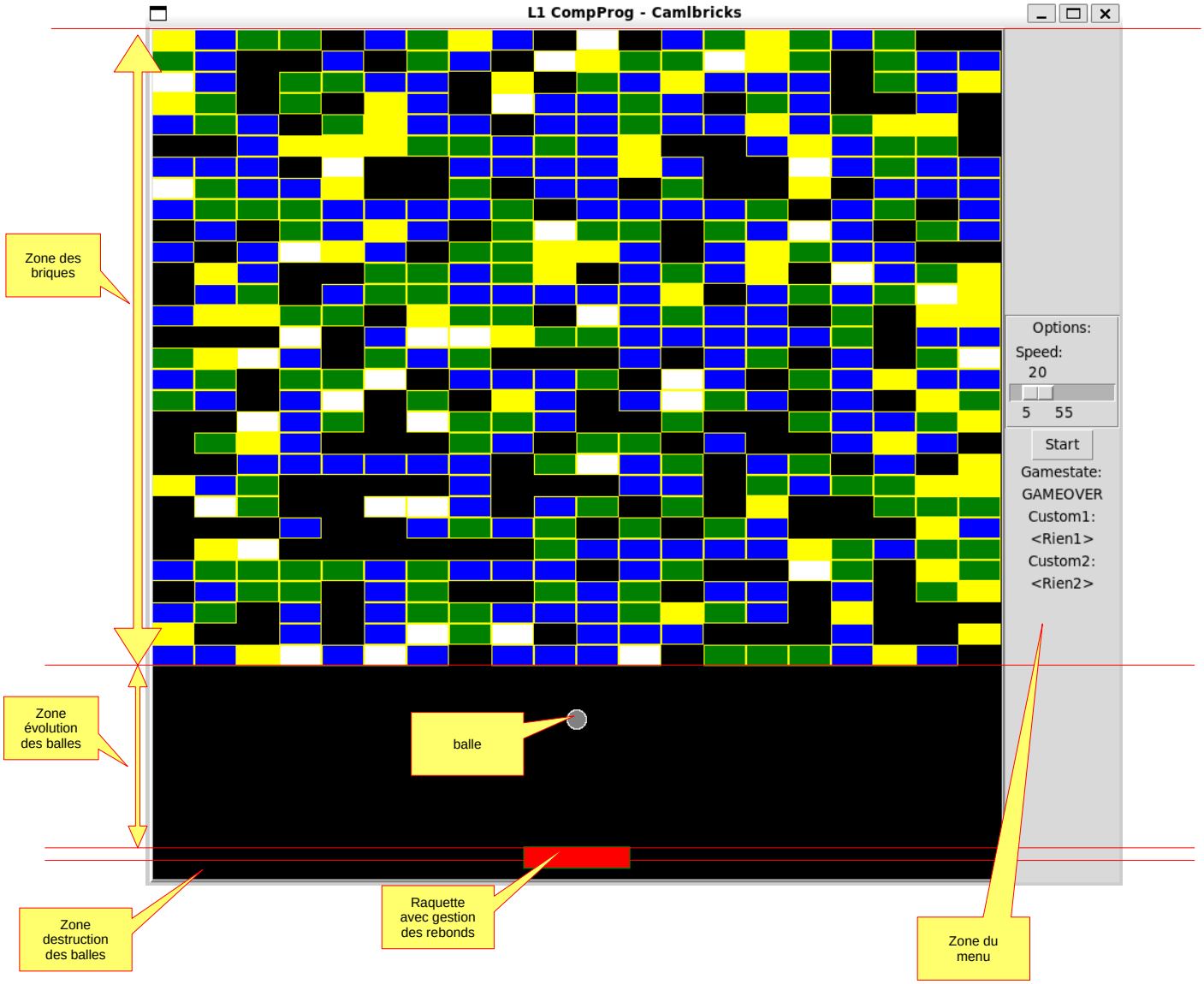
Complément de Programmation
Projet – *Camlbrick*

1 Présentation du jeu

Le jeu du casse-brique donne la possibilité de contrôle, à un humain, d'une raquette qui ne peut que se déplacer horizontalement en bas de l'écran. Cette raquette permet de faire rebondir une balle qui se déplace dans le niveau. Si une balle touche une brique alors cette dernière est détruite. L'objectif est donc de détruire toutes les briques destructibles du niveau.

