



BURP

Projet de CP6



13 MAI 2020

PARIS VII

Dao THAUVIN, Liège CHERCHOUR, Thomas BIGNON

Table des matières

1) Informations générales	2
1.1) Informations sur le groupe de projet.....	2
1.2) Lien vers notre projet	2
2) Descriptions des fonctionnalités implémentées ²	3
3) Démonstration du programme	4
4) Description de l'architecture du programme.....	8
4.1) Patrons d'architectures suivis.....	8
4.2) Diagramme d'architecture.....	8
4.3) Choix des bibliothèques externes.....	9
4.4) Description des interfaces des modules principaux.....	10
5) Organisation du travail	11
5.1) Méthodes de travail.....	11
5.2) Outils adoptés	11
5.3) Processus de développement choisi.....	11
5.4) Technique de développement adopté.....	12
5.5) Répartition du travail.....	12
6) Test.....	13
6.1) Modules couverts par les tests.....	13
6.2) Code coverage	13
7) Remarques sur le projet	14

1) Informations générales

1.1) Informations sur le groupe de projet

Notre groupe de 3 étudiants est composée de :

[Dao Thauvin](#)

[Liece Cherchour](#)

[Thomas Bignon](#)

1.2) Lien vers notre projet

Le lien GitHub vers notre projet se trouve à l'adresse suivante :

<https://github.com/daothauvin/burp>

Notre CI est disponible à l'adresse suivante :

<https://app.circleci.com/pipelines/github/daothauvin/burp>

Nous avons utilisées l'issue tracker de GitHub disponible ici :

<https://github.com/daothauvin/burp/issues?q=>

Nous avons également une page internet permettant de visualiser la structure de notre projet à l'adresse suivante :

<https://daothauvin.github.io/burp/html/>

2) Descriptions des fonctionnalités implémentées²

Le sujet de base à était réalisé dans son intégralité.

Aucune extension n'a été réalisé, sachant les extensions du sujet nécessitent un programme permettant de faire une traduction automatique des scripts, ce qui demande beaucoup de temps, nous avons préféré nous concentrer sur les outils de conduite de projet, bien plus intéressant pour ce cours.

3) Démonstration du programme

Il est nécessaire en premier temps d'avoir les librairies indiquées dans le README installé et d'avoir cloner le projet à l'aide de la commande

git clone <https://github.com/daothauvin/burp.git>

Voici les fichiers présents dans le répertoire « burp » après le clone :

```
~/Documents/Cours/CP6/burp master
> ll
total 48
-rw-r--r-- 1 totocptbgn staff 2.7K Apr 22 11:35 CMakeLists.txt
-rw-r--r-- 1 totocptbgn staff 2.5K May 6 19:07 JOURNAL.md
-rw-r--r-- 1 totocptbgn staff 1.1K Apr 22 17:53 LICENSE
-rw-r--r-- 1 totocptbgn staff 3.2K May 13 15:46 README.md
-rwxr-xr-x 1 totocptbgn staff 97B Apr 22 11:35 build_main.sh
-rwxr-xr-x 1 totocptbgn staff 152B Apr 22 11:35 build_test.sh
drwxr-xr-x 4 totocptbgn staff 128B Apr 22 11:35 cmake
drwxr-xr-x 7 totocptbgn staff 224B May 13 15:46 doc
drwxr-xr-x 3 totocptbgn staff 96B May 5 17:01 docs
drwxr-xr-x 4 totocptbgn staff 128B May 13 15:46 scripts
drwxr-xr-x 11 totocptbgn staff 352B May 13 15:46 src
drwxr-xr-x 7 totocptbgn staff 224B Apr 22 12:27 tests

~/Documents/Cours/CP6/burp master
>
```

Ensuite, vous pouvez compiler le projet à l'aide d'un de nos 2 scripts Shell :

