

Rapport Projet Pilote Clavier-Ad1a

Ayadi Jana, Gorce-Neiva-da-Costa Alexandre, Hamme Laure, Ranjalahy Elisa, Singh Navdeep, MI1

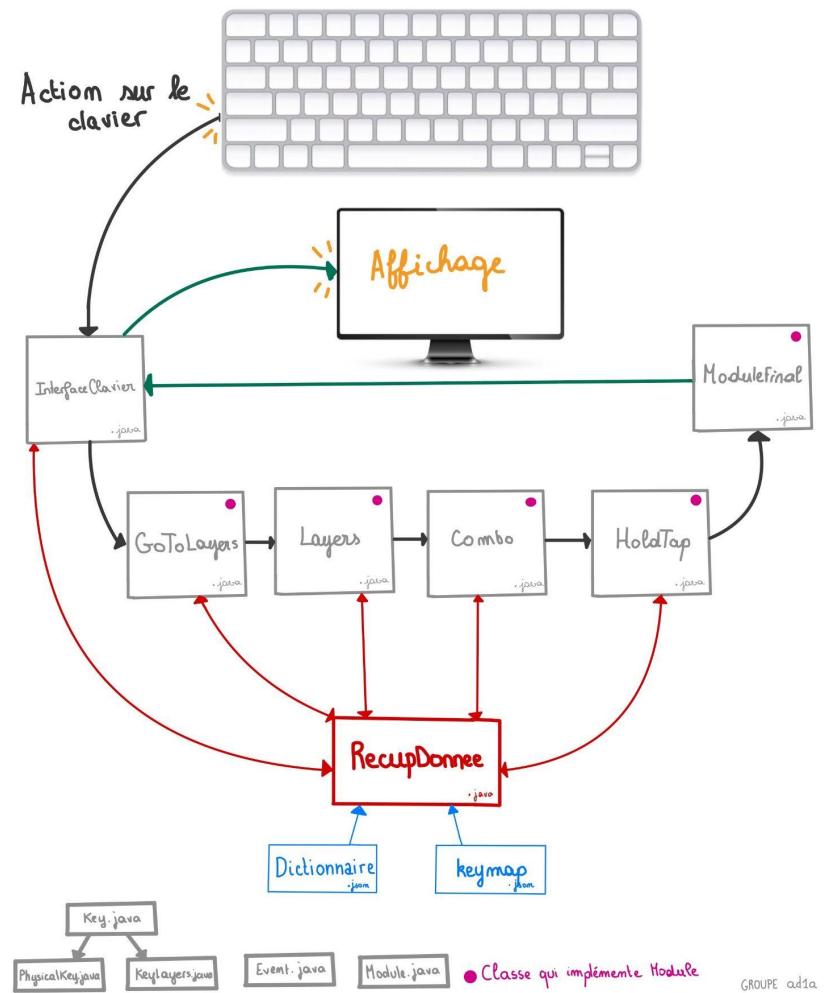
Dans le cadre de nos études universitaires en mathématiques et informatique, nous avons été amenés durant notre seconde année de licence à travailler sur un projet de programmation en java. Notre objectif a été d'implémenter un pilote clavier.

Ce projet nous faisait totalement sortir de notre zone de confort, c'était une approche et un raisonnement loin de ce dont nous avions l'habitude de coder. Nous avons divisé notre travail : d'abord implémenter notre clavier avec une interface graphique Swing puis par la suite implémenter le pilote.

I. Le concept et les grandes lignes de notre projet

Considérons qu'un individu écrit sur un clavier d'ordinateur : il tape sur une touche et immédiatement un effet est produit sur l'écran. Nous avons eu pour tâche de modifier ce système en agissant sur ce qui se passe entre l'entrée (action de l'utilisateur) et la sortie sur l'écran, dans le but d'ajouter de nouvelles fonctionnalités ou de modifier celles déjà existantes.

Organisation des classes



La classe principale de notre projet, celle qui contient le main, *InterfaceClavier*, intercepte les entrées claviers, et après le parcours des autres classes intermédiaires, génère la sortie désirée. Elle étend la classe JFrame de Swing et possède un attribut JTextField qui permet la saisie et l'affichage du texte. Pour ce faire, elle implémente l'interface KeyListener, dont les fonctions KeyPressed et KeyReleased nous permettent de récupérer les événements provoqués par l'utilisateur au clavier, qui peut être l'appui ou le relâchement d'une touche.

Remarque : Puisque notre projet a pour but de modifier les événements claviers, nous devons "annuler" leur comportement par défaut. C'est pour cela qu'à la réception d'un événement, nous utilisons la méthode event.consume() de java.awt.event, qui bloque son effet. Notons que certains événements ne peuvent pas être atteints par cette méthode, c'est par exemple le cas de la touche *supprimer*.