La difficulté du jeu consiste à rattraper la balle avec la raquette pour la guider vers les briques restantes du niveau. Une balle peut rebondir sur les bords latéraux et le bord haut de la carte. Elle rebondit aussi sur les briques. Cependant, si une balle dépasse la zone de rebond de la raquette alors la balle est détruite. Le jeu est perdu lorsqu'il n'y a plus de balle à l'écran.



La figure ci-dessus montre les différentes parties de l'interface graphique du jeu :
— la zone de menu contient des boutons et des informations qui pourront influencer le jeu. Elle se situe sur la droite de l'affichage (avec un fond gris).
— le monde du jeu est la zone affichant l'intégralité du jeu (c'est la zone entière à gauche avec un fond noir). Dans la suite, le monde est la zone où le jeu peut dessiner (il s'agit d'un canevas).
— la zone des briques est la zone où les briques seront affichées. La largeur de la zone des briques est égale à la largeur du monde.

— la zone d'évolution des balles qui est une zone où ni la raquette, ni une brique ne peut y pénétrer. C'est une zone où les balles peuvent évoluer et il faut qu'elle soit suffisamment haute pour qu'un humain puisse anticiper les réactions et déplacer la raquette en conséquence.
— la raquette et sa zone de rebond est la partie dans laquelle si une balle arrive en descendant alors la balle est renvoyée vers le haut.
— la zone de destruction est la limite où si une balle l'atteint alors nous devons la détruire. Attention, cette zone doit être dans une zone visible et ne dois pas correspondre à la bordure inférieure afin que l'humain puisse constater que la balle est bien détruite.

Les informations ci-dessous sont encodées dans le type structure `t_camlbrick_param` qui encode les paramètres du monde. Vous trouverez dans les commentaires de documentation les contraintes qui lient les différentes valeurs entre elles (par exemple, la largeur du monde doit être un multiple de la largeur d'une brique). Les valeurs que nous avons utilisé dans les démonstrations faites en cours et sur la capture d'écran précédente est la suivante :

```
world_width = 800;
world_bricks_height = 600;
world_empty_height = 200;

brick_width = 40;
brick_height = 20;

paddle_init_width = 100;
paddle_init_height = 20;

time_speed = ref 20;
```

Vous pouvez utiliser les mêmes ou alors mettre vos propres valeurs en accord avec les contraintes mentionnées dans la documentation.

2 Présentation des fichiers fournis et généralités sur vos réalisations

Le projet que nous vous proposons contient des trous que vous devrez remplacer avec du vrai code. Le projet que vous avez initialement s'interprète et compile très bien. Cependant, il ne se produit rien hormis l'affichage de la fenêtre du jeu.

Le projet que nous vous proposons contient les fichiers suivants :
— `camlbrick.ml` est le fichier qui contient l'ensemble des fonctions/types que vous utiliserez et que vous devrez définir/redéfinir. C'est ce fichier que vous modifierez principalement. Lisez-bien les commentaires des fonctions qui vous donnerons des indications supplémentaires. L'étiquette `@deprecated` est souvent mentionné pour les fonctions qui ne doivent être pas modifiées et nous avons ajouté un commentaire des différentes itérations où la fonction/le type devront être réalisés.
— `camlbrick_gui.ml` s'occupe de toute la partie affichage et interface graphique. Il ne vous est pas demandé de faire des modifications ou de l'ouvrir. Cependant, vous devez savoir que ce fichier a besoin de `camlbrick.ml` qui soit fonctionnel pour qu'il puisse fonctionner correctement.
— `camlbrick_launcher.ml` gère la préparation d'un niveau. Il s'occupe d'appeler les fonctions des fichiers précédents pour démarrer une partie. En particulier, c'est ici que vous chargerez votre niveau (que vous devrez faire) avant de lancer les fonctions qui démarre le moteur du jeu.

