José MAILLARD
Lucas TRAMPAL

Tuteur : Roland GROZ

# Ensimag
# Année 2017-2018

## Développement d'un émulateur web pour le Bull Gamma 3 E.T.

# Remerciements

Nous tenons à remercier Messieurs Roland Groz et Alain Guyot pour leur suivi tout au long de ce projet filé, Monsieur Vincent Joguin, dont l'émulateur sous DOS nous aura été d'une grande aide, ainsi qu'ACONIT pour son fond documentaire sur la machine.

# Table des matières

**Remerciements** 1

**Introduction** 3

**Présentation synthétique du projet** 4
- Le Bull Gamma 3 E.T. 4
- Fonctionnalités attendues 5
- Outils utilisés 6
  - Node Package Manager (npm) 6
  - Angular 6
  - Mocha 6
  - GitHub 6

**Développement** 7
- Organisation 7
  - Méthodologie 7
  - Analyse logicielle 8
- Fonctionnalités 10
  - Éditeur de code 10
  - Exécution et débogage 11
  - Tambour magnétique 12
  - Programmes pré-enregistrés 13

**Bilan technique** 14

**Conclusion** 15

**Webographie** 16

# I. Introduction

Dans le cadre de notre deuxième année à l'Ensimag, nous avons travaillé sur un projet filé durant un semestre ayant pour but de réaliser un émulateur web pour un calculateur des années 50 : le Bull Gamma 3. Pour ce faire, nous avons utilisé le langage JavaScript. L'interface graphique a elle été développée à l'aide d'AngularJS. Ce projet a été réalisé sous la tutelle de M. Roland Groz et commandé par M. Alain Guyot pour ACONIT (Association pour un conservatoire de l'informatique et de la télématique). ACONIT est une association qui a pour but de "favoriser la conservation, la diffusion et le développement du patrimoine matériel, intellectuel et des savoir-faire constitués au cours de l'évolution de l'informatique". Le développement d'un émulateur, qui plus est en JavaScript et pour le Web, a été pour nous une expérience nouvelle qui nous a permis d'acquérir des connaissances neuves en informatique.

Le résultat final de ce projet devait être un émulateur complet du jeu d'instructions de la machine et fonctionnel sur n'importe quel navigateur web. Notre but est de rendre accessible au plus grand nombre cette machine méconnue. La finalité à long terme de ce projet est sa mise à disposition lors d'une exposition sur l'Histoire de l'informatique dans l'un des halls de l'école.

Ce rapport décrit dans un premier temps l'objectif à atteindre en détail : quel était l'émulateur que nous souhaitions obtenir ? Dans un second temps, il présente notre organisation lors de ce projet, c'est-à-dire notre gestion d'équipe et notre analyse logicielle du problème. Il détaille ensuite le développement de notre jeu et chacune des fonctionnalités implémentées. Pour finir, il résume les résultats obtenus.