Notre projet permet donc d'intercepter et de modifier tous les événements pouvant effectivement être consommés par cette méthode.

Dans le cas d'un tel événement (pouvant être consommé), nous utilisons les informations fournies par le keyEvent récupéré pour créer une instance de notre propre classe Event. Elle est caractérisée par :

- Un attribut *pressTime* de type long, qui représente le moment exact où l'événement a eu lieu.
- Un booléen *isReleased*, qui vaut true si l'événement est un relâchement de touche, false si c'est un appui
- Un attribut entier *final originalKey*, qui enregistre le keyCode de la touche concernée. En Java, un keyCode est un entier, unique, associé à une touche.
- Un objet nommé *key* de type Key. La classe abstraite Key et ses classes filles proposent une classification des keyCode associés à un événement.
 - une PhysicalKey a un attribut entier qui représente le keyCode mentionné au point précédent. La différence est que cet objet peut évoluer en un autre sous-type de Key.

Une fois cet objet Event créé, c'est le début pour lui d'un parcours de classe en classe, dont chacune en changera éventuellement les caractéristiques. Puis il reviendra, modifié ou non, à l'interface visuelle, pour être exécuté.

II. Quelles sont ces fameuses classes ? Que font-elles ?

Nous appelons ces classes des "modules", puisqu'elles implémentent une interface *Module*. Une instance d'*InterfaceClavier* possède ainsi une liste de modules, définissant le parcours que doit suivre chaque événement.

Une fois qu'un module a fini sa tâche, il enverra l'événement, modifié ou non, au module suivant de la liste. Voyons, dans l'ordre, les différents modules :

Modules numéros 0 et 1 : GoToLayer et Layers.

Bien que *GoToLayer* soit le premier module de la liste, nous allons d'abord nous intéresser au deuxième, *Layers*.

Les Layers permettent d'avoir plusieurs claviers en un seul. En bref, un layer est une couche du clavier, à l'instar des claviers de téléphones portables qui comportent une couche pour les lettres, une pour les ponctuations et les chiffres, une autre pour les emojis... Notre projet en comporte 4 dont voici les caractéristiques principales :

- **Layer 0** : reconstitution du clavier AZERTY.



- **Layer 1** : reconstitution du clavier QWERTY, célèbre concurrent du clavier AZERTY.



- **Layer 2** : clavier numérique.



- **Layer 3** : un clavier original qui confine l'ensemble des lettres dans une moitié de clavier et les ponctuations dans l'autre moitié. Nous nous pencherons ultérieurement sur la manière dont cela est rendu faisable (cf Hold-Tap).



ps : Se rendre à ★(quelques paragraphes plus bas) pour connaître la significations des touches L0,L1,L2 et L3.

La classe Layers contient un attribut de type `List<Map<PhysicalKey,KeyLayers>>`. Il s'agit d'une liste de map. Chacune des map correspond à la description d'un Layer. Grossièrement, sans rentrer dans les détails du code, ces maps ont pour rôle d'associer une touche à une autre. Par exemple, dans le Layer 1 (ie, QWERTY), nous associons à la touche A la touche Q. Ainsi, si l'utilisateur appuie sur la touche A en se trouvant dans le layer 1, c'est un Q qui sera écrit.

Lorsque ce module reçoit une événement, il effectue une recherche dans la map correspondant au Layer actuel.

S'il trouve un association, il modifie l'attribut `key` de l'événement en mettant à jour son `keyCode` avec le `keyCode` de la touche associée. Notons qu'une touche peut être associée à elle-même, dans le cas où l'on ne souhaite pas modifier son effet.

Autrement, si la recherche n'aboutit pas, c'est-à-dire, qu'aucune association n'a été définie, alors le parcours de l'événement s'arrête ici.

Afin de minimiser cet effet de "vide juridique", ie, de non-définition, nous avons choisi d'effectuer une recherche en profondeur. Si la recherche dans le layer numéro i ne donne rien, alors elle se poursuivra dans les Layers i-1, i-2, ..., 0.

Maintenant vous nous direz : comment passer d'un Layer à un autre ?

C'est ici que nous allons parler du tout premier module de la liste, le module GoToLayer. Lui aussi contient une liste, mais cette liste associe une touche à un numéro de Layer.

★ Ces touches sont représentées sur les schémas par les identifiants L0, L1, L2 et L3, déplaçant respectivement le système sur les Layers 0, 1, 2 et 3.