L'objectif est de travailler de manière collaborative sous la forme d'une équipe de 4 étudiants. L'équipe peut-être indépendante du numéro de TD à condition que vous soyez dans les mêmes créneaux horaires (c'est-à-dire si vos TP/TD sont aux mêmes heures).

Lorsque vous modifiez une fonction vous devrez toujours faire un commentaire de documentation et mentionner au moins l'étiquette `@author` du développeur (et non pas l'ensemble de l'équipe). Vous devrez aussi faire un ou plusieurs tests de la fonction et/ou d'une fonction auxiliaire selon les situations. Le commentaire de votre fonction de test doit mentionner l'objectif de test explicitement et l'étiquette auteur doit permettre d'identifier le testeur qui a écris la fonction de test. Il vous est donc enfin demandé de faire des tests fonctionnels et des tests structurels.

Enfin, nous vous demandons d'ajouter les noms des membres de l'équipe dans la liste des auteurs dans le premier commentaire du fichier `camlbrick.ml`.

Le projet est découpé en 4 itérations :
— Itération 1 : se focalise sur la compréhension du projet, la conception des premiers types et les premières fonctions.

— Itération 2 : s'intéresse à la gestion complète de la raquette dont ses déplacements et le début des balles.
— Itération 3 : termine la gestion des balles, en particulier la gestion des collisions balles/briques et balles/raquette. Lie toutes les mécaniques d'interactions pour que le jeu puisse être jouable.
— Itération 4 : réalisation des extensions de manière assez libre.

Pour chaque itération vous aurez une soumission de l'état d'avancement de votre projet. Vous devrez en l'occurrence fournir vos fichiers sources sur Updago et réalisez une campagne de test pour chaque itération (le fichier devra s'appeler `tests_iterationX.ml` où X est le numéro de l'itération). Vos enseignants pourront exiger des fichiers complémentaires selon la situation.

3 Itération 1 : réalisation des premières fonctions outils et des briques

3.1 Préparation des vecteurs

Pour démarrer ce projet, nous vous proposons de commencer avec la notion de vecteur. Cette notion essentielle dans les jeux vidéos et que vous avez appris en mathématique, va permettre de représenter des coordonnées ou des vitesses pour les différentes entités de notre jeu.

Nous vous demandons de compléter le type `t_vec2` à votre convenance pour représenter la notion de vecteur 2D (c'est-à-dire ayant une composante en x et une autre en y). Vous réaliserez aussi les fonctions utilitaires pour manipuler ce type à savoir les signatures suivantes en ajoutant bien un commentaire de documentation à chaque fonction et des tests adéquats (comme dit précédemment il faut le faire à chaque fois).

```
(* Construit un vecteur à partir de deux entiers *)
let make_vec2(x,y : int * int) : t_vec2 =

(* calcul la somme de deux vecteurs*)
let vec2_add(a,b : t_vec2 * t_vec2) : t_vec2 =

(* calcule la multiplication des composantes de deux vecteurs entre eux*)
let vec2_mult(a,b : t_vec2 * t_vec2) : t_vec2 =
```

Vous pourrez réaliser d'autres fonctions si vos extensions ou votre code le nécessite. Enfin vous devez réaliser toutes les fonctions liées au paramétrage et en particulier enregistrer les paramètres dans le type `t_camlbrick` afin que l'interface graphique puisse extraire les paramètres depuis une partie de casse-brique (`param_get`). Enfin, pour vous aider dans vos tests vous pouvez aussi réaliser les fonctions `make_camlbrick`. Vous pouvez vous aider des commentaires pour identifier les fonctions avec le commentaire `Itération 1`.

3.2 Préparation des briques

Les briques sont les éléments passifs du jeu et sont représentés par des couleurs selon leur type. Habituellement, la zone des briques est une grille 2D où chaque case va indiquer le type de la brique présente. Par défaut, l'absence de brique est identifiée avec une valeur particulière et nous dirons que la brique est vide (`BK_empty`). Lorsqu'une brique est touchée par une balle nous avons les effets suivants :

— une brique simple (`BK_simple`) disparait.
— une brique double (`BK_double`) devient une brique simple.
— une brique bonus (`BK_bonus`) disparait et une action devra être lancée (vous pouvez laisser l'action pour l'itération 4).
— un bloc (`BK_block`) ne peut pas être détruit.