# II. Présentation synthétique du projet

## A. Le Bull Gamma 3 E.T.

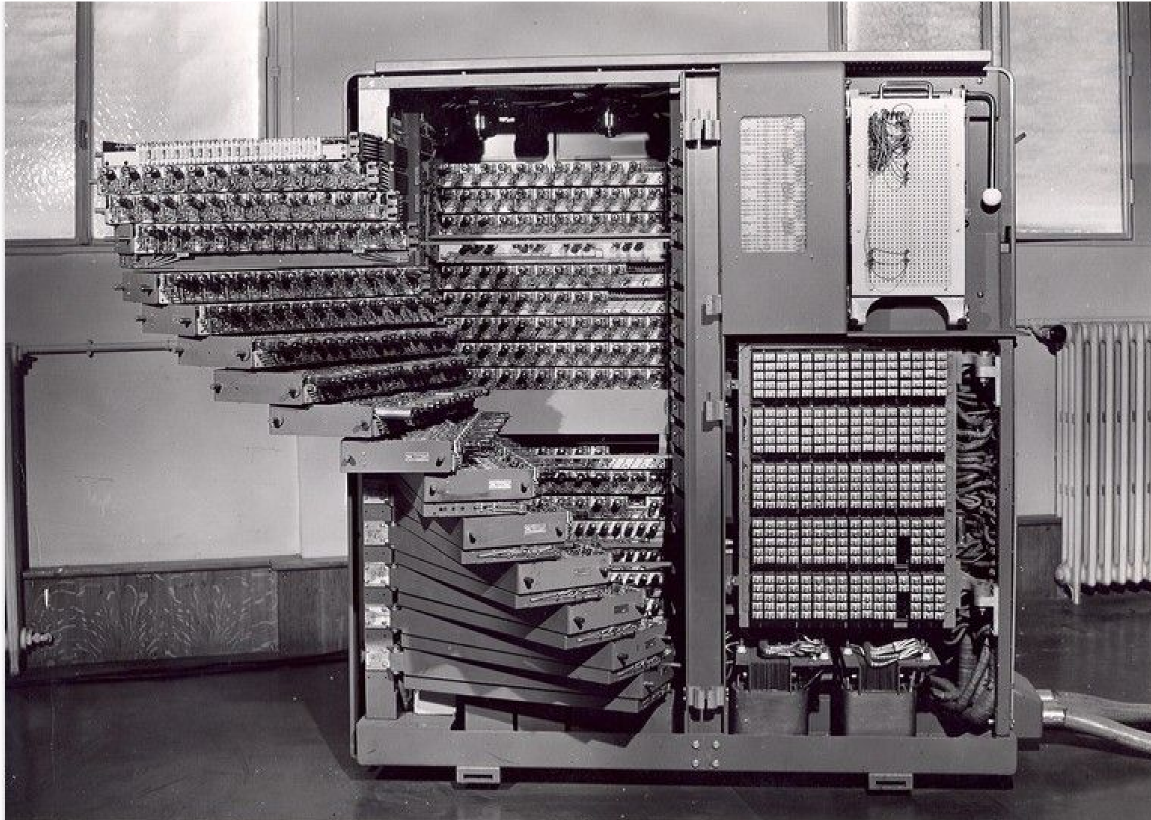

Figure 1 : Armoire du Bull Gamma 3

Le Bull Gamma 3 est un calculateur vendu à partir de 1953 par la société française Bull. Sa technologie précède celle des transistors, il fonctionne donc à l'aide de lampes et de diodes au germanium. Bien que physiquement très éloigné des machines modernes, son architecture interne et son fonctionnement logique ne sont pas sans rappeler ce que nous connaissons aujourd'hui. Des mémoires volatiles de douze chiffres décimaux (les tiroirs visibles sur la figure 1) peuvent être altérées à l'aide d'un jeu d'instructions similaires à ceux actuels. La machine originelle, sans extension, possédait sept de ces mémoires et seule la première pouvait subir une affectation à l'issue d'un calcul du fait du fonctionnement physique du calculateur. Les résultats devaient être transférés vers les autres mémoires dites “banales”. La programmation sur le Gamma s'effectuait initialement via un tableau de connexions qui devait être câblé à la main afin d'encoder les instructions. Une mémoire dédiée stockait le numéro de ligne courant, à la manière du compteur ordinal que nous connaissons dans les architectures modernes.

L'Extension Tambour (E.T.), apparue quelques années plus tard, complète le Gamma 3 par davantage de mémoires, un mode binaire qui transforme ces mémoires de douze chiffres décimaux en mémoires de quarante-huit bits ainsi que par un dispositif de stockage persistant : le tambour

magnétique, visible en figure 2. Plusieurs capacités de tambours existaient, la plus grande permettant de stocker pas moins de quatre-vingt-seize ko de données ! En plus de tout cela, il devenait possible d'exécuter du code lu à partir des mémoires banales, que l'on appelait *séries* lorsqu'elles étaient traitées comme des instructions, et plus seulement à partir du tableau de connexions.

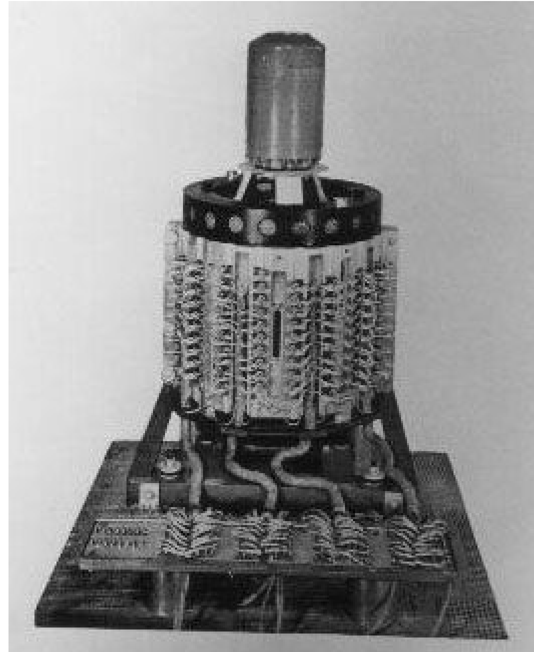

Figure 2 : Tambour magnétique

## B. Fonctionnalités attendues

Un autre émulateur pour le Gamma 3 existe en réalité déjà. Il s'agit d'une version développée par M. Vincent Joguin, fonctionnant uniquement sous DOS, un système d'exploitation aujourd'hui obsolète qui fonctionnait sans souris. L'ergonomie de l'émulateur existant n'est plus tout à fait au goût du jour. De plus, le faire fonctionner nécessite l'utilisation d'un autre émulateur, cette fois-ci pour exécuter DOS sur un système moderne… Notre objectif était donc d'obtenir au moins un émulateur fonctionnant aussi bien que celui de M. Joguin, c'est à dire émulant tout le jeu d'instructions du Bull Gamma 3 E.T., mais avec une interface plus attrayante et capable de fonctionner sur tous les navigateurs web. M. Guyot souhaitait aussi que l'on puisse modéliser le tableau de connexions à partir des instructions entrées au clavier et que l'inverse soit également possible, c'est-à-dire câbler le tableau via l'interface pour ensuite générer des instructions. Un objectif secondaire que nous nous étions fixé était l'émulation du temps d'exécution réel des instructions.

## C. Outils utilisés

### 1. Node Package Manager (npm)

*npm* est le gestionnaire de paquets officiel de Node.js. Il permet de télécharger et garder à jour des bibliothèques de code (paquets). Nous avons créé notre propre paquet Node.js pour ce projet. Il contient

le moteur d'exécution, c'est-à-dire les algorithmes qui permettent l'émulation du Bull Gamma. Ce module est utilisé par l'interface graphique, grâce à npm.

## 2. Angular

Angular est un *framework* qui facilite le développement d'applications web. Il fournit des outils pour lier les vues (*templates* HTML) à du code TypeScript. Il permet également d'intégrer facilement des modules complémentaires tel que celui du moteur d'exécution que nous avons crée. Un projet Angular doit être compilé pour obtenir du code JavaScript standard.

