. (*exemple : Afficheur1*)

 Il faudra porter attention à ne pas créer de boucle dans les dépendances entre les différents plugins.

Certains plugins peuvent être chargés au **démarrage de l'application**. Pour cela, il faudra compléter le fichier *autorun.txt* qui se trouve dans le dossier *config/autorun/* de l'application en ajoutant l'ordre de chargement et l'identifiant du plugin concerné.

Exemple :

```
1=Afficheur1
2=Modifier2
3=Battle1
```


Exemple d'ajout d'un nouveau plugin

Dans cet exemple, nous allons modifier l'affichage, nous allons montrer pas à pas comment ajouter un nouveau plugin à l'application

1. Créer une nouvelle classe dans le **package tiers**
2. Importer l'interface de votre choix, ici nous importons **IAfficheur**

```
package tiers;
import javax.swing.JPanel;

import appli.Character;
import appli.IAfficheur;
public class AfficheurNouveau implements IAfficheur{

    @Override
    public void affiche(JPanel panel, Character c) {
        // TODO Auto-generated method stub

    }

    @Override
    public void afficheDetail(JPanel panel, Character c1, Character c2) {
        // TODO Auto-generated method stub

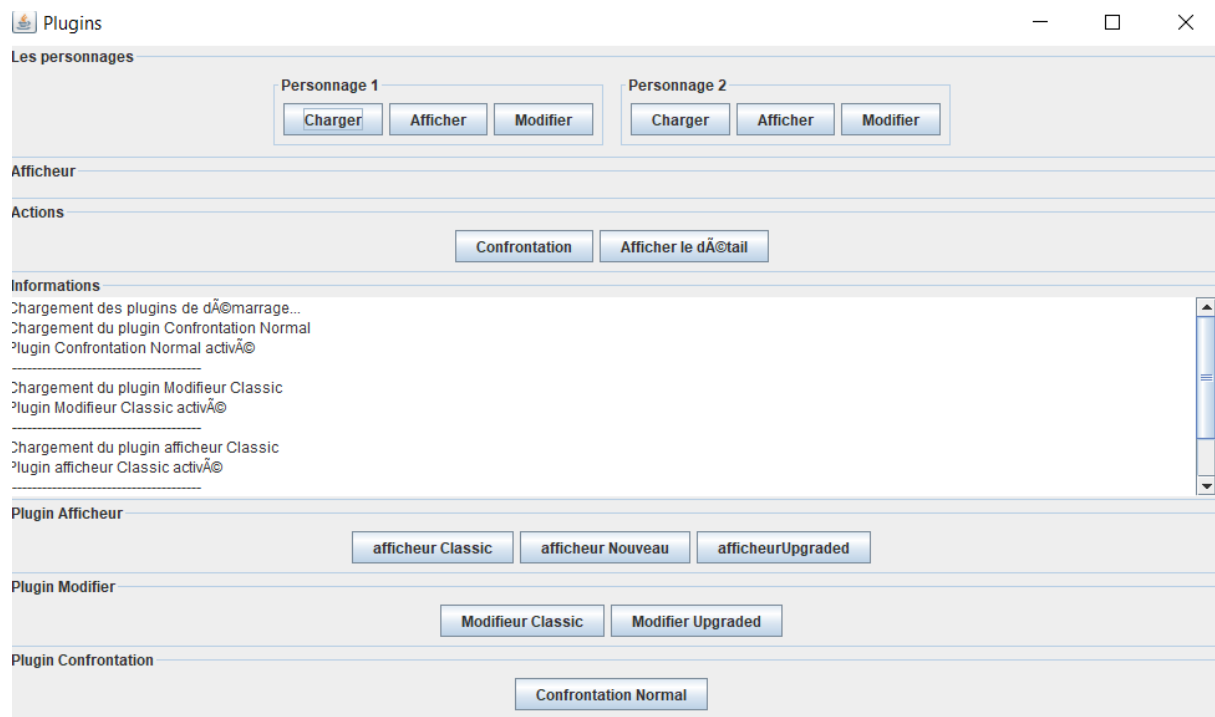
    }

}
```