Concrètement, vous devez déterminer la structure de donnée permettant de représenter toutes les briques de la zone des briques. Vous devez modifier le type `t_camlbrick` pour qu'il puisse stocker ces briques et réalisez les fonctions :

— `brick_get` qui renvoie le type de brique à partir des coordonnées dans la zone de briques.
— `brick_hit` qui réalise les modifications dans la zone de brique pour faire évoluer une brique comme si elle était touchée par une balle.
— `brick_color` qui renvoie une couleur en fonction du type de brique.

Vous pouvez aussi modifier le fichier `camlbrick_launcher.ml` pour faire des essais de monde pour vérifier que l'interface graphique comprend votre code.

4 Itération 2 : Gestion de la raquette et préparation des balles

A la fin de cette itération vous devez avoir l'affichage des briques sur un niveau de votre choix, l'affichage d'une balle inerte et enfin l'affichage et la possibilité de déplacer la raquette da manière cohérente vis-à-vis du bord du niveau.

4.1 Gestion de la raquette

Pour préparer la raquette vous devez d'abord réfléchir au type d'une raquette afin que nous puissions extraire les informations de taille, de position de la raquette. Pour cela vous devez compléter le type `t_paddle` et modifier votre type `t_camlbrick` pour qu'une instance d'un jeu puisse avoir les informations de sa raquette.

Dans un premier temps et pour nous assurer vous pouvez créer l'opération `make_paddle` qui vous sera utile lors de vos tests logiciels et vérifier que votre structuration.

A présent, réalisez les différents accesseurs à partir d'une raquette contenu dans un `t_camlbrick`, pour obtenir les informations nécessaires à l'affichage par l'interface graphique :
— `paddle_x` renvoie la position gauche du rectangle symbolisant la raquette.
— `paddle_size_pixel` renvoie la largeur en pixel du rectangle.
Nous allons maintenant nous occuper de la partie interaction de la raquette. Pour cela vous devez écrire le code de `paddle_move_left` (resp. `paddle_move_right`) qui permet de déplacer la raquette vers la gauche (resp. vers la droite). Attention, ces fonctions ne doivent pas déplacer la raquette en dehors de la zone visible de l'écran et ne doivent pas lever d'erreur.

Les fonctions de déplacement seront appelées selon une interaction possible offert par le jeu (dans le jargon informatique, nous les appellerons les contrôleurs). Vous pouvez en l'occurrence appeler vos fonctions de déplacement selon les événements suivants :
— sur un clique souris avec `canvas_mouse_click_press` ou `canvas_mouse_click_release`.
— avec une touche du clavier avec `canvas_keypressed` ou `canvas_keyreleased`.
Vous pouvez bien sur adapter plusieurs actions pour le même traitement dans votre jeu afin de faciliter la prise en main si vous le souhaitez.

4.2 Préparation des balles

Dans cette première nous devez concevoir le type des balles (`t_ball`) qui se caractérise avec une position dans le monde, son vecteur vitesse (en nombre de pixel par frame) et sa taille. D'un point de vue affichage une balle se représente par un cercle. Comme précédemment, vous devez modifier le type `t_camlbrick` pour pouvoir mémoriser la ou les balles présentes dans le niveau. Attention, le nombre de balles peuvent évoluer au cours de la partie.