## 3. Mocha

Mocha est un *framework* très populaire qui permet l'écriture de tests pour du code JavaScript. Il fournit une bibliothèque de fonctions et d'objets qui automatisent l'exécution des tests et facilitent leur écriture. Il permet également d'obtenir un rapport très clair à l'issue de l'exécution. C'est ce *framework* que nous avons utilisé pour tester le moteur de l'émulateur.

## 4. GitHub

GitHub est une plateforme internet qui permet l'hébergement gratuit de code *open source* en utilisant le gestionnaire de version Git. En plus de cela, GitHub fournit de nombreux outils et services qui aident à la présentation d'un projet *open source* comme le notre ou bien facilitent la gestion de projet. On y retrouve entre autres un système de tableaux pour la méthodologie *Scrum*, un wiki personnalisable pour chaque projet ou une page de garde (le *README*) présentant succinctement un projet. En outre, il est possible d'y héberger la documentation d'une bibliothèque logicielle.

Notre projet et son code peuvent donc être trouvés sur GitHub à l'adresse :
https://github.com/lutrampal/bullgammator
La documentation est quant à elle accessible à :
https://lutrampal.github.io/bullgammator

# III. Développement

## A. Organisation

### 1. Méthodologie

La première phase du projet a été la découverte et la compréhension de la machine. Afin de mieux cerner le fonctionnement du Gamma 3, nous disposions du cours de M. Bolliet écrit en 1959. Ce cours destiné aux élèves de l'université de l'époque a été une véritable mine d'or pour nous. Il détaille précisément chaque organe du calculateur et présente une spécification complète de chaque instruction (hors E.T.). Le développement de la partie E.T. a lui été permis grâce au manuel de M. Chabrol similaire à celui de M. Bolliet mais plus succinct.

Il nous a ensuite fallu prendre en main l'environnement de développement. José avait déjà travaillé par le passé avec AngularJS mais je (Lucas) ne disposait d'aucune connaissance sur ce *framework* ni en JavaScript d'ailleurs. Il a en particulier été difficile au début de rendre exportable la bibliothèque du moteur de l'émulateur, afin de la rendre utilisable via l'interface. Il a de plus fallu apprendre à écrire des tests avec Mocha, bien que la documentation de ce dernier outil soit très bien écrite.

Ces deux travaux préliminaires évacués, il a été possible de démarrer le développement. Afin de nous organiser, nous avons classé les différentes fonctionnalités par ordre de difficulté estimée. Notre but était d'obtenir le plus rapidement possible un émulateur très basique mais fonctionnel, capable d'exécuter une poignée d'instructions simples comme un transfert de mémoire à mémoire ou une addition. Les instructions plus avancées comme les sauts de ligne et les multiplications et divisions ont ensuite été développées tandis que le tambour a été conçu en dernier. L'interface graphique a elle été développée en parallèle. La figure 3 résume l'organisation du développement.

| | PHASE 1 | PHASE 2 | PHASE 3 | PHASE 4 |
|---|---|---|---|---|
| Apprivoisement de la machine | José, Lucas | | | |
| Prise en main de l'environnement de développement | José, Lucas | | | |
| Instructions de base | | Lucas | | |
| Editeur de code | | José | | |
| Débogueur | | José | José | |
| Instructions avancées | | | Lucas | |
| Extension Tambour | | | | José, Lucas |
| Visualisation du tambour | | | | José |

Figure 3 : Diagramme de Gantt

Nous avons de plus utilisé la méthodologie du *test-driven development*. C'est-à-dire que pour chaque instruction, nous avons écrit les tests avant d'en développer l'algorithme. Cela permet de s'assurer une bonne compréhension de ce que doit faire une instructions avant de commencer à l'écrire. Nous nous sommes pour ce faire appuyés sur l'émulateur de M. Joguin qui nous permettait de connaître à l'avance précisément la sortie d'une instruction pour une entrée donnée. On aboutit ainsi à une très bonne couverture du code de près de 80%. Nous avons en outre repris le principe de la programmation défensive que nous avions apprise en projet Génie Logiciel. Cela consiste à vérifier la validité des paramètres de toute méthode publique avant d'en exécuter l'algorithme. Il devient ainsi très aisé de repérer l'origine d'un bug dans beaucoup de cas.