- **./build_test.sh**
- **./build_make.sh**

Le premier script Shell permet de compiler et de lancer les tests alors que le second permet de compiler le main.

```
~/Documents/Cours/CP6/burp
-- Generating done
-- Build files have been written to: /Users/totocptbgn/Documents/Cours/CP6/burp/build
Scanning dependencies of target game
[ 5%] Building C object src/model/game/CMakeFiles/game.dir/arene.c.o
[ 11%] Building C object src/model/game/CMakeFiles/game.dir/commands.c.o
[ 17%] Building C object src/model/game/CMakeFiles/game.dir/missile.c.o
[ 23%] Building C object src/model/game/CMakeFiles/game.dir/robot.c.o
[ 29%] Linking C static library libgame.a
[ 29%] Built target game
Scanning dependencies of target view
[ 35%] Building C object src/view/CMakeFiles/view.dir/ui.c.o
[ 41%] Linking C static library libview.a
[ 41%] Built target view
Scanning dependencies of target controller
[ 47%] Building C object src/controller/CMakeFiles/controller.dir/controller.c.o
[ 52%] Linking C static library libcontroller.a
[ 52%] Built target controller
Scanning dependencies of target file_reader
[ 58%] Building C object src/model/file_reader/CMakeFiles/file_reader.dir/interpreter.c.o
[ 64%] Building C object src/model/file_reader/CMakeFiles/file_reader.dir/syntax_analyse.c.o
[ 70%] Linking C static library libfile_reader.a
[ 70%] Built target file_reader
Scanning dependencies of target model
[ 76%] Building C object src/model/CMakeFiles/model.dir/cycle.c.o
[ 82%] Linking C static library libmodel.a
[ 82%] Built target model
Scanning dependencies of target burp
[ 88%] Building C object src/CMakeFiles/burp.dir/main.c.o
[ 94%] Building C object src/CMakeFiles/burp.dir/game.c.o
[100%] Linking C executable burp
[100%] Built target burp

~/Documents/Cours/CP6/burp master
>
```

Ensuite, pour exécuter le programme vous pouvez effectuer la commande suivante :

```
~/Documents/Cours/CP6/burp master
> ./build/src/burp scripts/shooter scripts/shooter scripts/shooter scripts/shooter
```