3. Ecrire le code d'implémentation du nouveau plugin
4. Créer un fichier de configuration dans le dossier *config/plugins/* en respectant le schéma d'un fichier de config. Exemple :

```
id=Battle1
className=tiers.BattleClassic
iface=appli.IBattle
name=Confrontation Normal
description=Joue sur la force et l'agilite
dependencies=Afficheur1
```

5. Lancer l'application et charger le plugin que vous venez d'écrire



IV. Architecture de l'application

Pour faciliter le développement et la structure de notre application, nous avons séparé les différentes classes dans différents packages.

Ci-dessous le diagramme UML qui résume notre architecture logicielle et donne une meilleure visibilité de l'ensemble de notre projet :

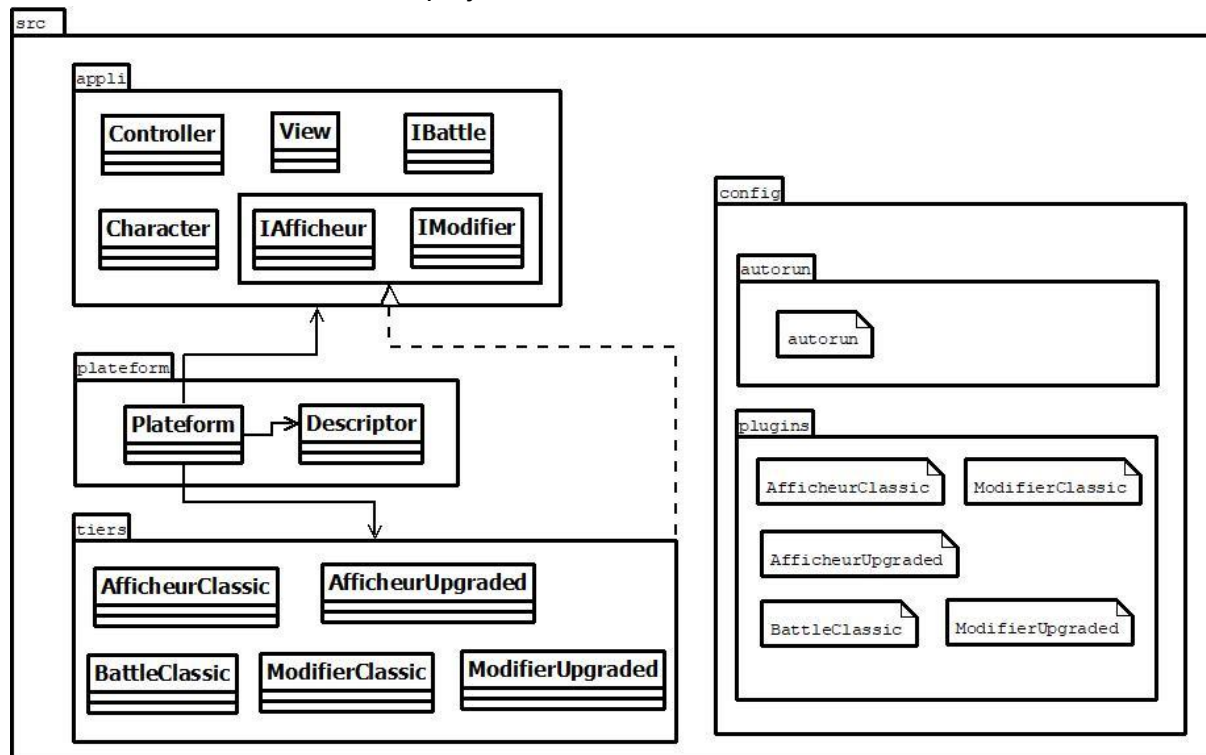
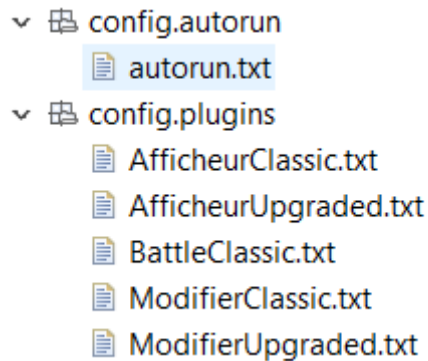