## 2. Analyse logicielle

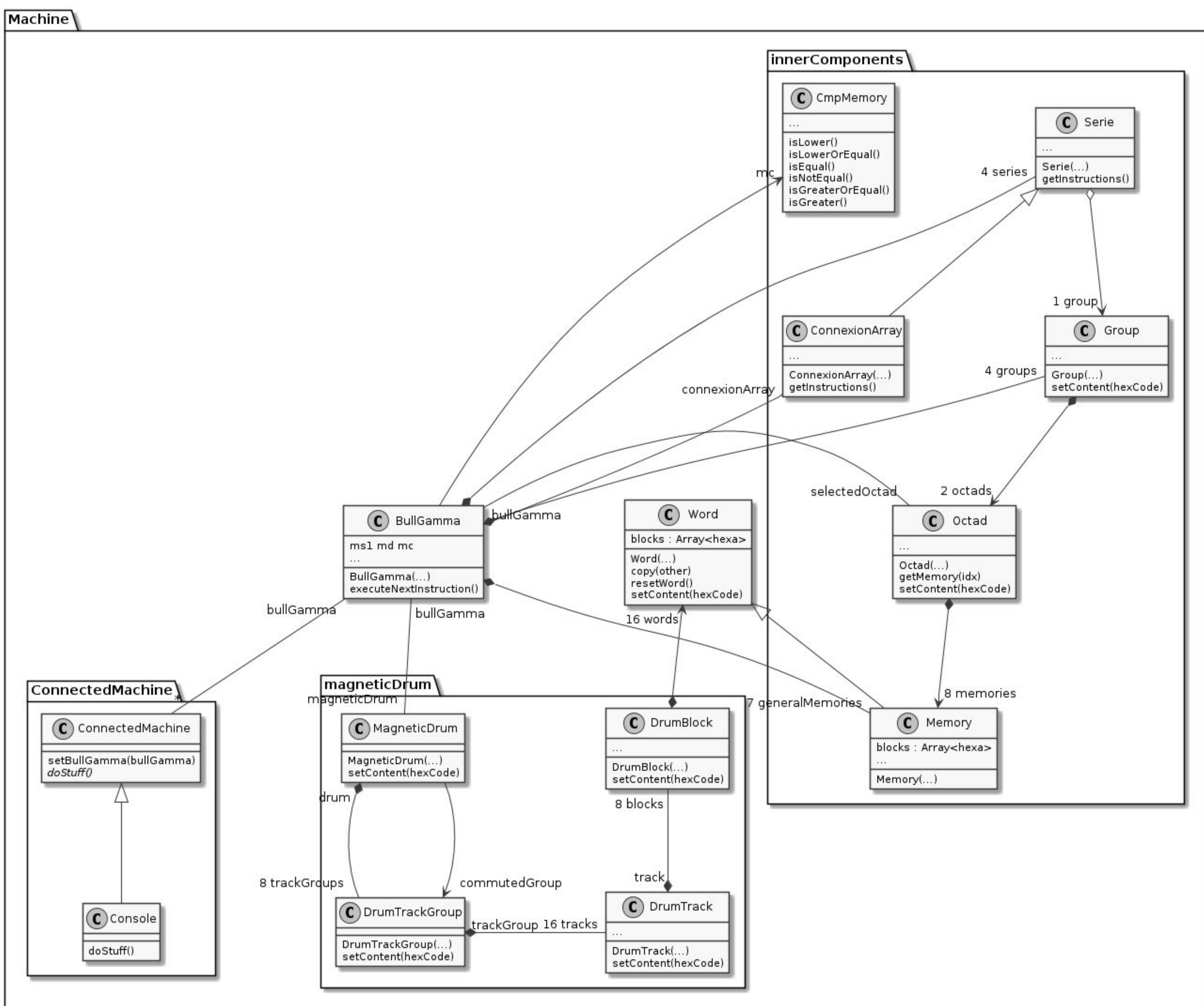


Figure 4 : Diagramme de classes du paquetage Machine (simplifié)

Comme le montre la figure 1, nous avons tenté dans notre conception objet de coller le plus possible à l'architecture physique de la machine réelle. On a ainsi une classe centrale *BullGamma* qui possède tous les éléments du calculateur :

- *CmpMemory* : Une mémoire de comparaison pour stocker le résultat de l'instruction *CN*.
- 7 *GeneralMemory* : Elles représentent les mémoires banales et la mémoire d'opération.
- 1 *ConnectionArray* : Le tableau de connexions.
- 1 *Octad* : Ajoutée par l'E.T., l'octade courante est un ensemble de 8 mémoires qui étend les 7 mémoires générales
- 4 *Group* : Ajoutés par l'E.T, ce sont des ensembles de 2 octades parmis lesquels ont sélectionne l'octade courante via l'instruction *CO*.
- 4 *Serie* : Ajoutées par l'E.T., ce sont en fait physiquement les groupes (ils partagent leur mémoire) mais on lit le contenu des mémoires comme s'il s'agissait d'instructions. Le groupe 3 qui est reliés aux périphériques d'entrées/sorties n'est pas utilisable en tant que série. La quatrième série est donc le tableau de connexions, ce n'est pas un groupe.
- 1 *MagneticDrum* : Ajouté par l'E.T., il s'agit du tambour magnétique. Il est composé de 8 *DrumTrackGroup* que l'on appelle seizaines. Chaque seizaine possède 8 *Track* (des pistes) de 8 Block chacune. Chaque bloc contient 16 *Word* (des mots) de 12 valeurs hexadécimales soit 48 bits. Un bloc contient donc 768 bits, une piste 6144 bits et une seizaine 98304 bits. On a alors une capacité totale pour notre tambour de 786432 bits soit 96 kilo octets.

Une instruction pour le Gamma 3 se compose de 4 valeurs hexadécimales. La première nommée TO (Type Opération) désigne l'instruction qui va être exécutée. La deuxième AD (Adresse) est la mémoire qui servira de seconde opérande. La première opérande est toujours M1. OD et OF sont respectivement l'Ordre de Début et l'Ordre de Fin qui désignent quels chiffres de la mémoire seront utilisés pour le calcul. Il est en effet possible par exemple, d'effectuer une addition entre les nombres représentés par les chiffres en positions 5 à 9 de deux mémoires. Tous les types d'opération n'utilisent pas nécessairement AD, OD et OF mais une instructions fait toujours 4 valeurs hexadécimales. Le cas échéant les valeurs qui ne sont pas utilisées peuvent être fixées arbitrairement.