Voici l'affichage d'un déroulement de partie.

```
burp: ./build/src/burp scripts/shooter scripts/shooter scripts/shooter

Arena

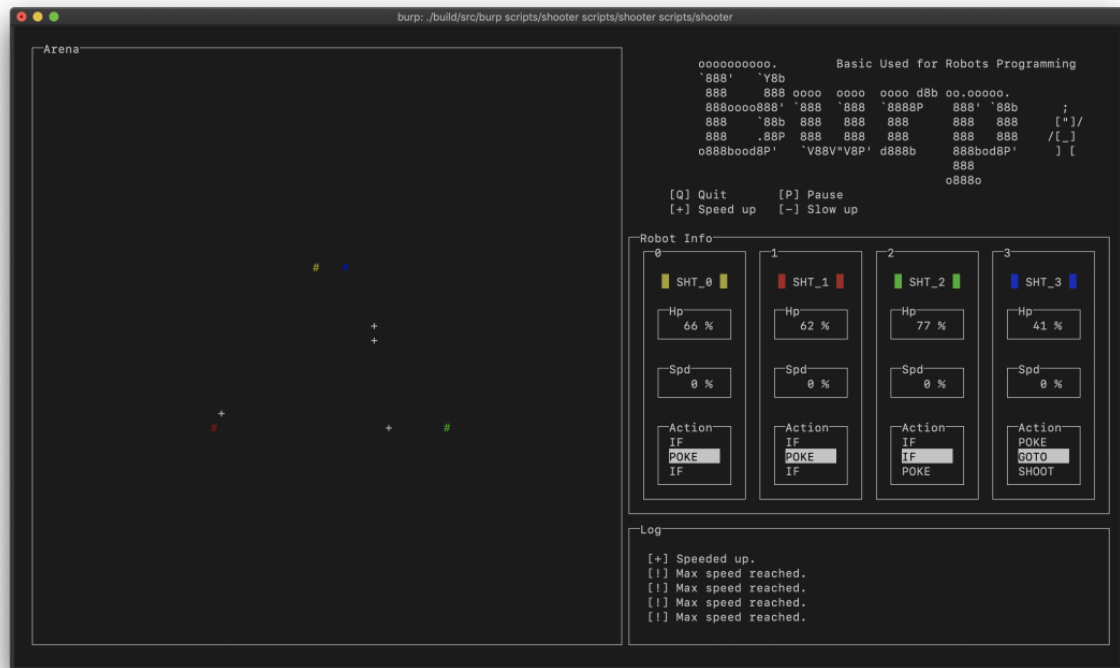
8888888888 d8b      888      888      888
888      Y8P      888      888      888
888      888      888      888      888
888888888 888 d88P*88b 888 *88b 888 888
888      888 d88P*88b 888 *88b 888 888
888      888 d88P*88b 888 *88b 888 888
888      888 Y88b 888 888 888 Y88b.  "
888      888 "Y88888 888 888 "Y888 888
      888
      Y8b d88P
      "Y88P"

Basic Used for Robots Programming
'888' `Y8b
888 888 0000 0000 0000 d8b 00.00000.
8880000888' `888 `888 `8888P 888' `88b ;
888 `88b 888 888 888 888 888 888 [*]/
888 .88P 888 888 888 888 888 888 /[_]
o888b00d8P' `Y88V"Y8P' d888b 888b0d8P' ] [
      888
o888o

[Q] Quit [P] Pause
[+] Speed up [-] Slow up

Robot Info
0 1 2 3
[ ] [ ] [ ] [ ]

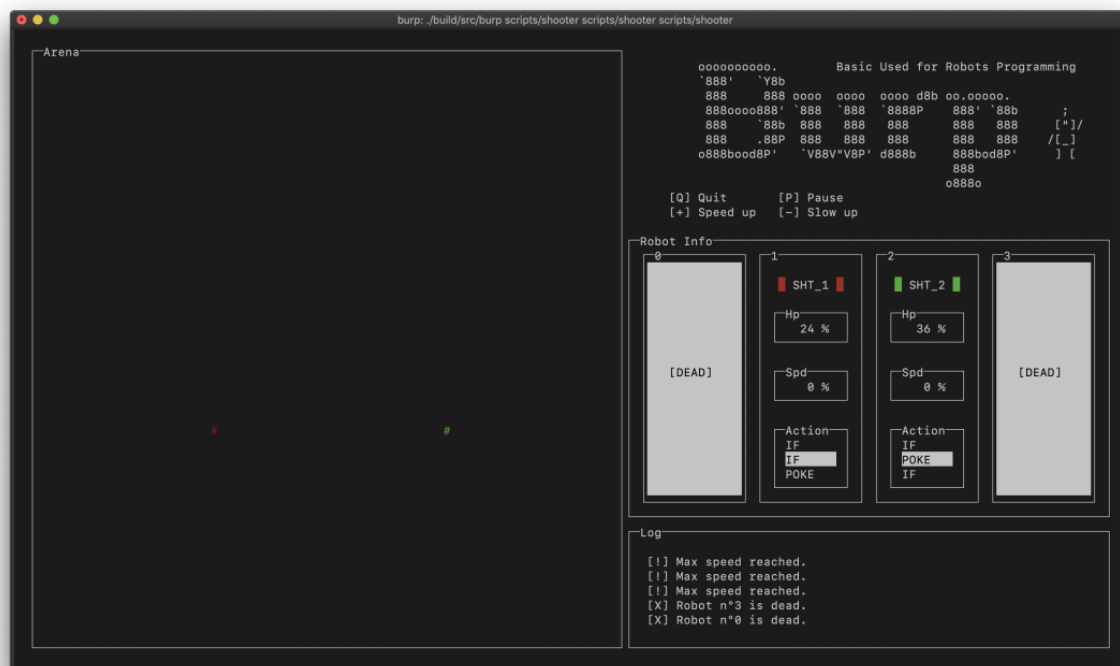
Log
```



