Montage & réalisation

2013

Le Dourak

By Okulichev & Munuku

Isims-2IRT-gr1

Table des matières

[INTRODUCTION 4](#_Toc355760003)

[Règle du jeu-Déroulement d’une partie 5](#_Toc355760004)

[*Le but du jeu* 5](#_Toc355760005)

[Les différents mouvements du jeu & subtilités 6](#_Toc355760006)

[*Les différents mouvements* 6](#_Toc355760007)

[*« Le Give »* 6](#_Toc355760008)

[*« Le Cover »* 7](#_Toc355760009)

[*« le Take »* 8](#_Toc355760010)

[*« Le Turnover »* 8](#_Toc355760011)

[Interprétation en langage Java 11](#_Toc355760012)

[*Difficultés rencontrées : Interprétation & codage* 11](#_Toc355760013)

[Comment faire une carte ? 11](#_Toc355760014)

[De la carte au paquet ! 12](#_Toc355760015)

[Du paquet au choix de l’atout! 13](#_Toc355760016)

[L’affichage et la distribution des cartes 14](#_Toc355760017)

[Description de l’Intelligence Artificielle (IA) 15](#_Toc355760018)

[Comment interpréter les mouvements du jeu en langage java ? 19](#_Toc355760019)

[L’affichage 20](#_Toc355760020)

[Difficultés 20](#_Toc355760021)

[Réalisation 20](#_Toc355760022)

[Dimensionnement de l’espace de jeu 20](#_Toc355760023)

[Mise en place du panel du milieu 21](#_Toc355760024)

[Conclusions 24](#_Toc355760025)

[Durak-Listing des erreurs 25](#_Toc355760026)

[SOURCES 26](#_Toc355760027)

# INTRODUCTION

Dans le cadre de notre cours de montage et réalisation (MOR), cette année nous avons eu quartier libre concernant le choix d’un projet à remettre en fin d’année.

Après quelques jours de réflexions nous avons choisi comme sujet un jeu d’origine russe nommé le « Durak ». Le « Durak » signifie « idiot » en russe, c’est un jeu de carte assez dynamique auquel nous jouons régulièrement, voir même très souvent avec un groupe d’ami, que ce soit le matin, le midi et même lors des pauses.

En considérant cela, et notre passion pour la programmation, nous avons donc relevé le challenge de réaliser ce jeu dans le cadre de notre cours de montage et réalisation et par simple curiosité par rapport à la difficulté de conception qu’allait nous apporter le projet.

Nous avions donc pour objectif de :

* Réaliser une interface graphique affichant les cartes et la tables de jeu.
* Etablir les règles du jeu de manière précise, et avec le plus de mouvement possible pour rivaliser avec ce côté dynamique du jeu.
* A la base, nous avions aussi pour but d’implémenter une option « turnover » expliquée plus loin. (laissé en suspens par manque de temps)
* Créer une IA avec des choix tactiques aussi proche des nôtres.
* Mettre en place une option pour une partie multi-joueurs en local (4 IA vs Humain) (abandonné par manque de temps, on s’est focalisé sur un jeu de base à un contre un)
* (optionnel) : mettre à disposition une plateforme multi-joueurs en réseau (abandonné par manque de temps et complexité de base du jeu en lui-même).

# Règle du jeu-Déroulement d’une partie

## Le but du jeu

Le durak, comme tous les jeux de cartes est soumis à des règles. Tout d’abord, il faut savoir que c’est un jeu de trente-six cartes, et que le nombre maximum de joueurs est de quatre, mais il est aussi possible de le jouer à six, et c’est alors avec cinquante-quatre cartes et sans le joker qu’il faudra jouer mais le principe de jeu restera le même.

Le déroulement d’une partie donc commencera avec le mélange des cartes suivit de la distribution de ces dernières. Chaque joueurs possèdera six cartes dans sa main, et le reste des cartes seront posées sur la table face « dos ». Ensuite on prendra soins de retourner soit la première, soit la dernière carte du paquet, et cette carte que l’on retournera sur la table (c’est-à-dire face « image ») représentera la carte dite « d’atout » (à noter que cette carte retournée ne peut pas être un «as », qui est considéré comme la plus haute carte mais il arrive que dans certain jeux cette règle ne soit pas prise en compte), ce sera donc les cartes du même symbole (carreau, pique, trèfle ou cœur) qui constitueront en fonction de leur valeur aussi les cartes les plus haute de la partie. (Voir *fig1)*

*Exemple :*

Exemple d’un début de partie, ou les cartes sont distribuées et ou l’atout (ici carreau) est sélectionné, il se trouve juste en dessous du paquet contenant le reste des cartes.

***Fig.1***

Carte d’atout-couleur d’atout

La partie se termine lorsque qu’un joueur (lors d’une partie un contre un), ou trois joueurs (lors d’une partie à quatre joueur) n’ont plus de carte en main et que le paquet contenant le reste des cartes est vide. C’est donc le joueur qui sera en possession de carte à la fin du jeu qui sera désigné comme étant le perdant ou « durak ».