Chaque instruction est une classe à part entière qui hérite de la classe abstraite *Instruction*. Elle implémente la méthode abstraite *execute()* qui altère l'instance *BullGamma* pour reproduire le comportement de l'instruction. D'autres classes abstraites filles d'*Instruction* ont également été ajoutées lorsqu'une partie de l'algorithme de certaines instructions était généralisable. On utilise alors un patron de méthode : On définit l'algorithme d'*execute()* dans la classe mère et on déclare de nouvelles méthodes abstraites appelées dans *execute()* qui seront redéfinies par les filles. On a notamment la classe *OperationWithPreShift* dont toutes les filles nécessitent un cadrage préalable, c'est à dire un décalage de valeurs (décimales ou binaires). On a également la classe *TrumTransfer* pour les transferts tambour. La figure 5 présente succinctement cette architecture.

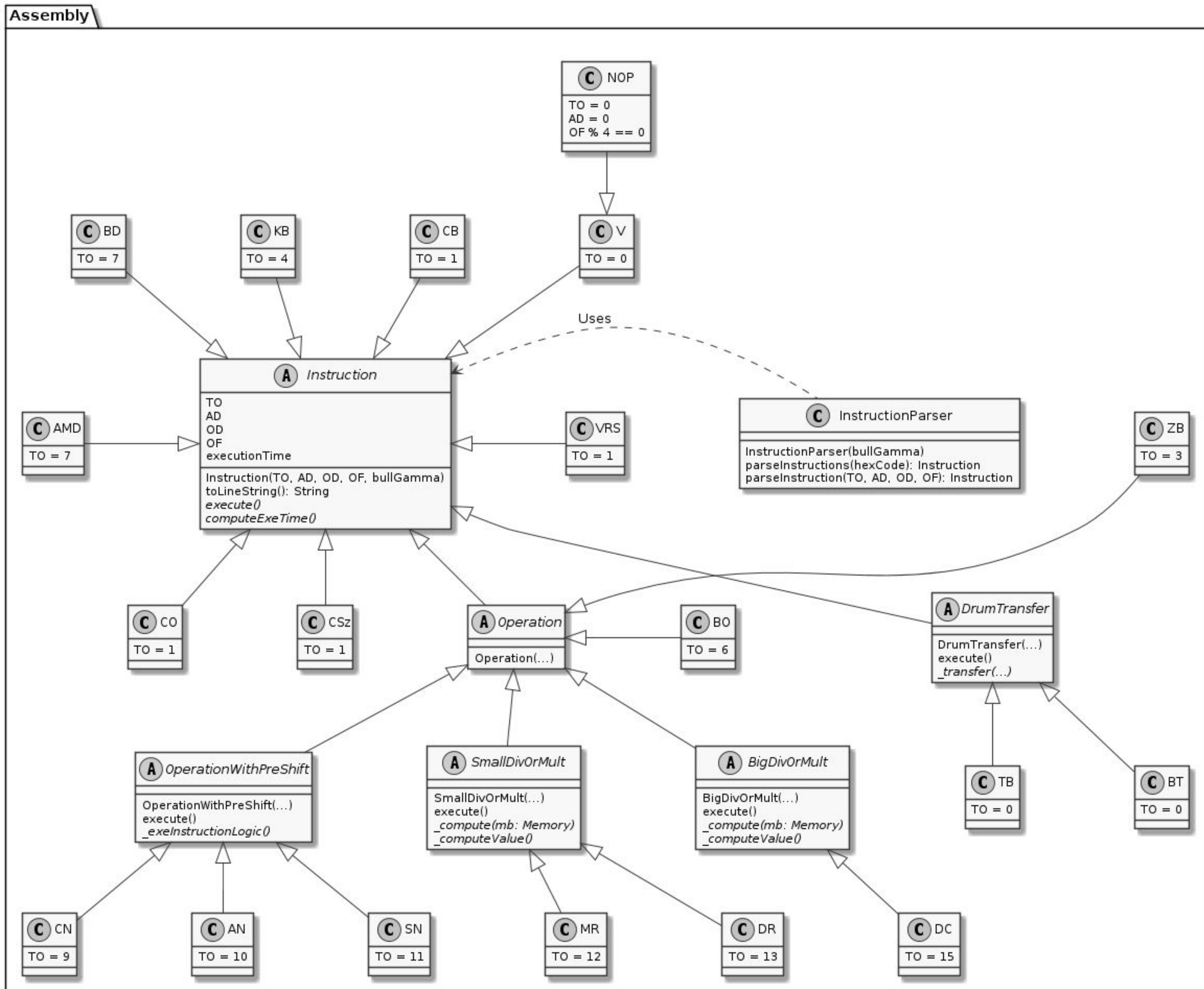


Figure 5 : Diagramme de classes du paquetage Assembly (simplifié)

Quant à l'interface graphique, nous avions besoin d'une architecture modulable et qui implémente le minimum de logique possible, car celle-ci est probablement destinée à être changée. Le diagramme ci-dessous montre comment nous nous sommes adapté à ces contraintes.

En haut, les vues existent indépendamment du moteur d'exécution. Elles sont regroupées entre elles par fonction. Avec Angular, un composant est associé à une vue.

En bas, le moteur d'exécution est complété par des outils qui font également partie de notre bibliothèque Node. Ces outils peuvent être considérés comme un ensemble d'APIs qui permettent d'utiliser facilement le moteur d'exécution :

- *Debug* : Accède aux mémoire en lecture ou écriture
- *Execution* : Lit ou exécute des instructions
- *Editor* : Accède au tableau de connexions ou au tambour en écriture

Chacun de ces outils doit être affecté à une instance de *BullGamma.*

Au centre, l'interface entre les vues et les API du moteur d'exécution. Elle adapte les résultats des APIs pour les vues. Avec Angular, cette couche s'implémente avec des services qui sont injectés dans les vues.

