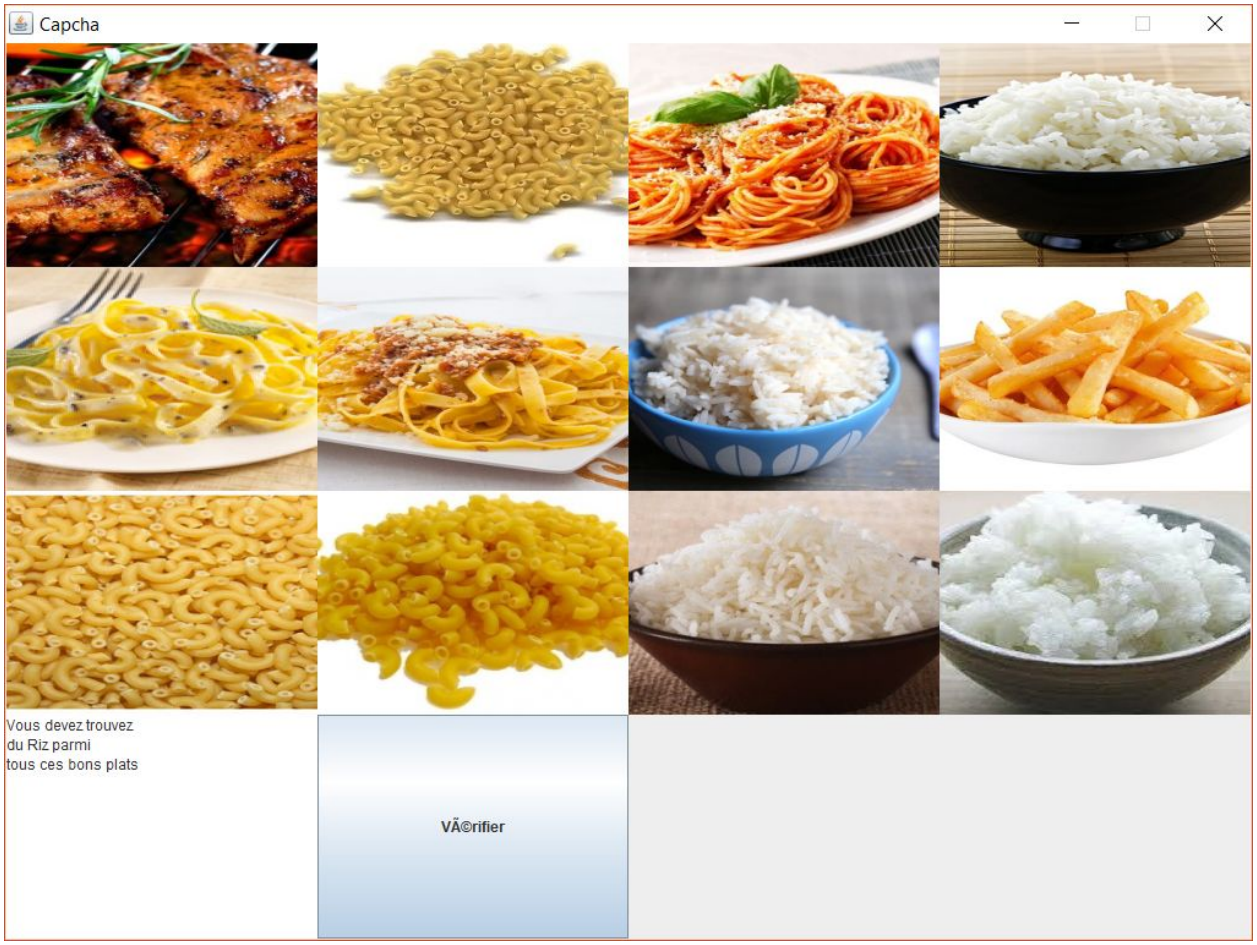
# Projet Java Captchmiam

**IMAC2 - 20 Mai, 2019**

Pour le projet java de deuxième année d'IMAC, nous avons réalisé un captcha par sélection d'images comme on peut en trouver sur internet.