## **Les différents mouvements du jeu & subtilités**

## Les différents mouvements

Comme vous avez pu le constater avec la *fig1,* le dourak va nous permettre de faire une série de mouvements qui seront tactique les uns comme les autres. Avant tout, au début de chaque partie, un joueur est choisit pour commencer la partie. Ce joueur est désigné en fonction de l’atout ; le joueur possédant la carte d’atout la plus basse (à savoir le six) commencera la partie.

Abordons ici les mouvements de bases essentiel s au déroulement d’une partie ;

***On peut définir quatre catégories de mouvement au durak***

### *« Le Give »*

Il s’agit du simple fait de donner la carte, en trop terme le fait « d’attaquer » l’adversaire suivant en lui donnant une carte que ce dernier devra défendre avec un second mouvement que nous citerons plus loin.

Analysons ce petit bout d’image qui nous illustre cette action de « give » ;



Slavik possède donc le 6 de carreaux, et effectue un « give » sur le joueur suivant

Slavik effectue donc une attaque, que nous devrons couvrir, le fait de couvrir sera appellé « covert »

### *« Le Cover »*

Second mouvement du dourak qui intervient lors d’une attaque, il s’agit d’un mouvement de défense donc, qui a pour simple but de couvrir la carte de l’attaquant par une carte de même symbole et de valeur supérieur **OU**  par une carte d’atout.



PAQUET

Le seul mouvement possible et autorisé au vue de notre main, sera de couvrir avec le valet de cœur.

Et à la suite d’un cover, on repart sur le premier mouvement qui est un give, c’est la que le jeu se corse un peut, à savoir que lors de notre premier cover, la carte que nous avont posée fait partie intégrante de la table.

Petite précision, le joueur qui effectue la première attaque à quartier libre concernant le choix de la première carte qu’il voudra posé (ce choix est la base fort tactique, vu que le but est de se débarasser de toute ces cartes, en général on attaquera en début de partie avec les carte qui ne peuvent pas couvrir donc les « 6 » et donc le joueur qui couvre est soumis à son choix, ce qui justifie bien sa position d’attaque.

Donc, une fois notre carte couverte, l’attaquant à la possibilité de poursuivre son attaque en posant une carte de même valeur qu’une autre carte présente sur la table de jeu. Dans notre exemple, cela donnerai la possibilité à Slavik l’opportunitée de poser soit un 7, soit un valet.

Une fois que Slavik aura terminer son action « Give », le joueur adjacent suivant au joueur qui défends (dans notre exemple il s’agit de Temkaaa) passera lui aussi en mouvement « Give » sur le joueur qui défendait à savoir nous dans notre exemple.

***Il s’agit en réalité d’un système d’attaque à deux contre un !***

Notre mouvement covert prendra fin lorsque six paquets de cartes seront sur la table (donc par groupe de deux) ou lorsque les attaquant ne pourront plus poser de carte par manque de possibilité qu’offre leurs mains respective ou par pu choix tactique. A la fin du tour, chaque joueur repioche des cartes égales au nombre de carte donnée dans le paquet, le but étant d’avoir en permanence six cartes en main.

### *« le Take »*

Mais que se passerait-il si un joueur ne peut pas assurer sa « cover » ? Et bien il ramasse la table ! Donc toutes les cartes déposées sur la table se retrouveront dans sa main! Avec une difficulté en plus qui est qu’une fois que ce joueur qui défend passe en mouvement « take », les joueurs adjacents ont la possibilité de lui ajouter des cartes, en respectant bien sur le fait que la valeur de ces dernières devait être présente sur la table.

### *« Le Turnover »*

Viens maintenant un mouvement purement tactique ! En effet le turnover permet de se sortir de bien difficile situation tout en se mettant soit même en péril

*Exemple :*

*Fig.2*

**

Ici, nous avons la possibilité d’appliquer cette règle du turnover, en mettant notre 9 de trèfle à côté de la carte de l’attaquant

Le principe du Turnover, est de placer lorsqu’on subit un give, une carte de même valeur à côté de la carte présente sur la table et ainsi passer de statut de défenseur à celui d’attaquant. C’est un moyen efficace d’éviter de se défendre lorsqu’on est en possession de carte très petite en se mettant directement en position d’attaque sur le joueur qui nous suit.