Si la key de l'événement reçu correspond à une de ces touches, alors l'attribut statique `currentLayer` de la classe Layers, qui indique le Layer dans lequel se trouve l'utilisateur, sera mis à jour. Le cas échéant, le parcours de l'événement s'arrête ici, sinon, il est transmis au module Layers, expliqué à l'instant.

Finalement l'événement est transmis du module Layers au module HoldTap.

Nous avons également implémenté des macros qui permettent d'écrire une phrase ou un mot entier en appuyant simplement sur une touche. Parmi les macros disponibles, on retrouve celle qui

insère la date et l'heure, ainsi qu'une autre qui nous offre la possibilité de supprimer tout le contenu de notre interface visuelle. Les macros s'intègrent parfaitement à notre projet car elles nous font gagner du temps en écrivant directement des mots ou des phrases que nous répétons souvent (par exemple, lors de la création d'une fonction "main").

Pour créer une nouvelle macro, il suffit simplement de lui attribuer une valeur de sortie dans le fichier Dictionnaire et de la mettre en relation avec une touche d'entrée dans le fichier Keymap. (cf partie 4)

Module numéro 2 : Combo

Ce module contient des listes (*ComboPossible* et *ResultCombo*) de touches qui, combinées entre elles, produisent un effet particulier. Par exemple, si l'on appuie assez rapidement sur les touches o puis e, cela écrira “œ” collé, ou encore p puis i qui écrira “π”.

Combo transmet ensuite l'événement, modifié ou non, au module suivant : HoldTap.

Module numéro 3 : HoldTap

Lorsque nous sommes arrivés au moment de la conception du module Hold-Tap il nous a été assez difficile de bien comprendre et de visualiser les étapes nécessaires. Pour rappel, un Hold-Tap correspond à un appui long (“hold”) sur une certaine touche, couplé simultanément à des clics simples (“tap”) sur d’autres touches.

Dans HoldTap, nous avons eu besoin, entre autres de :

- Une liste de *holdPossible* : celle-ci contient des Key, plus simplement, des touches, qui sont des “hold” pouvant, si longuement appuyées, avoir un effet particulier.
- Une liste *hold* contenant les touches actuellement maintenues en appui long
- Les listes *changedTo* associent des touches “hold” à d’autres touches, “tap” et définissent un effet dans le cas où la deuxième touche est cliquée pendant le “hold” de la première.

A la réception d'un événement, nous déterminons quel est son rôle.

S'il s'agit d'une touche pouvant être un “hold”, et que le temps d'appui est assez long, l'événement est ajouté à la liste *hold*.

Dans le cas où l'événement reçu est un “tap” modifié par un “hold”, nous déterminons son nouvel effet et transmettons l'événement au module suivant.

Lors de nos réunions, il nous avait été conseillé d'imaginer comment nous pourrions adapter le clavier à un utilisateur n'ayant qu'une seule main, et qui voudrait donc minimiser les déplacements de sa main sur le clavier. C'est ce qui a motivé la création du Layer 3, dans lequel toutes les lettres sont confinées dans une seule moitié de clavier : certaines sont accessibles par un clic simple, d'autres par la réalisation d'un Hold-Tap.

Après avoir été traité par le module Hold-Tap, l'événement est transmis au module final.

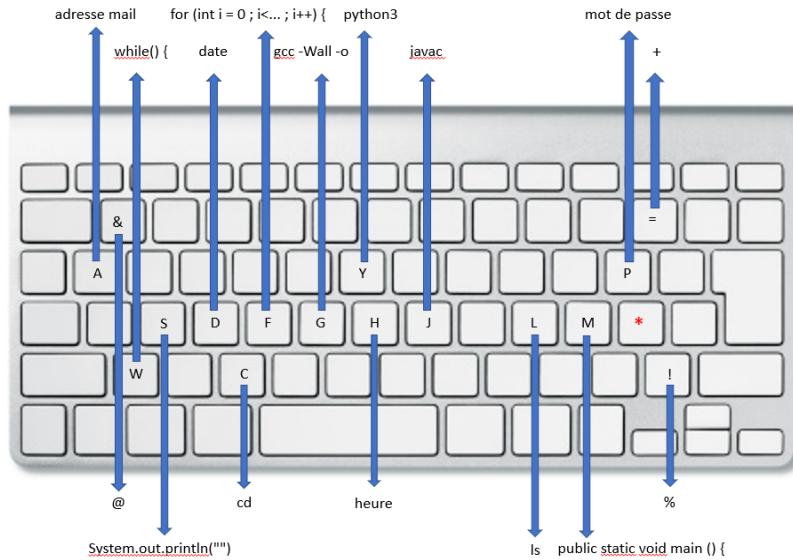
Voici des schémas des HoldTap possibles selon les layers (en rouge la touche Hold) :

dans Layer 0 :

HoldTap pour les majuscules



HoldTap pour les macros



HoldTap pour les accents



dans Layer 1

HoldTap pour les accents



dans Layer 3

HoldTap pour la moitié des touches du clavier



Module numéro 4 : Module Final.