Un service Angular a également été utilisé pour garantir l'unicité de l'instance de BullGamma utilisée afin d'obtenir des résultats cohérents avec la suite de commande donnée.

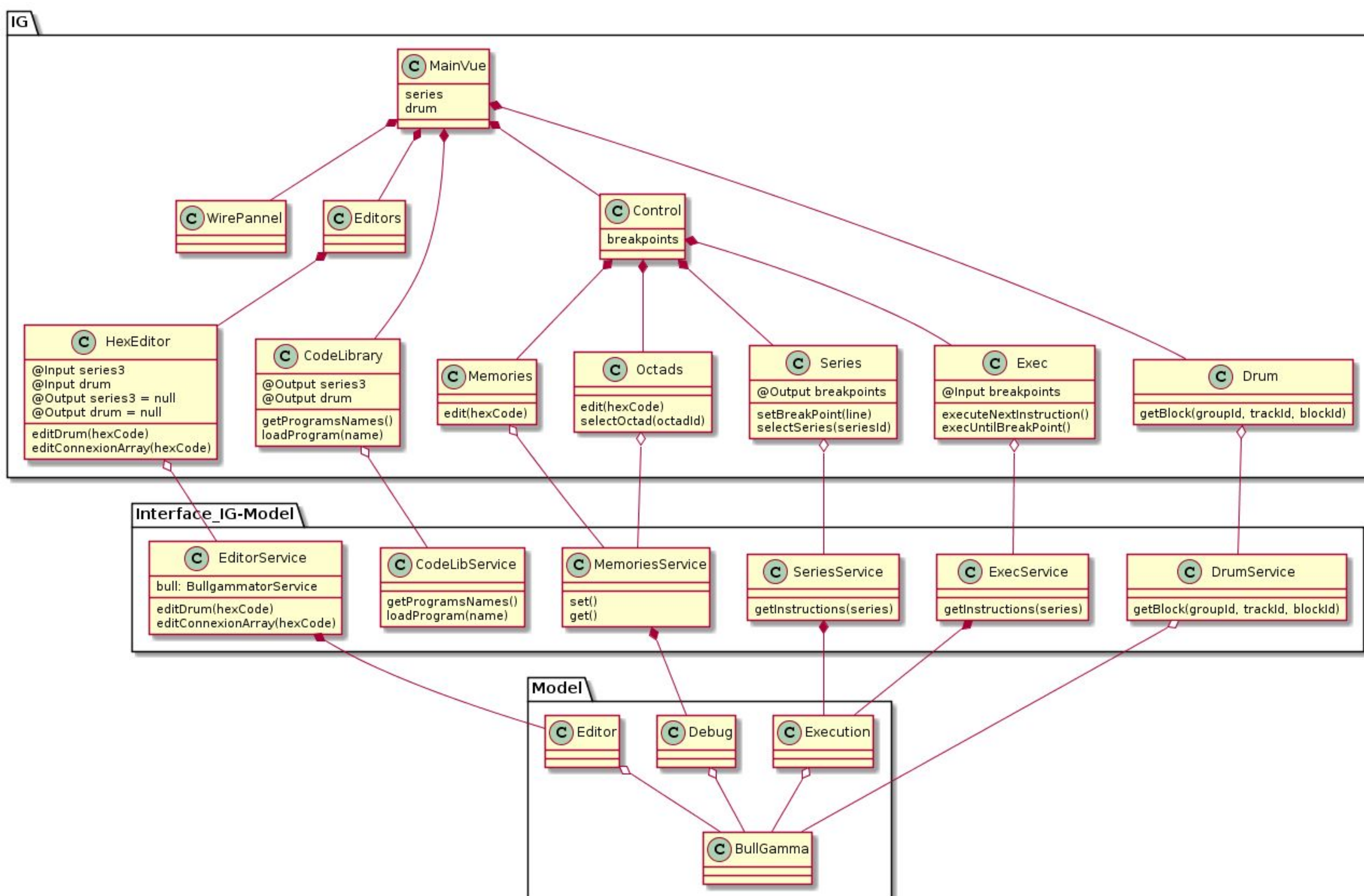


Figure 6 : Diagramme de classes de l'interface graphique

*Les versions complètes des diagrammes peuvent être trouvées dans le dossier *design* du code du projet.*

## B. Fonctionnalités

### 1. Éditeur de code

L'éditeur de code permet de transmettre à la machine le code à exécuter. Un premier onglet permet l'édition du tableau de connexions et un deuxième l'édition du tambour. Lors de la validation, le texte est analysé : les commentaires, les espaces, les retours et tabulations sont retirés ; puis chargé dans la mémoire correspondante. Des erreurs sont levées en cas d'entrées incorrectes. L'analyse syntaxique et le chargement en mémoire sont effectués par la classe *Editor* de notre bibliothèque NodeJS, qui interagit avec une instance de *BullGamma*.

