RASQUIER Vincent
ROUGIER Adrien

IR 1 - JUIN 2017

# Projet Programmation JAVA

# Rapport : Machine à registres non limités

1. Objectifs du projet
2. Choix d'implémentations
3. Problèmes rencontrés

ESIPE MLV

Enseignants : DURIS Etienne
VIALETTE Stéphane

# Introduction

Au cours du second semestre cette première année d'école d'ingénieur, les cours de programmation JAVA nous ont permis de découvrir la programmation orientée objet. L'objectif a principalement été d'apprendre les rudiments de développement dans ce langage et les spécificités inhérentes à la programmation-objet.
Pour la première fois dans cette matière, nous avons eu l'occasion de travailler en collaboration sur un projet de fin d'année. Afin de répondre au mieux à la problématique du travail de groupe nous avons utilisé la plateforme git rendu disponible sur la plateforme forge de l'univ-mlv (https://git-etud.u-pem.fr/ir1-rougier-rasquier-projet-java-urm.git).

Le projet que nous avons développé permet de simuler l'exécution d'un programme sur une machine à registre non limité et il est de ce fait capable d'exécuter des commandes URM. Un compilateur est également présent afin de pouvoir utiliser des commandes plus pratiques que celles disponibles en URM. Ces commandes améliorées sont des commandes EURM et doivent être, à terme, traduite en commande URM pour pouvoir être exécuté.

Nous avons couvert tous les points demandés dans le sujet et avons essayé d'apporter une réflexion toute particulière sur la mise en place de notre projet et la propreté de notre code.

# 1. Objectifs du projet

Le projet de fin de semestre a pour but de concrétiser notre apprentissage du langage JAVA. Il consiste en la réalisation d'un logiciel pouvant ouvrir des fichiers ****.urm* ou ****.eurm* contenant les programmes écrits dans ces mêmes langages. Il est également possible de soumettre directement des instructions au logiciel en list dans le code.

Les programmes soumis permettent de réaliser des calculs plus ou moins complexes. À l'issu de l'exécution, le résultat du calcul (stocké dans le registre R1) est affiché.

Il est possible de voir l'état des différents registres de la machine, de connaître la valeur du pointeur de commande, le nombre de commandes déjà exécutées, et le mode de la machine grâce à l'objet StateView. Cet affichage de l'état peut-être invoqué à chaque étape de l'exécution afin d'avoir un rendu temps réel.

Les commandes URM, utilisées exclusivement pour la création de programmes en début de projet, ont laissé place aux commandes EURM. Leur utilisation lors du développement de programme est plus aisée. La machine ne sachant exécuter que des commandes URM, il a été nécessaire de réaliser un compilateur. Les programmes EURM sont donc traduits/compilés en URM avant exécution. Les commandes EURM étant plus nombreuses, elles permettent de réaliser des actions plus complexes en un minimum de lignes.

| URM Commandes | EURM Commandes | |
|---|---|---|
| COPY<br>JUMP<br>SUCCESOR<br>ZERO | ADD<br>COMMENT<br>COPY<br>DEC<br>EQ?<br>GEQ?<br>GOTO | INC<br>LABEL<br>MULT<br>QUIT<br>SUB<br>ZERO<br>ZERO? |

En cas  d'erreur de syntaxe dans le nom des commandes une erreur est envoyée à l'utilisateur. Le programme propose également 3 modes d'exécution. Le mode STRICT, qui arrête le programme en cas d'utilisation de registre non initialisé. Le mode SAFE, qui initialise automatiquement les registres à 0 avant leur utilisation. Enfin, le mode LOOPDETECT permet de détecter les boucles infinies à l'exécution.

En plus d'utiliser une machine qui n'a pas de limite (théorique) en terme de nombre de registre et de valeur d'index, nous avons également implémenté la class BigInteger afin de gérer les valeurs de registres arbitrairement longues.

Un environnement de test a également été mis en place afin de vérifier l'ensemble des fonctionnalités du logiciel.

# 2. Choix d'implémentations

Nous ne pouvons pas parler de nos choix d'implémentations sans d'abord expliquer la logique que nous avons essayé d'appliquer tout au long de ce projet. Cette logique réside dans notre constat de départ que les commandes URM et EURM ne sont finalement pas si différentes que cela. C'est fort de ce constat que nous avons essayé au mieux de factoriser le code en rapport avec ces deux types.

La première class a avoir bénéficié de cette factorisation est la class ProgramAbstract. En effet, à partir du point numéro 3 du projet, il nous a été demandé de regrouper URMProgram et EURMProgram sous une seule interface commune de type Program. Ce regroupement a été l'occasion pour nous de voir que beaucoup de code pouvait être déplacé dans une class abstrait du nom de ProgramAbstract. Cette class permet entre autres de vérifier si un index est une position valide pour un commandPointer, d'obtenir la longueur d'un programme, d'obtenir une commande du programme à un index particulier ou encore d'obtenir l'index du registre maximum utilisé par le programme de l'utilisateur. Pour pouvoir effectuer ce regroupement de code nous avons dû ajouter un type paramétré à la class (type paramétré qui extend l'interface commune de URMCommand et EURMCommand qui est, simplement, Command).