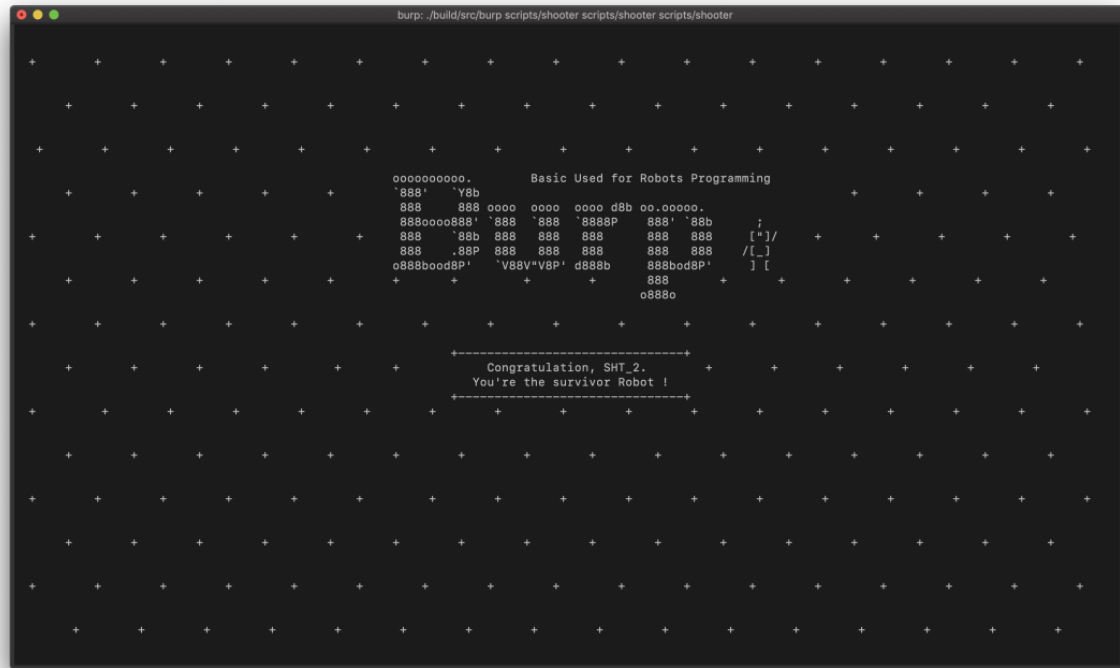
Nous pouvons voir en **haut à droite** les différentes touches permettant d'interagir avec le programme.

En **bas à droite**, il s'agit de l'écran de log, on peut y voir les morts des robots, des avertissements, ...

Au-dessus on peut voir, pour chaque robot, quelle commande est exécutée, le nom du robot, sa vie et sa vitesse.



Lorsqu'un robot meurt, sa colonne est alors grisée et nous affichons DEAD, de plus un log nous indique la mort du robot.



A la fin du programme, le robot gagnant a son nom affiché.

4) Description de l'architecture du programme

4.1) Patrons d'architectures suivis

- **Modèle-vue-contrôleur** : Notre programme sépare vue, contrôleur et modèle.
- **Architecture « par interprétation »** : Le sous-module « file_reader » de modèle.
- **Architecture « par abstraction des données »** : Le sous-module « game » de modèle.

4.2) Diagramme d'architecture

Diagramme des modules :

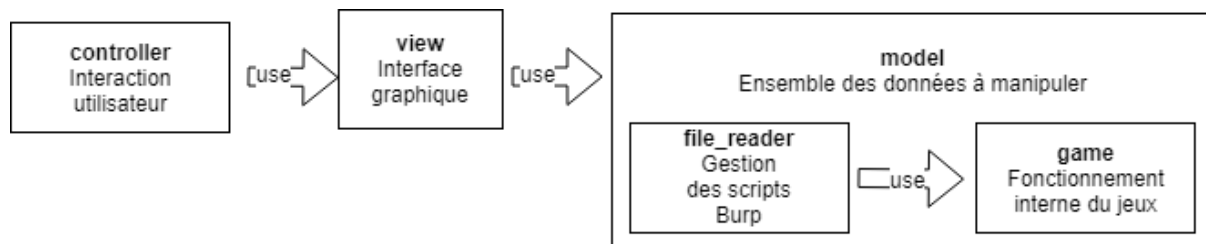


Diagramme d'un tour de jeu :