Figure 7 : Diagramme de l'architecture logicielle de notre projet

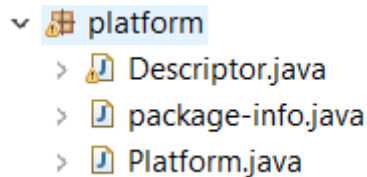
- **appli** : Ce package contient une application suivant une architecture MVC, où le controller gère aussi l'import des plugins en faisant appel à la classe platform et ajuste la vue.

```
▼ appli
> App.java
> Bouton.java
> Character.java
> Controller.java
> IAfficheur.java
> IBattle.java
> IModifier.java
> View.java
```

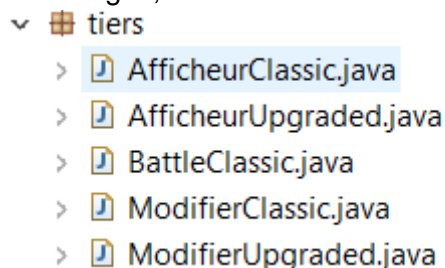
- **config** : Ce dossier contient les fichiers textes qui permettent de décrire les plugins à importer. Dans le dossier 'config', on distingue deux sous-dossiers : autorun et plugins. Le dossier autorun contient le fichier autorun.txt qui indique les plugins à charger au démarrage de l'application pour pouvoir utiliser les personnages par la suite. Ensuite, il y a le dossier plugins qui regroupe l'ensemble des fichiers de configs qui décrit chacun des plugins de l'application.



- **platform** : Le package platform contient les deux classes qui vont interagir avec les plugins. La classe Platform.java va permettre de charger les plugins et la classe Descriptor.java va permettre de récupérer les informations des plugins chargés en fonction des informations saisies dans les fichiers de config des plugins.



- **tiers** : Dans ce package on regroupe les classes qui vont implémenter les interfaces de notre application. Chacune de ces classes est liée à un plugin, se sont ces classes qui vont permettre à l'application d'avoir un comportement différent. En effet, chacune de ces classes vont avoir un comportement différent sur l'affichage, les caractéristiques des personnages, et la confrontation des personnages.



V. Conclusion

Ce projet nous a permis de mieux comprendre le fonctionnement d'une architecture logicielle qui est ouverte à l'implémentation de divers plugins. En effet, nous avons appris à réaliser une application qui peut utiliser des plugins différents afin de changer le comportement de l'application. Pour cela, nous avons dû créer des interfaces qui sont ensuite implémentées par les plugins pour changer la conduite des programmes. Ainsi, chaque utilisateur peut personnaliser l'application avec les plugins qu'ils souhaitent.

Lors de la réalisation de ce projet, nous avons eu diverses difficultés dont la création de l'IHM permettant de faire interagir l'utilisateur grâce à des boutons. En effet, nous n'avons jamais réalisé d'IHM en JAVA avant ce projet, nous avons donc cherché des solutions sur internet afin de répondre à notre besoin. De plus, nous avons eu certaines difficultés à prendre en main le langage JAVA, car nous étions très peu à avoir de l'expérience dans ce langage.

Pour finir, ce projet nous a permis de voir une approche différente sur l'architecture logicielle. En effet, nous n'avons jamais eu l'occasion de créer une application comme celle-ci en entreprise et à l'université.

Perspective d'amélioration de l'application :

- Rechercher une solution pour ajouter un plugin sans avoir à redémarrer l'application lorsque l'on ajoute des plugins supplémentaires.