Figure 6 : Éditeur de code

### 2. Exécution et débogage

L'interface d'exécution est découpée en plusieurs zones. On a tout d'abord une console (panneau "Exécution"), qui affiche tout ce qui est écrit sur la sortie du Gamma 3. Dans la réalité, il s'agirait plutôt d'une imprimante à cartes perforées. Le bouton "titiller" (vocabulaire de l'époque) permet de passer à l'instruction suivante tandis que "continuer" exécute tout le code jusqu'au prochain point d'arrêt. Le panneau "Contrôle" présente quant à lui le contenu des mémoires internes à la machine. Il est possible de les éditer et des les réinitialiser manuellement. Ce qui n'aurait bien entendu pas été faisable sur le véritable calculateur.  La panneau "Octade" montre le contenu des octades ajoutées par l'extension tambour. A droite, le panneau des séries montre le code qui sera exécuté. On peut obtenir un descriptif de chaque instruction en passant la souris au dessus et fixer des points d'arrêts en cochant les cases avant chaque instruction. Un point d'arrêt obligatoire est fixé à la ligne 0. En effet, l'émulateur retourne à l'instruction 0 après la soixante-troisième ligne comme le véritable calculateur. Le passage à une autre série nécessite une instruction.

La console est connectée au Bull Gamma comme le serait n'importe quelle machine connectée. Elle implémente des *callbacks* sur les instructions d'extractions statiques (ES) qui lui permettent d'afficher les contenus des mémoires quand le programme le demande. Le paramètre OF de l'instruction ES permet de sélectionner l'extracteur à afficher et le paramètre OD permet de le mode d'affichage de cet extracteur (brut, float Deca, int Deca). La classe *Console* hérite de la classe *ConnectedMachine* et est fournie avec notre bibliothèque NodeJS.

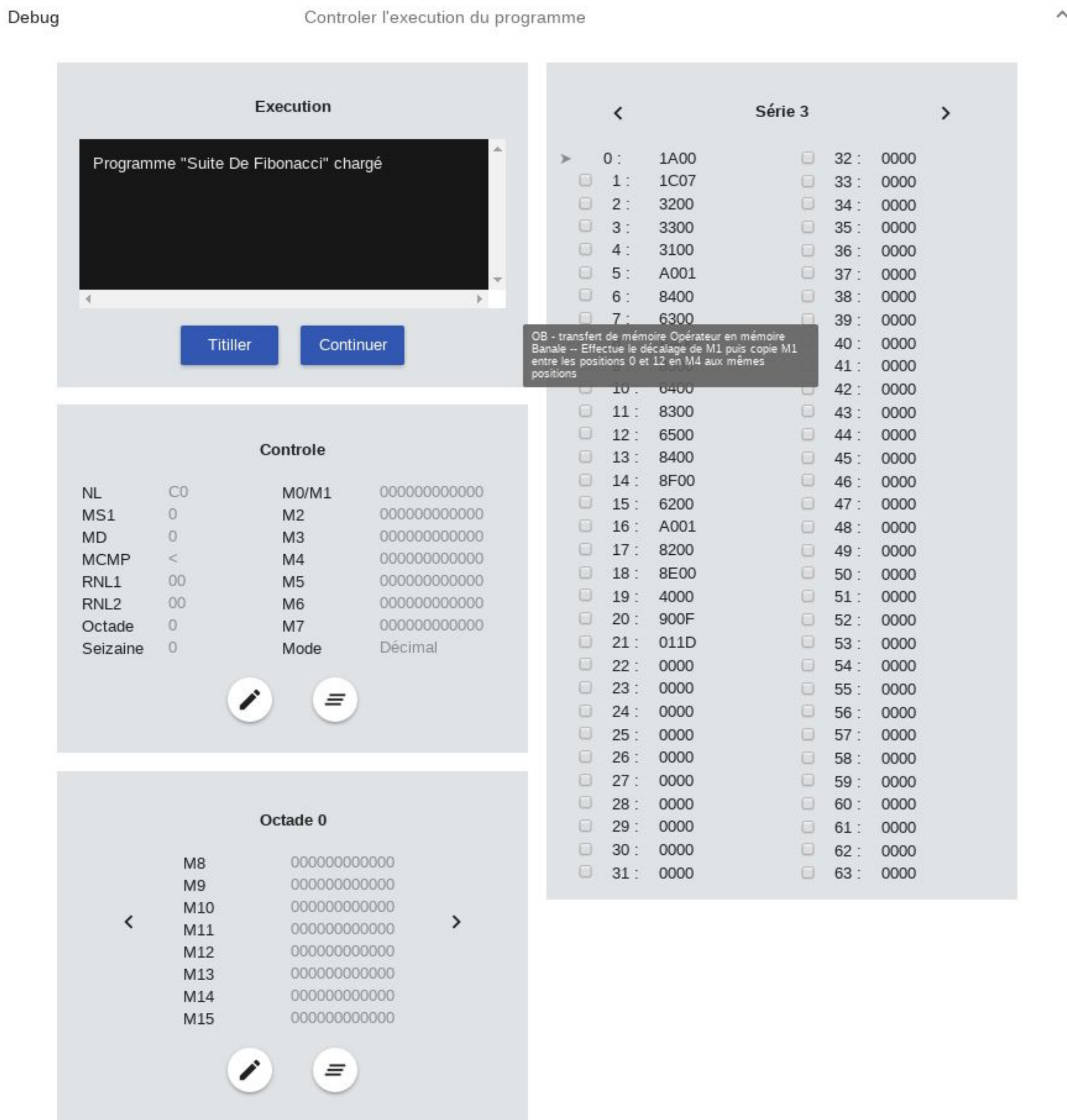


Figure 7 : Interface de d'exécution et de débogage

Le titilleur utilise la classe *Execution* de notre bibliothèque NodeJS qui contrôle une instance de *BullGamma*. A chaque instruction, le compteur programme est incrémenté et l'instruction précédente est exécutée.