Nous allons maintenant préparer les accesseurs pour nos balles afin que l'interface puissent les afficher, pour cela vous devrez faire les fonctions suivantes :
— `has_ball` indique si la partie en cours possède des balles.
— `balls_count` renvoie le nombre de balle présente dans une partie.
— `balls_get` récupère toutes les billes d'une partie.
— `ball_get` récupère la i^e balle d'une partie, avec $i \in [0; n[$ où n est le nombre de balles.
— `ball_x` renvoie l'abscisse du centre d'une balle.
— `ball_y` renvoie l'ordonnée du centre d'une balle.
— `ball_size_pixel` indique le diamètre du cercle représentant la balle en fonction de sa taille.
— `ball_color` donne une couleur pour une balle.
Enfin pour préparer la prochaine itération nous terminons cette itération avec des fonctions qui seront manipulées dans la suite pour détecter certains comportements. Pour cela, je vous propose de faire les deux fonctions :
— `ball_modif_speed` qui modifie la vitesse d'une balle en accumulant avec un vecteur donnée en paramètre. Il permettra ainsi d'augmenter ou de diminuer la vitesse de la balle.
— `ball_modif_speed_sign` qui modifie la vitesse d'une balle par multiplication avec l'argument sv.
Comme vous avez pu le voir sur les démonstrations en cours et sur les images, une balle est représentée par un cercle. Et nous devrons dans la suite détecter si un point est présent dans ce cercle ou non. Pour cela, dans le domaine du jeu vidéo nous utilisons des boîtes de collisions qui sont une vision mathématique simple de nos objets. Ainsi, vous devez préparer les deux fonctions de détection suivantes :
— `is_inside_circle` permet de détecter si un point (`x`,`y`) se trouve à l'intérieur d'un disque de centre (`cx`,`cy`) de rayon `rad`.
— `is_inside_quad` détecte si un point (`x`,`y`) se trouve à l'intérieur d'un rectangle formé

5 Itération 3 : Gestion des collisions et animations

Pour résumé les étapes précédentes, vous avez réalisé l'affichage de toutes les entités de base et nous pouvons déplacer la raquette sur l'écran. Nous allons maintenant animer les balles.

5.1 Animation des balles

L'animation d'une balle consiste juste à faire évoluer la position d'une balle en sommant avec sa vitesse à chaque frame du jeu. Ainsi la balle va évoluer dans la direction de sa vitesse. Compléter la fonction `animate_action` pour animer toutes les balles d'une partie.

A présent, en modifiant la fonction précédente, ajoutez la gestion des collisions avec les bords latéraux et supérieur de l'écran.

Enfin, complétez la fonction `ball_remove_out_of_border` qui renvoie une nouvelle liste sans les balles qui dépassent la zone de rebond et met à jour la partie.

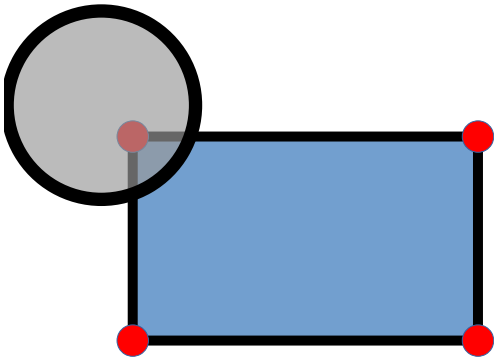
5.2 Gestion des collisions

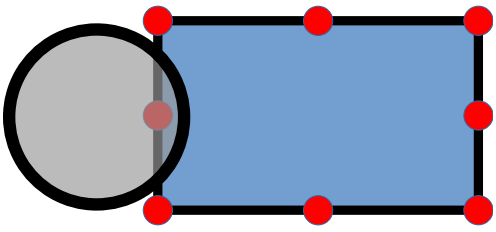
Pour gérer les collisions nous allons d'abord nous intéresser aux collisions balles et briques. Pour vous rassurez, la gestion des collisions est quelques choses de difficile (mêmes les jeux actuels sont truffés de bug de collision), donc nous ne demandons pas une gestion des collisions mais une gestion raisonnable. En particulier, nous pouvons faire deux conceptions bien distinctes :

— Soit nous considérons que le centre de la balle pour faire nos collisions et donc il suffit de tester si le centre de notre cercle est dans le rectangle de la brique et agir en conséquence.

— Soit nous considérons que la balle est un disque avec donc un certain rayon.

Si une balle est un disque, nous pouvons avoir plusieurs cas pour gérer les collisions avec les briques les deux images ci-dessous. A gauche, nous considérons uniquement les 4 coins d'une brique et nous nous contentons de vérifier qu'un coin entre dans le disque pour agir en conséquence (`ball_hit_corner_brick`). Mais cette approche est très approximative. Sur l'image de droite, nous ajoutons plus de points sur la brique en particulier les milieux des arêtes de notre rectangle où nous pourrons détecter plus aisément certaines configurations (`ball_hit_side_brick`).