## Présentation du projet

Ce programme permet d'identifier un humain en demandant à l'utilisateur de sélectionner des images représentant certains types de nourriture. Une douzaine d'images sont affichées à l'écran et l'utilisateur doit repérer entre 2 et 5 images d'un certain type parmis elles.

Si l'utilisateur se trompe (sélectionner au moins une image incorrecte ou ne sélectionne pas toutes les images du bon type), un deuxième captcha plus spécifique est chargé. Par exemple le premier niveau peut être d'identifier des pâtes parmi des images de nourriture, le second de trouver des spaghettis parmi des pâtes, le troisième des spaghettis à la tomate parmi des spaghettis…

## Exécuter le programme

Pour exécuter le logiciel, vous devez exécuter le fichier *windows_startup.bat* ou *linux_startup.sh* (selon votre systeme d'eploitation) en cliquant dessus ou en exécutant une des commandes suivantes :

```
./windows_startup.bat
```

Ou

```
./linux_startup.sh
```

Pour plus d'information sur la compilation et l'exécution, voir le fichier README.md.

Malgré les précisions de consigne données le mardi 28 mai, nous n'avons pas pu réaliser un fichier JAR permettant l'exécution du programme, car celui-ci étant incompatible avec d'autres consignes (voir chapitre "Exécution en JAR").

## Packages utilisés

Pour la gestion de l'interface graphique, nous avons utilisé le package swing et ses Jframe, Jbutton et jLabel.

## Exécution en JAR

Comme mentionné sur le forum (discussion “Impossible de charger des ressources avec getResource()” de Valérian DAUL), la création d’un fichier jar fonctionnel est incompatible avec l’introspection, le parcours de fichiers et l’utilisation d’url également imposés par le sujet. Il semblerait effectivement que ce soit impossible de parcourir proprement les fichiers du .jar (avec Files.walk par exemple) pour récupérer les images et les classes enfants automatiquement. La seule solution serait de parcourir le jar comme un zip (avec ouverture et fermeture) et trouver le dossier correspondant, avec un traitement différent selon si le code est exécuté depuis un IDE ou un jar. De plus, le .jar empêche également l’utilisation d’url à l’intérieur de son fichier, elle aussi imposée par le sujet (cf. interface "Images" et chapitre “Architecture”). A la place, il faudrait les stocker en stream avec *getClass().getResourceAsStream(fileName)* et les afficher avec *ImageIO.read(…)*.

Comme pour beaucoup de groupes, l’architecture du programme devrait donc être à revoir pour pouvoir intégrer l'exécution en JAR. En prenant en compte le temps que nous pouvons alloué au projet, le fait que nous n’ayons jamais étudié cette fonctionnalité en cours et en TD et qu’elle va à l’encontre d’autres éléments imposés par le sujet, nous avons pris la liberté de passer outre cette consigne à la suite du manque de réponses ou de précision sur le forum

## Choix de fonctionnement

D’un point de vue utilisateur, l’application se compose d’une interface et d’une collection d’images à identifier. Ces images sont organisées sous forme d’arbre (voir chapitre “Hiérarchie de l’application”) avec des images de plus en plus spécifiques quand on avance le long des branches. L’utilisateur peut faire plusieurs essais et à chaque essai raté, il avance le long d’une branche. Si l’utilisateur atteint le bout d’une branche, le programme se ferme et on lui indique de contacter l’administrateur du service utilisant le captcha. Ce choix a été fait pour éviter des attaques de par flooding qui ralentiraient les serveurs dans le cas d’un service en ligne.

Le nombre d’essais est donc intrinsèquement lié à la profondeur de l’arbre. Dans notre prototype, toutes les branches n’ont pas la même profondeur, ce qui signifie qu’on n’a pas toujours le même nombre de tentatives avant d’être bloqué. Dans une version en production, on prendra soin d’avoir un arbre équilibré.