*Fig.3 pas de bol c’est un retour à l’envoyeur !*

***On remarque ici que l’adversaire possédait encore un neuf, qui est celui d’atout! Tentons un « mauvais » mouvement, qui est de couvrir les cartes présentes alors qu’on ne peut au final qu’en couvrir deux***

Fig4 : *« le mauvais mouvement »*

***Ici je couvre donc deux carte alors qu’il me sera impossible de couvrir la troisième ! => mouvement qui peut être tactique comme dangeureux…***

*Fig.5 : « les conséquences de ce mauvais mouvement »*

******

***On remarque, qu’à cause de cette action, Le joueur adverse en a profité pour ajouter des cartes de même valeur que les cartes présentes sur la table, et donc nous sommes doublement puni en plus de prendre de « mauvaise » carte , on prends une dame de cœur en plus car nous avions posé une dame sur la table en cover.***

#### Les subtilités

Comme vous avez pu le constater, il existe dès lors de nombreux choix quand au mouvement qui sont effectués. Ces choix dépendront soit de la situation dans laquelle vous vous trouvez, soit de où en est le jeu.

A savoir que si vous avez beaucoup d’atout en début de partie, il vaudrait mieux pour vous de prendre si vous devez les jeter, car vous n’êtes pas sur de piocher des cartes d’atout à la fin du tour. Alors que d’une autre part, le fait de constamment prendre vous mettra en situation fort inconfortable ! C’est donc à vous de déterminer quel est la solution la mieux adaptée, et c’est la qu’apparaît la grande difficulté et la complexité du jeu tant sur le plan jeu en lui-même, que sur le côté programmation.

# Interprétation en langage Java

## Difficultés rencontrées : Interprétation & codage

Après avoir déterminé l’ensemble des règles, intervenait dès lors la partie interprétation et codage sous java.

Pour cela, nous avions fixé un canevas de base que nous avons suivit tout au long de la conception de notre programme. Mais nous avons aussi constaté que l’implémentation de certaine fonctionnalités du jeu ne serait vraiment pas tâche facile.

Nous nous somme d’abord heurté avant toute *chose à « Comment faire une carte ? »*

* *Comment déterminer notre atout ?*
* *Et notre IA ? Comment devra-t-elle réagir avec le reste ?*
* *Comment interpréter les mouvements du jeu en langage java ?*
* *Comment afficher notre jeu ? Quels sont les éléments indispensables lors de cet affichage ?*

### Comment faire une carte ?

Tout d’abord de quoi est composée une carte ?

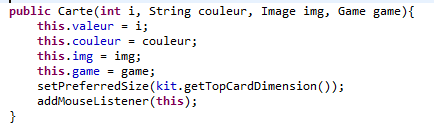
CARTE

Couleur

Valeur

Une carte comporte en en réalité deux éléments, la valeur et la couleur. Partant de cette base il était facile de créer l’objet carte.

*Fig.6*

**

*On remarque qu’effectivement le constructeur de la classe carte, prends bien en paramètre, une valeur et un String couleur. A noter qu’ici on insère déjà l’élément image, qui va nous aider à faire correspondre la carte (valeur, couleur) à son image. Le paramètre game ici c’est appelé pour avoir l’accès aux méthodes de la classe game.*

### De la carte au paquet !

Et oui, cela coulait de source, après avoir crée notre carte, il fallait passer par l’étape paquet qui était la difficulté suivante. Reprenons, une carte est donc composée d’une couleur et d’une valeur. Le paquet lui est composé de carte ; donc un ensemble de couleur et de valeur !

Mais comment créer et représenté ***un élément qui contient plusieurs éléments d’un autre ?***

En java, nous avons trouvé une classe qui nous permettant de regrouper plusieurs élément en un seul. Il s’agit des Array-List, le principe était simple ; chaque carte représentera un élément de cette Array-List nommé « paquet », c’est-à-dire que chaque « case » de notre Array-List, sera une carte composée d’une couleur, et d’une valeur !

*Fig.7*

*2013-05-08_020155.png La figure ci-contre nous montre que le paquet est en réalité une extension d’une Array-List qui contient des cartes.*

***Une Array-List c’est très bien oui ! Mais…Comment je crée mon paquet vraiment ?***

Et bien, la difficulté dans ce cas, était simplement de trouver un moyen de rassembler ces deux informations (valeur, couleur) ensemble alors qu’elles sont de base de type différents, (valeur de type « int », couleur de type « String ») et c’est ce qui fait la force des Array-Lists, due au fait qu’elles peuvent contenir des éléments de type différents, cela fut facile pour nous de créer un ensemble de carte (donc un paquet), à deux éléments (valeur, couleur) .

Ainsi donc l’Array-list se composait comme tel ;

monPaquet <Carte>

Etc.

D

9

7

R

V

Viens alors le remplissage du paquet proprement dit ;

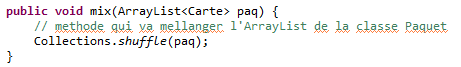
*Fig.8*

**

Le principe est le suivant, on détermine un tableau de couleur, tableau que l’on va parcourir pour ajouter et associer à chaque couleur un numéro, les deux éléments ensemble forment notre carte, et les éléments ensembles formeront notre paquet.