Le bouton 'Continuer' utilise la même fonction de la classe *Execution* que le titilleur, il l'exécute en boucle jusqu'à ce que la ligne de l'instruction à exécuter soit marquée d'un point d'arrêt. Si le programme ne passait jamais par un point d'arrêt, l'émulateur serait boucle infinie et ferait crasher le navigateur, le point d'arrêt à la ligne 0 sert à minimiser ce risque.
Les points d'arrêt sont communiqués au composant d'exécution par le composant de visualisation des séries, qui utilise également la classe *Execution* pour obtenir son contenu. Ce contenu est d'ailleurs mis à jour en temps réel grâce au *binding* d'Angular.

Le contenu des mémoires du Gamma (banales et commutées), éditable, est lié à la classe *Debug* de notre bibliothèque NodeJS. Cette classe permet de lire et écrire les mémoire de l'instance de *BullGamma* qui lui est donnée.

## 3. Tambour magnétique

C'est ici que l'on peut lire le contenu du tambour en hexadécimal. On sélectionne de façon intuitive la seizaine, la piste et le bloc que l'on souhaite voir. Les mots de ce bloc sont ensuite numérotés et affichés. Il n'est pas possible de modifier le contenu du tambour via cette interface. L'édition doit se faire via l'éditeur prévu à cet effet.

Figure 8 : Visualisation du tambour magnétique

## 4. Programmes pré-enregistrés

Des programmes de démonstration ont été inclus dans l'interface. Il est par exemple possible de calculer une racine carrée ou bien la suite de Fibonacci. Il suffit pour cela de cliquer sur le programme que l'on veut exécuter et le code sera chargé automatiquement dans les séries et le tambour. Il figure également dans l'éditeur.

Certains programmes, dont le nom se termine par 'Deca' sont obtenus par la compilation de code Deca, un langage de programmation assez simple, en code pour le Bull Gamma. Ce compilateur, non fourni avec l'émulateur, est un autre projet sur lequel j' ai (José) travaillé à l'Ensimag.

Le code des programmes pré-enregistré est lu à partir du fichier *code_samples.json* par la classe *CodeLibrairy* de de notre bibliothèque NodeJS. Ce fichier est généré par le script python *gen_js_code_lib.py* qui compile les fichier textes du dossier *sample_programs* au format de donnée JSON.

Les fichiers textes doivent être un minimum formatés afin d’être compilés correctement :
- le code du tableau de connexions doit être au début du fichier
- il est possible de renseigner le contenu des mémoires banales et du tambour en les encadrant de balises type HTML afin qu’il soit chargé automatiquement

Exemple de fichier de code :

```
-- Intitulé du programme
6400 -- charge la mémoire 4
A001 -- incrémente de 1

<m4>000000000011</m4>
<drum></drum>
```

Lors du chargement de ce fichier, la valeur 11 sera affectée à la mémoire 4 et le tambour sera vide. Le programme, qui utilise la mémoire 4, ajoutera donc 1 à 11 sans que l’utilisateur n’ait eu à intervenir.

La modification ou la création d’un programme nécessite une exécution du script python et un nouveau déploiement du site. Une piste d’amélioration serait donc d’interroger un serveur pour obtenir le fichier *code_samples.json*.

# IV. Bilan technique

A l'issue de ces 4 mois de développement, nous sommes parvenus à obtenir un émulateur parfaitement fonctionnel bien que certaines fonctionnalités n'aient pas pu être implémentées par manque de temps. En particulier, il était prévu de modéliser le tableau de connexions mais cela n'a pu être fait. Le calcul du temps d'exécution des instructions, qui était un objectif secondaire que nous avions fixé nous-même, n'existe pas non plus même si la conception logicielle le prévoit. Il sera donc aisé d'ajouter ces fonctionnalités à l'avenir pour l'équipe qui maintiendra le projet. La séparation moteur d'exécution/interface que nous avons mise en place permet de réutiliser facilement le moteur sur une vue différente. Étant donné qu'ACONIT souhaiterait une modélisation en trois dimensions du calculateur à l'avenir, nulle doute que cela leur sera utile.

# V. Conclusion

Nous sommes dans l'ensemble très satisfaits du logiciel que nous avons développé et nous avons appris beaucoup sur les technologies utilisées. Le JavaScript étant un langage très en vogue, les compétences acquises seront sûrement réutilisées dans le futur. L'aide d'une troisième personne sur le projet n'aurait pas été de refus pour s'occuper notamment du tableau de connexions mais nous n'étions malheureusement pas parvenus à trouver quelqu'un. Nous avons tous les deux pris plaisir à découvrir cette machine qu'est le Gamma 3 et à développer cette application. Nous espérons qu'ACONIT appréciera le résultat et souhaitera la réutiliser à l'avenir.

# VI. Webographie

Article d’ACONIT à propos du Bull Gamma 3 : http://www.aconit.org/spip/spip.php?article246

Fond documentaire d’ACONIT sur le calculateur : http://aconit.org/histoire/Gamma-3/Articles

Cours de M. Bolliet : http://aconit.org/histoire/Gamma-3/Articles/Gamma-Bolliet.pdf

Manuel de M. Chabrol : http://aconit.org/histoire/Gamma-3/Articles/Cours_Gamma_3_Chabrol.pdf

Jeu d’instructions de la machine : http://aconit.org/histoire/Gamma-3/Articles/tableau-de-code.jpg