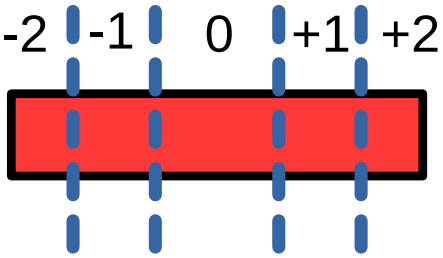




A partir des propositions faites ci-avant et des deux images fournies. Vous devez réaliser la détection des collisions et implémenter les réactions. Les réactions est de partir dans le sens opposé à la collision par défaut mais vous pourrez adapter ce comportement si vous le souhaitez du moment que cela soit raisonnable. Bien évidemment, on n'oubliera pas de déclencher une mise à jour de la brique si nécessaire (`brick_hit`).

Pour information, vous pouvez ajouter une touche qui créera à chaque appuie une balle dans la zone prévue avec une vitesse aléatoire pour tester rapidement vos collisions et le comportement de votre jeu.

A présent, que vous avez réussi à faire les collisions avec les briques, vous devez réaliser la collision avec la raquette (`ball_hit_paddle`). Vous pouvez vous inspirez très fortement de ce que vous avez fait précédemment, mais vous devez ajouter le comportement suivant. Pour que le joueur puisse guider la balle vers une zone votre raquette doit être découpée en un nombre impair de morceaux. Le rebond sur chaque morceau va influencer la vitesse de la raquette en conséquence. L'image ci-dessous montre une raquette morcelée en 5 parties et l'impact de la vitesse en x sur chacune d'elle.



Bien évidemment l'impact en y sur la balle va rester inchangée sur notre exemple (et dans notre démonstration) mais si vous souhaitez modifier celle-ci vous êtes libre de le faire.

Enfin pour terminer cette partie et pour pouvoir faire des découpages de vos tests logiciels, on vous demande (si ce n'est pas déjà fait) de mieux découper la fonction `animate_action` en faisant une fonction qui s'occupe uniquement de l'animation de la balle, une fonction qui s'occupe de toutes les collisions (`game_test_hit_balls`) et une fonction qui met à jour l'état du jeu (détruit les balles en dehors des limites et vérification si la partie est gagnée ou perdue).

6 Itération 4 : Fonctionnalités supplémentaires

Pour cette itération, vous pouvez réaliser vos extensions. Vous connaissez le jeu et vous pouvez ajouter soit une des propositions ci-après, soit toutes. La note prendra en compte l'originalité, la réalisation et l'appréciation par votre enseignant de la difficulté à faire la tâche. Si vous jugez que c'est trop simple vous pouvez réaliser plusieurs extensions.

Proposition si vous n'avez pas d'idée :
— Gestion de l'état du jeu en autorisant la mise en pause du jeu avec le bouton start/stop.
— Gestion d'un score pour chaque brique touchée ou autre.
— Proposez des actions bonus lorsqu'une balle touche une brique bonus (exemple : ajout d'une nouvelle balle, agrandissement de la raquette, ...)

7 Livraison finale

Pour la livraison finale, vous devez réaliser une archive compressée au format ZIP ou TGZ uniquement. Cette archive devra contenir un répertoire nommé `camlbrick_WWW_XXX_YYY_ZZZ` où les lettres doivent être remplacées par les noms des membres d'équipe sans accent ni espace. Ce répertoire doit contenir les éléments suivants :
— les trois fichiers sources fournis au début que vous avez modifié ou non.
— tous les fichiers de tests avec le nom demandé.
— un répertoire `doc` avec la documentation générée complète de toutes les fonctions réalisées/modifiées.
— un fichier rapport qui contiendra la liste des étudiants et les lignes de commande que vous avez utilisez pour pouvoir compiler votre projet avec des captures d'écrans à l'appui.
— les éléments que vos enseignants pourront ajoutés selon la situation.