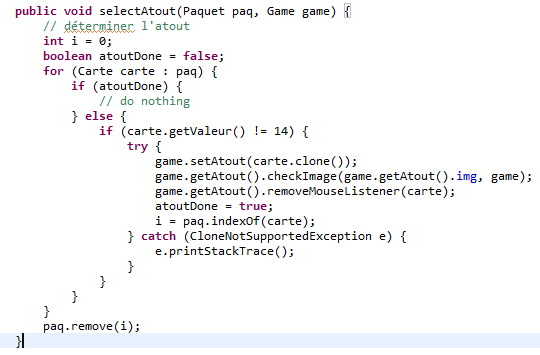
Ensuite afin de rendre le plus aléatoire possible le paquet, nous avons utilisé la méthode « arrayList.shuffle » qui s’applique sur les Array-list, et qui a pour rôle de mélanger ces dernières.

*Fig.9*

**

### Du paquet au choix de l’atout!

Une fois notre carte et notre paquet enfin fini, il nous a fallut déterminer l’atout. Au vu des difficultés précédentes, le choix d’une carte (ou plus précisément ici d’une couleur) parmi notre paquet c’est ne c’est pas révéler compliqué.

Fig.10

On définit donc un variable de type « boolean » que l’on définit à false de base, elle va représenter le fait d’avoir trouvé un atout ou pas. Et on se sert de cette astuce d’utilisation de variable pour pouvoir parcourir notre Array-List paquet et y cloner la première carte (différente de l’as) qu’il voudra, et ensuite prévenir le programme que son exécution touche à sa fin car l’atout à été défini.

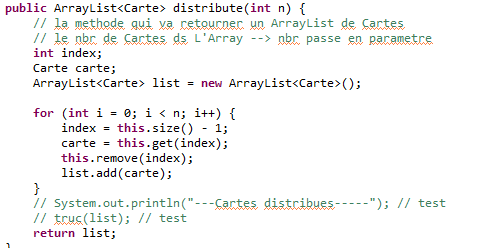
### L’affichage et la distribution des cartes

#### Distribution des cartes

Nous avons trouvé comment créer une carte, pour ensuite se tourner vers le paquet et terminer par la sélection de notre atout.

Parlons dès à présent de la distribution des cartes à chaque joueur, il s’agit la de la dernière étape concernant la manipulation de l’objet carte en lui-même dans notre projet.

*Fig.11*

**

Comme le montre la figure (*fig.11*) ci-dessus, la distribution des cartes est encore une opération, une application des méthodes de la classe Array-List, étant donné que nos éléments carte et paquet en héritent, cela va de soit que ces méthodes puissent être utilisée par nos deux classes crée.

Il s’agit donc d’utiliser les méthodes :

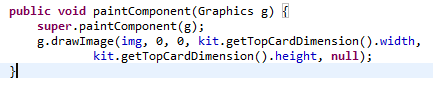
* List.Remove (int élément) ; qui aura pour but d’enlever l’élément à la liste
* List.add (carte carte) qui aura pour but d’ajouter une array-liste à une autre

Avec ces deux méthodes et une boucle for, on parvient donc à distribuer équitablement chaque carte à chaque joueur.

#### L’affichage des cartes

Pour afficher une carte, on utilisera les méthodes liée aux interfaces graphiques et au panel (que nous expliquerons plus loin)

*Fig.12*

*On spécifie bien les paramètres de dimension liés à la carte avant de l’afficher.*

# Description de l’Intelligence Artificielle (IA)

#### Et notre IA ? Comment devra-t-elle réagir avec le reste ?

L’IA est conçue de sorte qu’elle regarde les cartes qu’elle a dans sa main (c’est-à-dire dans l’ArrayList que possède la classe). Avant de commencer à jouer, à chaque tour, l’IA range (listing 🡪*fig.12*) les cartes par valeurs dans l’ordre croissant des cartes. Puis sépare le paquet (la main) en deux sous-paquets. Le premier contiendra toutes les cartes non-atouts. Le second rien que des atouts.

##### ***Défense***

Comme les deux paquets sont rangés au préalable, l’IA parcourt le premier paquet en cherchant une carte de couleur similaire et de valeur supérieure à celle qu’elle doit défendre (à savoir on défend carte par carte jusqu’au moment où on ne sait plus se défendre et on est obligé de prendre toutes les cartes posées sur la table).

S’il n’y a pas de carte supérieure de même couleur que la carte à défendre, l’IA joue la première carte du paquet d’atout. Donc l’atout le plus petit, car les cartes sont rangées.

S’il n’y a pas de carte d’atout pour se défendre, l’IA envoie un signal au gestionnaire du processus du jeu que l’IA ne sait plus se défendre et donc elle prend les cartes.

##### ***Attaque***

De nouveau, comme les cartes sont rangées l’IA joue la plus petite carte de son paquet de carte non-atout. Ou alors s’il n’y a pas de cartes non-atouts, c’est la carte d’atout la plus petite qui sera posé sur la table.

##### ***Difficultés***

La difficulté de concevoir une IA performante se trouve non pas seulement dans la dynamique du jeu, mais dans l'interprétation et le codage des différents choix tactique qu'un véritable joueur aurait fait.