## Choix d'implantation

Notre système de catégorie se base sur l'introspection et un parcour automatique de fichier. Lorsqu'une classe de catégorie est instanciée, elle va parcourir tous les fichiers de son package/dossier pour y trouver et stocker les URL de ses images (directement dans le dossiers), puis pour y chercher, tester et instancier les classes de catégorie enfant (ses sous-catégories présentes dans les sous-dossiers). Ainsi, pour créer une nouvelle catégorie, il suffit simplement de créer un nouveau dossier dans la catégorie parente, y mettre des images (png ou jpg) et créer une classe qui hérite de la catégorie du dessus, avec un constructeur qui fait appel à *super()* et un override de la fonction *getName*(). Ainsi, il n'y a pas besoin de déclarer les catégories enfant dans le parent, ni ses images.

Ensuite, pour obtenir les images d'une catégorie, la fonction *getPhotos()* renvoie une copie de sa liste d'images directes, qui est jumelée avec la liste des photos de ses enfants à l'aide d'un appel récursif. Ainsi, même si chaque catégorie ne stocke que les photos directement présentes dans son dossier, un appel à la fonction *getPhotos()* permet de retourner toutes les photos de la partie de l'arbre correspondante jusqu'à ses feuilles.

Pour ce qui est de la logique du déroulé de l'application, nous nous sommes basés surtout sur des fonctions appelées "init" et "reset" dans les Classes Logic et MainUI. Elles servent respectivement à initialiser l'application à son lancement et à remettre à zéro certaines variables en prévision du prochain test.

Dans le cas ou on atteint une feuille de l'arbre et qu'il est alors impossible de lancer un test plus compliqué, on fait remonter une exception. Cette exception est traitée dans MainUI et l'application est réinitialisée quand elle surgit.

# Hiérarchie de l'application

On donne ici le détail pour une branche de l'arbre d'images : la branche images > pates

Src > fr > upem > captcha >

- **Logic.java**
- Images (dossier) >
    - **AllCategory.java**
    - **Category.java**
    - **CategoryTools.java**
    - *plusieur fichier images*
    - Patates (dossier, non détaillé)
    - Pates (dossier) >
        - **Pates.java**
        - Coquillettes (dossier) >
            - **Coquillettes.java**
            - *plusieurs fichiers images*
            - Natures (dossier) >
                - **Natures.java**
                - *plusieurs fichiers images*
        - Spaghetti (dossier) >
            - **Spaghetti.java**
            - Cru (dossier) >
                - **Cru.java**
                - *plusieurs fichiers images*
            - Tomate (dossier) >
                - **Tomate.java**
                - *plusieurs fichiers images*
        - Tagliatelles (dossier) >
            - **Taglaitelles.java**
            - *plusieurs fichiers images*
    - Riz (dossier, non détaillé)
- Ui (dossier) >
    - **MainUI.java**

## Répartition des tâches

| | |
|---|---|
| **Nicolas SENECAL** | Classe abstraite Category et interface Images<br>Hiérarchie des images et des classes correspondantes<br>Système de sélection d'images aléatoirement au bon endroit de l'arbre<br>Système introspection et de recherche de sous classe automatique |
| **Olivier MEYER** | Classe Logic et déroulé de l'application<br>Intégration UI (classe MainUI)<br>Possibilité d'enchaîner plusieurs tests<br>Bug tracking |

## Pour aller plus loin

Ce programme pourrait être simplifiée en choisissant de ne pas avoir une classe java par type d'image, mais simplement une unique classe qui s'occupe de parcourir les dossier et de représenter l'arbre, ou une partie de celui ci en fonction des besoins.

Par ailleur, ce projet nous a permis de voir la facilité de coder avec java sur des plateformes différentes, sa versatilité en fait un véritable atout. Malheureusement nous n'avons pas pu générer un exécutable jar qui nous permettais d'explorer les fichiers images simplement et avons donc abandonné cette idée.