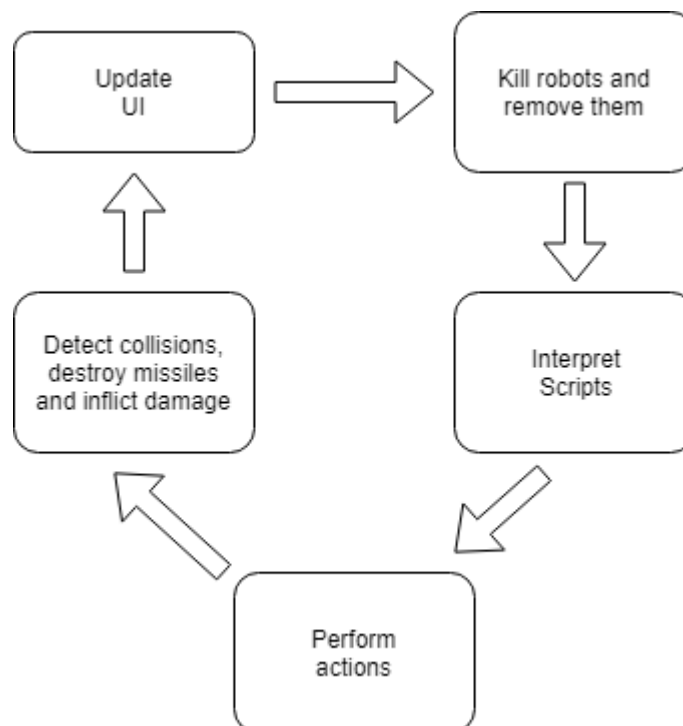
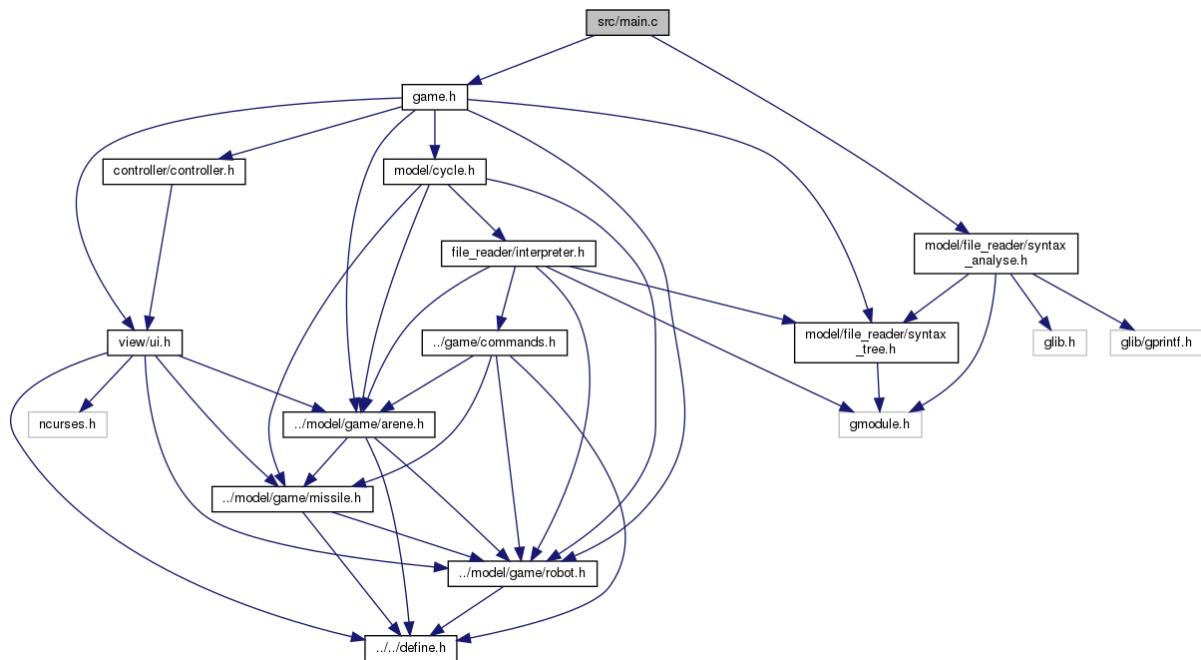