Premier cas, un vrai joueur pourra tenir compte des cartes qui ont été déjà jouées et donc sorties. En fonction de ça, il peut deviner la probabilité des cartes qui pourront être posées sur la table pour nous attaquer.

*Exemple concret : Deux cartes de valeur « 8 » sont sorties du jeu. On se fait attaquer avec un « 6 de pique ». Dans la main on a, admettons un «7 de pique » et « 8 de pique ».*

Dans la logique des choses on devrait se défendre avec le sept qui est la plus petite carte qui peut couvrir le six. Mais en tenant compte que tous les autres sept sont encore dans le jeu et qu’il est probable que l’adversaire ait au moins un sept. On se fera attaquer de nouveau. Dans ce cas on préférera jouer le huit, car il y a moins de chances que l’adversaire ait un huit. Ou encore si c’est un huit d’atout il préfèrera le garder pour pouvoir se défendre en cas de besoin.

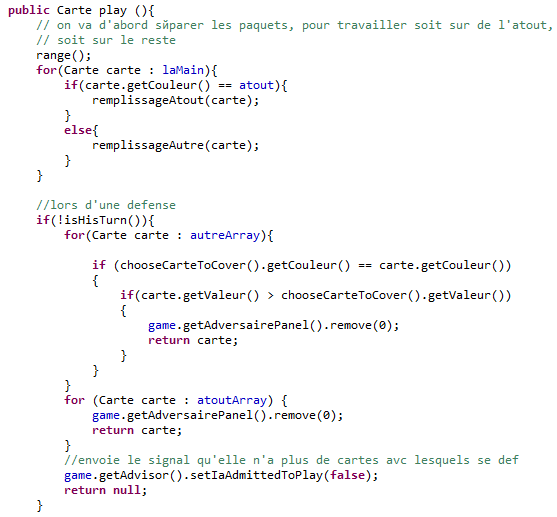
Deuxième cas, si on fait prendre notre adversaire. Du coup on connait une partie des cartes que l’adversaire a en main. Et on est susceptible de jouer non les plus petites cartes qu’on possède, mais adapter notre jeu pour en tirer le meilleur profit. Soit faire jouer l’adversaire les grandes cartes qu’il vient de prendre pour le mettre en difficulté lors des prochaines attaques. Soit attaquer de sorte que les cartes prises vont être jouées, ce que nous donnera l’occasion d’attaquer.

*Exemple concret : on vient de faire prendre l’adversaire. Admettons, il y avait deux valets parmi les autres cartes qu’a pris l’adversaire*.

On pioche les cartes et on pioche un « 10 » et un valet. On privilégiera le 10 parmi les autres cartes à jouer, car l’adversaire est susceptible de jouer un valet (vue que c’est la plus petite carte qui couvre le dix, évidement il faut qu’il soit tout les deux de même couleur) et à ce moment-là on pourra jouer notre valet.

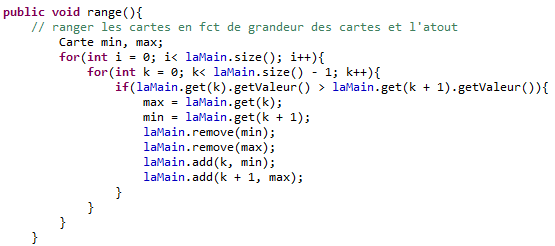
Cependant, rien ne dit que l’adversaire va jouer le valet. Il pourra aussi bien jouer une carte rien avoir plus haute, comme un roi ou encore une carte d’atout. Ainsi que rien ne dis qu’il n’y aura pas d’autres possibilités de jouer le 10.

Ces deux exemples nous montrent bien, qu’il sera difficile voire impossible d’établir un total listing des cas possibles, car chaque cas étant bien unique dans la plupart des situations. Ce qui nous a poussés à simplement nous focaliser sur les cas généraux.





*Fig.13* : listing IA.

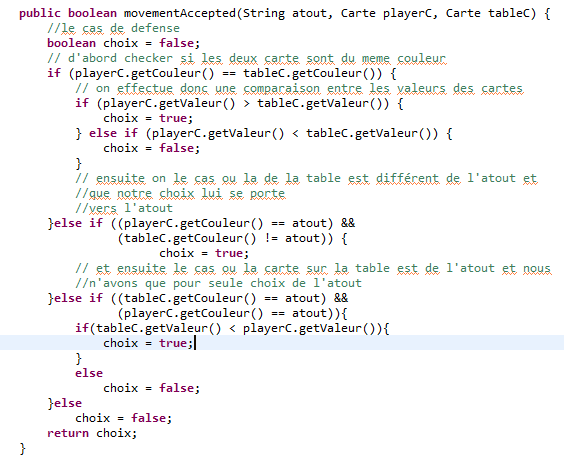


*Fig.14*: listing méthode range().