Celui-ci a une seule tâche simple, il fait revenir l'événement, éventuellement modifié, à l'InterfaceClavier en appelant sa fonction executeAction(Event e).

Recevant l'événement résultant de toutes ces étapes, la méthode utilise le keyCode de l'événement pour aller chercher, dans l'attribut de type List *dictionnaire2*, à quel sortie ce keyCode correspond, et l'afficher dans la JFrame.

Ainsi, le parcours est terminé, la boucle est bouclée... jusqu'au prochain clic !

Nous vous avons beaucoup parlé de listes, mais comment sont-elles définies et gérées ?

III. Gestion et organisations des données

Toutes nos classes possèdent des listes, des maps ou des listes de maps. Ce sont précisément ces listes qui définissent les associations, combinaisons et substitutions de touches. Elles sont remplies via un système de fichiers JSON et notamment par des fonctions qui lisent les champs du JSON et instancient en conséquence les listes des différents modules.

Notre pilote de clavier dispose de deux fichiers écrits en JSON : l'un s'appelle "Dictionnaire" et l'autre "Keymap". Chacun de ces fichiers possède des fonctions spécifiques.

Le fichier *Dictionnaire.json* stocke toutes les touches qui seront gérées par notre pilote. On y recense plus de 150 touches, et notre code est conçu pour en ajouter davantage si nécessaire.

Dans ce dictionnaire, chaque touche est représentée par un certain nombre de champs :

- le champ "nom" pour l'appellation de la touche (exemple "Key_A")
- le champ "code" pour son keyCode (qui peut être celui fourni par Java ou alors parfois un que nous inventons, s'il n'est pas déjà pris)
- le champ "sortie" qui correspond à l'affichage produit par la touche

Il permet aussi de définir et de stocker des macros, c'est-à-dire, des phrases ou mots entiers qui pourront donc être rendus accessibles via de simples touches.

Le fichier *Keymap.json* quant-à lui, stocke les différents "Layers" (couches) et la manière dont chaque Layer redistribue les rôles des touches. Dans keymap les touches sont représentées par :

- un champ "physicalKey" et un champ "Key" représentant respectivement la touche physique du clavier et celle par laquelle on la substitue.
- on peut éventuellement avoir : un champ "combo" si elle est concernée par un combo, un champ "ChangedTo" si elle est concernée par un HoldTap.

Keymap est étroitement lié au Dictionnaire puisqu'il reprend ses appellations pour les touches. Remarquez qu'il est important que les appellations et les keyCode soient uniques, pour le déroulement sans ambiguïté des fonctions de recherches.

Vous constaterez que les fichiers JSON sont beaucoup plus agréables à manipuler que les listes présentes dans les modules.

De plus, ils rendent notre clavier modulable : tant qu'il respecte le format de notre dictionnaire et de notre keymap, un utilisateur peut librement remplacer nos fichiers JSON par les siens. Ainsi, il pourra à son gré composer les Layers et les raccourcis claviers qui lui conviennent, en accord avec ses préférences et ses habitudes.

IV. Conclusion : ressentis et problèmes rencontrés :

Pour que notre projet soit totalement abouti, il aurait fallu que nous réussissions à coder la partie "système". En effet, notre projet se déroule exclusivement sur la zone de texte la fenêtre JFrame. Nous aurions voulu apporter certaines modifications dans la classe InterfaceClavier au niveau des entrées et sorties d'événement pour que notre pilote soit exécutable en dehors de cette fenêtre.

Nous avons eu beaucoup de mal à obtenir un résultat concret. Le projet n'attend pas d'animation particulière ou de visuel travaillé, ce qui nous a souvent donné l'impression de stagner, malgré les avancées dans la construction du code.

Ce dernier nécessite une organisation très stricte des divers fichiers et classes, ce qui demande une communication et une coordination accrues entre les tous membres du groupe. Nous avons donc généralement préféré réfléchir, travailler et même coder en équipe plutôt qu'individuellement. D'autant plus que nous manipulions pour la première fois les événements claviers et avons été confrontés à de nombreuses erreurs qui nous étaient jusque là inconnues, et par conséquent, difficiles à résoudre pour nous.

Finalement, nous sommes soulagés et satisfaits d'avoir pu implémenter la plupart des attendus malgré les obstacles rencontrés et les quelques manqués.