Diagramme des fichiers :



4.3) Choix des bibliothèques externes

Nous utilisons pratiquement que les bibliothèques indiquées dans le sujet pour 2 raisons :

- Limiter les dépendances externes qui ne nous aident pas beaucoup
- Une partie de la note vient de l'utilisation de ses bibliothèques, nous avons donc par exemple préférer l'utilisation du scanner de « Glib » à un autre scanner surement plus facilement utilisable et plus efficace

De plus, nous utilisons la bibliothèque externe « Check » pour pouvoir effectuer nos tests.

4.4) Description des interfaces des modules principaux

Model :

- Depuis extérieur, les seules fonctions utiles sont dans le fichier **cycle.h**, à chaque cycle on appelle comme le dit son nom, la fonction **cycle**. **init_next** doit être appelé avant le premier cycle permettant d'initialiser les premières actions que feront les robots et **getNextCommand** permet de récupérer cette action qui sera mise à jour à chaque tour de jeu.
- **file_reader**
 - Il permet de créer un arbre de syntaxe à partir d'un fichier avec **init_file_tree**, il est possible que l'arbre ne soit pas créé, une erreur a donc eu lieu récupérable avec **message_error**. Des fonctions permettant de libérer la mémoire allouée par l'analyse syntaxique. La fonction **interprete** permet d'interpréter une ligne de commande d'un arbre, on peut aussi récupérer la commande d'une ligne de l'arbre de syntaxe avec **getLine**. Nous avons ajouté une gestion de warning pour les cas possibles dans le langage burp mais peuvent poser des problèmes. Accessible avec **getWarnings** et qui doivent être libérés avec **freeWarnings**.
- **game** :
 - Un ensemble de getter et setter pour les robots, les missiles et l'arène avec d'autres fonctions utiles, notamment des fonctions permettant de détecter les collisions. Un fichier **commands.h** contient l'ensemble des expressions et des commandes qui seront appelées dans **file_reader** pour l'interprétation.

UI :

- On lance l'interface avec **void init()** et lance l'animation de départ avec **anim_begin**, pendant chaque cycle nous appelons **updateArena** pour mettre à jour l'affichage de l'arène. Nous pouvons ajouter des logs dans l'écran de log avec **add_log** et ajouter des actions dans la liste d'action d'un robot avec **add_action**. A la fin de la partie, on lance l'écran de fin avec **end_screen** et on enlève l'affichage avec **quit**.

Controler :

- L'interface contient une seule fonction **waitForInput** à appeler entre chaque tour de jeu