# Comment interpréter les mouvements du jeu en langage java ?

La réponse ne fût pas simple en réalité, après avoir décortiqué ce problème nous avons trouvé qu’il s’agissait « simplement » de conditions à vérifier ;

*Fig.15*

******

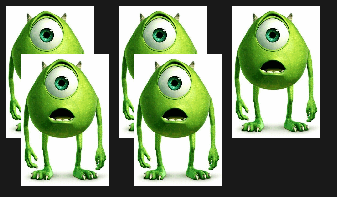
Comme l’illustre la figure 5, notre méthode qui aura pour rôle de vérifier les mouvements va se nommer « movementAccepted », elle prendra en paramètre un atout de type String (qui représente en réalité la couleur de l’atout et non sa valeur), ensuite un objet de type carte qui représentera l’objet carte comme définit précédemment ; à savoir ici que l’objet carte a été séparer entre la carte présente sur la table, qui est la carte que l’on doit défendre, et la carte que l’on focus (via le mouseclick).

# L’affichage

Afin d’avoir une vue d’ensemble du jeu correcte il nous a nous pencher sur les technologies du package « javax.Swing ». Ce dernier n’est pas aussi simple à utiliser qu’il peut paraitre à première vue. Surtout quand on sort de l’affichage ordinaire lié au placement des éléments basiques, tels que des labels, boutons, checkbox, etc. Alors au début on a mis longtemps avant de décider quelle technologie utiliser, soit classique Swing ou la nouvelle Java FX. Plusieurs ouvrages ont été consultés et le choix tomba sur Swing pour une seule et simple raison qui est que cela fait partie du cursus de l’établissement scolaire fréquenté. Ce qui par conséquence nous a permis de réduire le travail lié au cours de Java.

## Difficultés

La première difficulté pour la réalisation de l’affichage, c’est que selon notre idée, les cartes jouées doivent apparaitre une au-dessus de l’autre et avec un certain décalage. De sorte :

*Ne tenez pas compte de l’image, ce n’est pas une carte. Ce n’est qu’un exemple, parce que nous n’avons pas encore réussi à implémenter le bon fonctionnement de l’IA qui permettra de placer correctement la carte pour pouvoir montrer avec les vraies images du jeu.*

*Fig.16*

La deuxième difficulté, était le fait de ne pas se contenter d’une taille de fenêtre fixe. On a décidé de faire en sorte que la fenêtre du jeu, ainsi que la taille des composants de l’interface graphique du jeu s’adaptent automatiquement à la résolution de l’écran sur lequel le jeu sera lancé. D’autant plus qu’on utilise les Layouts.

Finalement, la troisième difficulté était la gestion de l’affichage des cartes en fonction de la phase du jeu. Car si c’est nous qui attaquons la carte joué devrait se mettre en dessous et la carte qui vient couvrir vient se placer au-dessus. Une fois la phase changée, le placement des éléments est inversé. Cette étape est réalisée partiellement et nécessite encore du travail et d’optimisation. Donc explication de cette partie sera omise.

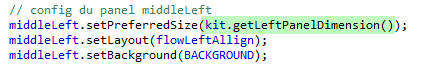
## Réalisation

### Dimensionnement de l’espace de jeu

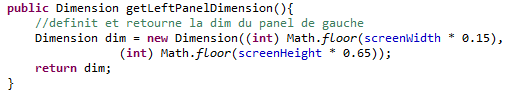
Pour commencer on a créé une classe « SizeKit » qui s’occupe de redimensionner la fenêtre et les panels pour que ces derniers aient une taille adéquate et bien adapté. Les valeurs de départ que la classe prend en paramètre sont ceux de la résolution de l’écran. Que l’on obtient à l’aide de classe « Toolkit » de sorte :

C:\Users\Okulich\Desktop\Rapport_MOR\img\toolkit.png

Ensuite, une fois que la taille de la fenêtre (à savoir la « javax.JFrame ») est établie, on a placé une « JPanel » principal selon les règles de l’utilisation de Swing auquel on a affecté un « BorderLayout ». Puis sur ce « JPanel », on vient placer les autres « JPanel’s » (déjà dimensionnées par « SizeKit ») qui vont être les conteneurs de leur propre composants. Le dimensionnement de ces derniers « JPanels »  se passe de telle manière que l’on passe en paramètre la méthode de dimensionnement de l’élément concerné l’objet « Dimension » qui est générée par une des méthodes appropriées de « SizeKit ».



*Fig.17:* on passe en paramètre de la méthode de dimensionnement

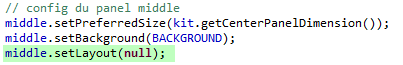


*Fig.18:* l’objet « Dimension » qui est génère par une des méthodes appropriées de « SizeKit ».

### C:\Users\Okulich\Desktop\Rapport_MOR\img\111.pngMise en place du panel du milieu

Pour avoir cet effet là il nous a fallu premièrement bien comprendre le fonctionnement des agents de placement automatique appelés les « Layouts ». Lesquels, nous ont forcés à combiner avec le positionnement absolue, ainsi que jouer sur l’opacité des JPanels sur lesquels viennent se poser les images de cartes.

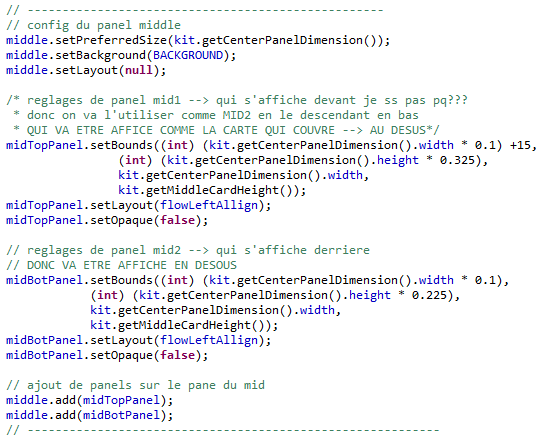
Tout premièrement on a placé un « JPanel » au milieu et on lui a retiré le « Layout » par défaut, à savoir « FlowLayout ». Afin de pouvoir placer librement les panels du milieu (qui contiendront les cartes) aux endroits souhaités. De sorte que :



Puis on place le  « JPanel’s » du milieu grâce à la méthode native des  « JPanel’s » - « setBounds()». Ainsi qu’on les rend transparents afin de pouvoir voir les images à travers pour avoir l’effet souhaitée. Et puis on ajoute ces « JPanel’s » à notre « JPanel » du milieu.

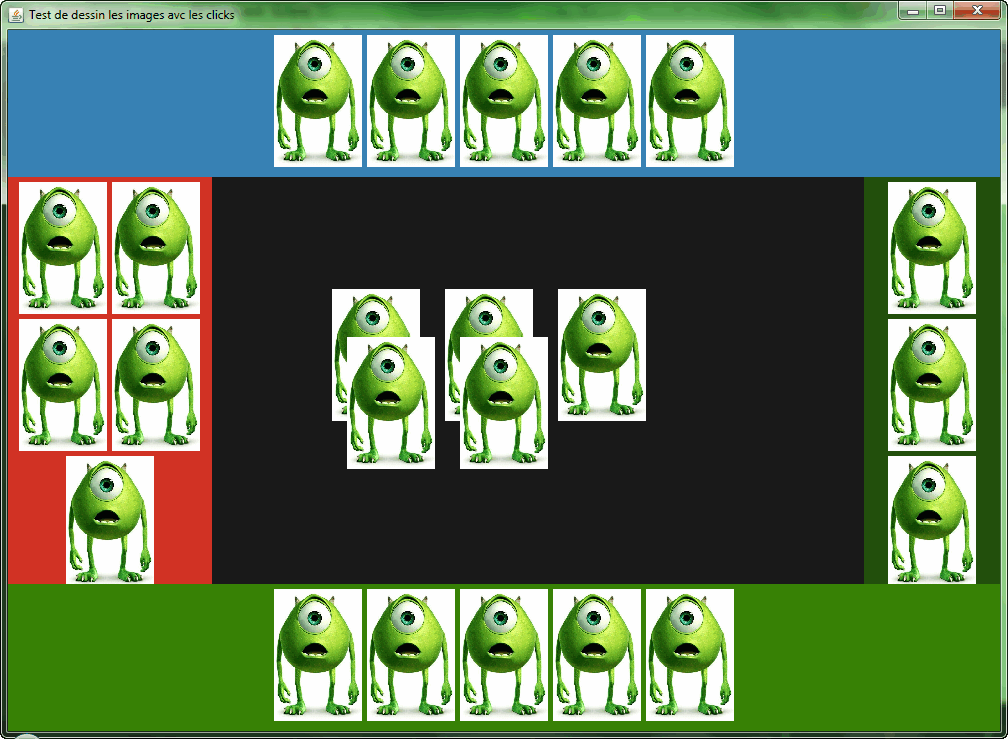
*Voir Fig.19*

***Remarque :*** *Dans le listing on peut voir que le midTopPanel est ajouté en premier et le midBotPanel est ajouté après. Donc dans la logique des choses, le premier ajouté doit être placé en dessous du deuxième vue qu’ils sont tous les deux en positions absolues. Mais pour des raisons qui nous échappent cela se fait de manière inverse. Donc pourquoi les noms sont adoptés pour la simple raison que cela nous évite de réfléchir à chaque fois à l’inversion des noms des panels parce que l’affichage ne se fait pas correctement.*