5) Organisation du travail

5.1) Méthodes de travail

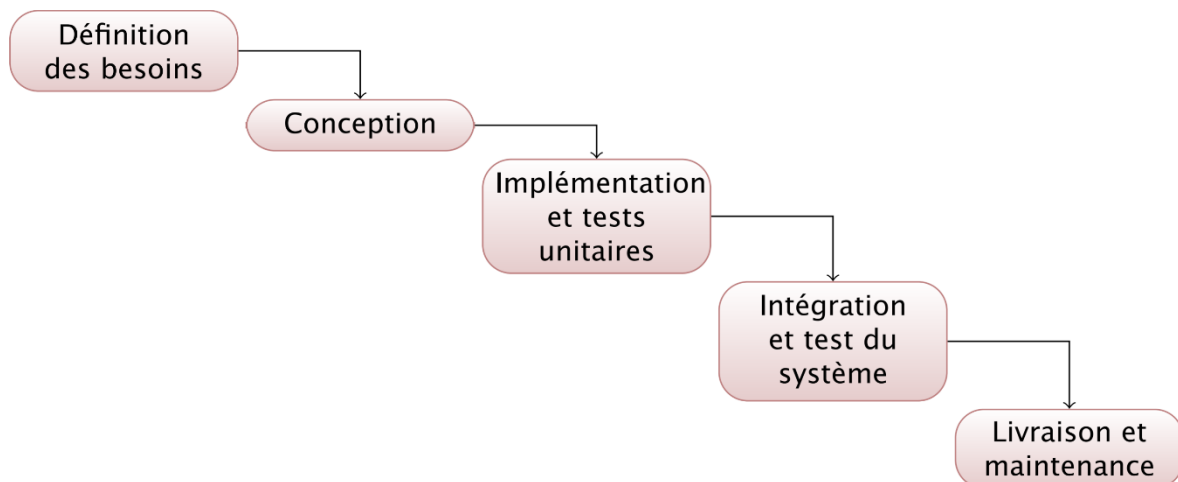
Avant le confinement, durant le cours dédié nous parlions des problèmes rencontrés, nous discussions entre nous de ce que nous avons fait et de ce que nous allions faire pour la semaine prochaine. Nous travaillons chacun de notre côté sinon. Pendant le confinement nous avons essayé de maintenir cela.

5.2) Outils adoptés

- Pipeline pour GitHub
 - o [CircleCi](#)
- Générateur de documentation
 - o [Doxygen](#)
- Générateur de make :
 - o [CMake](#)
- Outil de tests :
 - o [Check](#)
- Outils d'analyse d'exécution
 - o [Valgrind](#)
 - o [gdb](#)

5.3) Processus de développement choisi

Nous avons utilisé le modèle en cascade



5.4) Technique de développement adopté

Nous avons utilisé une technique TDD pour la partie model du sujet, avec en supplément de l'intégration continue pour ne pas avoir à compiler à chaque fois les tests sur notre machine.

Lorsque le projet a bien avancé, nous avons utilisé le système d'issues de Git qui nous a permit de faire du code review avant chaque merge request effectué par chacun.

5.5) Répartition du travail

- Interface graphique : Thomas BIGNON
- Ensemble du jeux (arenas, robots, missiles, commandes) : Liece CHERCHOUR
- Gestion de la lecture des fichiers et interpretation : Dao THAUVIN
- Gestion des actions utilisateurs : Thomas BIGNON, Dao THAUVIN
- Rassemblement des différentes parties : Dao THAUVIN, Liece CHERCHOUR

6) Test

6.1) Modules couverts par les tests

Nous avons un ensemble de 63 tests pour l'ensemble du model, avec :

- Des tests pour le file_reader
- Des tests pour les structures du jeu

6.2) Code coverage

Module file_reader :

Dans tests/check_file_reader/test_files se trouvent des fichiers permettant de tester notre programme.

Nous testons :

- L'ensemble de la sémantique du langage burp en testant chaque commande et expression du langage (Les fichiers commençant par s).
- Les différents messages d'erreurs possibles durant la construction de l'arbre de syntaxe (Les fichiers commençant par f).
- Les différents messages de warning possibles durant l'interprétation (Les fichiers commençant par w)

Module game :

Nous testons :

- Nous testons l'ensemble des getters setters de notre arene, de nos robots et de nos missiles avec les cas limites pour chacun
- Nous testons l'ensemble des commandes pouvant être utilisé dans un script burp en prenant en compte leur effet sur les robots et les missiles soit cohérent.

7) Remarques sur le projet

Le fait de n'utiliser que des entiers pour l'interprétation même a des problèmes de précision, il est possible que deux robots se tirent dessus en restant immobile sans jamais pouvoir se toucher