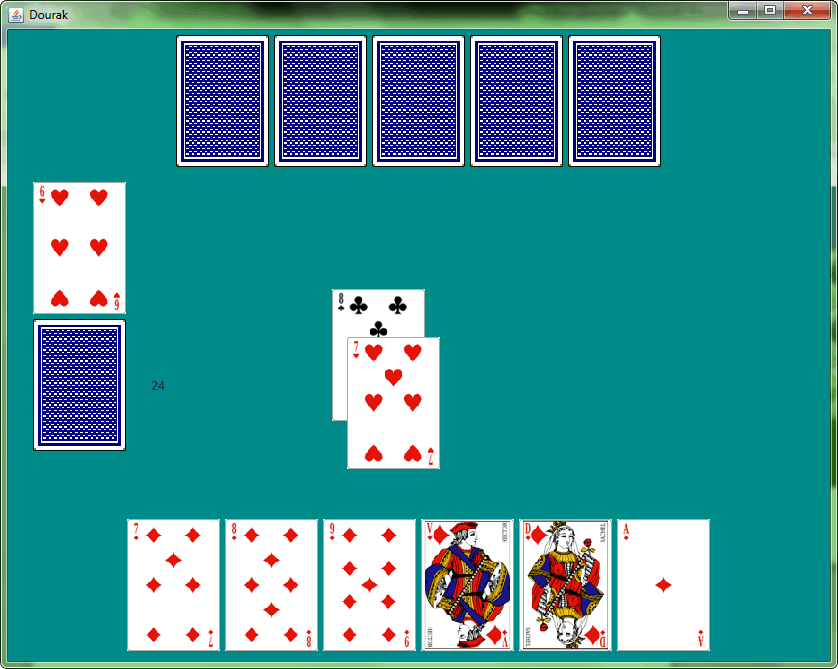


*Fig.19*

Avant d’entamer l’implémentation dans le projet même on a fait un test à côte avec un « JFrame » et « JPanels » tout comme on allait faire dans le projet. Chaque « JPanel » a une couleur différente pour voir les zones qu’il occupe. Voilà l’aperçu :



Et voilà ce que ça donne dans le jeu :



# Conclusions

Tout d’abord, ce qui nous a passionné dans ce projet était l’approche que nous avions au départ ; à savoir qu’on se lançait vraiment vers un jeu qui nous semblait facile à force d’y jouer mais qui n’en n’était pas moins complexe à force d’y réfléchir.

Une fois arrivé devant les murs et les obstacles de difficultés, nous avons pris du plaisir à décortiquer minutieusement et patiemment chaque cas, afin d’en faire ressortir le code ou la méthode ou simplement le raisonnement adéquat qui nous remettrait sur de bon rail.

Malgré le fait qu’à l’heure ou ce rapport est écris le jeu n’est encore qu’en phase béta test, il avance et nous touchons presqu’au terme de nos objectifs, que ce soit objectif personnel, comme objectif dans le cadre du cours de montage et réalisation.

A savoir, que nous avons aussi dressé un listing des erreurs de notre logiciel afin d’avoir une phase débogage des plus parfaites qu’ils soit en ne laissant absolument aucune erreur infime soit elle dans nos lignes de code.

Pour ce qui à déjà été réalisé, à savoir :

* L’interface graphique et l’affichage des cartes
* La méthode s’occupant des mouvements possibles
* Nos classes gérant la création de carte et de paquet de carte
* Etc…

Cela à vraiment été un véritable challenge que nous avons eu le plaisir de révéler, et nous espérons qu’a travers ces quelques lignes vous avez pu apprécier l’étendue de notre travail bien que le résultat final ne conviennent pas encore à nos attentes soyez-en sur que nous ne lâcherons pas ce projet tant que l’objectif ne sera pas pleinement atteint.

En annexe une feuille contenant un listing des erreurs ***connues*** à ce jour.

# Durak-Listing des erreurs

* **Au niveau de l’Ia :**
* L’IA joue le premier coup seulement, et pas les suivants
* Régler le problème du « return null » au niveau de la carte que l’IA renvoie
* Problème d’affichage au niveau de son panel (sur certaine partie il manque 1 ou 2 cartes au panel de l’IA) + sur certaine partie les cartes ne sont pas directement enlevées de son affichage.
* Problème de panel qui s’empile sur la carte qu’elle doit jouée.
* L’IA de parvient pas non plus à couvrir ne fusse qu’une seule carte quand elle est en mode « je défends »
* **Au niveau du joueur :**
* Le joueur joue alors que l’IA n’a pas encore posée de seconde carte
* Le joueur couvre une carte qui a déjà été couverte (du au problème ci-dessus, mais ça peut poser problème pour la suite)
* **Au niveau du jeu en lui-même :**
* (Secondaire) : toujours ce même problème des règles à faire glisser qu’il faut check
* Vérifier les panels « table de jeu » car l’empilement des cartes sur les panels est louche
* Vérifier que les panels tables sont bien limités à la taille de l’array-list correspondant

Reste du débug à venir en fonction des erreurs rencontrées sur les corrections des erreurs présentes…

# SOURCES

Fig1 :<http://wp.appadvice.com/wp-content/uploads/2010/02/durak.png>

Fig.2 et le reste: photo prise à partie d’une partie en cours sur un Ipad & screen personnel