Une autre class à avoir bénéficié du type paramétré est le CommandParser, qui a pour rôle de traiter chaque chaîne de caractère pour en obtenir une commande instanciée.

Ensuite, afin de faciliter la gestion des labels et de rendre possible l'ajout de nouvelle class qui aurait le même effet qu'un label, nous avons créé une interface Anchor qui étend l'interface EURMCommand. Une class qui implémente l'interface Anchor nous permet, à l'aide des visitors du système de compilation, de savoir exactement à quelle ligne se trouve le code qui suit la définition de cette ancre.

Pour faciliter la gestion des ancres et des registres lors de notre processus de compilation, nous avons créé des managers. Ces managers sont en charge du bon déroulement de la compilation des commandes EURM et permettent de fournir de précieuses informations aux commandes lors de leur compilation. Ainsi, le manager des ancres permet de générer des ancres aléatoires dans le cadre d'une utilisation interne d'ancre dans une commande. Il peut également faire la conversion d'une ancre vers son numéro de ligne. Le RegisterManager quant à lui permet d'obtenir des index de registres temporaires ou fixes non utilisés.

Enfin, le coeur de nos choix d'implémentations se situe au moment de la compilation des commandes EURM vers URM. Nous ne pouvons pas vous expliquer le processus de compilation sans avant vous présenter les différentes interfaces EURMCommand.

D'abord, nous avons les commandes de type, d'interface Substitutable (substituable). Cette interface permet de définir une commande EURM qui a pour vocation de se substituer par d'autres commandes EURM. Typiquement, le cas de la commande SUB qui est une commande qui permet de soustraire le contenu d'un registre à un autre. Cette commande peut se substituer avec l'aide des commandes EURM de type DEC, EQTEST, LABEL, GOTO et INC. Une commande substituable va donc renvoyer une liste de commandes EURM qui seront traitées par la suite.

Ensuite, nous avons les commandes de type, d'interface Compilable. Cette interface définit une commande qui est directement compilable en liste de commande URM.

Le processus de compilation va donc se dérouler en trois étapes. La première consiste à récupérer une liste complète de commandes non substituables. Nous allons donc utiliser notre visiteur SubstitutableVisitor pour aller effectuer cette première phase de compilation.

Une fois que nous avons cette liste de commande non substitutable, il faut gérer le cas des ancres. C'est donc naturellement que nous allons utiliser le AnchorVisitor pour aller récupérer la liste des ancres et les associées à un numéro de ligne. Ce visiteur sert également à obtenir la longueur totale du programme.

Enfin, le CompilerVisitor va aller récupérer la liste des commandes URM forte des informations obtenues lors de la phase numéro deux. Ainsi, couplé avec les managers, les visitors permettent un processus de compilation simple et suit, selon nous, une bonne pratique de développement.

# 3. Problèmes rencontrés

Au cours du développement de ce projet, une pré-soutenance nous a permis d'améliorer le code proposé pour la réalisation du programme. L'un des problèmes majeurs remontés lors de la pré-soutenance se situait dans la class Compiler (class qui nous permet de compiler un programme EURM en URM). En effet, pour traiter les différents types de commandes nous utilisions des conditions sur l'instanceof de la commande. Le professeur monsieur Mangue nous a fait remarquer à quel point cette solution était coûteuse en termes de temps mais également une très mauvaise pratique et il nous a orientés sur différentes solutions que nous pouvions mettre en place pour pallier à ce problème. La solution que nous avons finalement retenue est l'utilisation d'un design pattern bien connu en JAVA du nom de : Visitor. Ce design pattern nous permet d'instancier des objets appelés "Visiteurs" et qui vont avoir le rôle de "visiter" une commande afin d'en obtenir le type exact et appliquer un comportement particulier dessus. Nous avons ainsi pu corriger notre code avec l'emploi d'un design pattern performant et correct en tant que pratique de développement.

Suite au bêta soutenance nous avons également pu constater que notre système de factory (de création de commandes) n'était peut-être pas le plus judicieux. En effet, avant cette rencontre avec monsieur Mangue, nous avions une fonction statique dans chaque commande (du nom de build) qui servait à instancier un objet de la class voulu. Notre enseignant nous a, à juste titre, fait remarquer que la logique de création de commandes était assez similaire et que celle-ci pouvait être regroupée en un seul fichier. C'est ainsi que les class URMFactory et EURMFactory sont apparus avec comme objectif de créer les commandes voulues avec un minimum de duplication de code et ainsi plus de simplicité pour la modification et la lecture de notre code.

Cette modification nous a aussi permis d'améliorer le concept derrière la création de commande et de parseur de commande. Ainsi, nous nous sommes rendu compte que la logique derrière le parseur de commande URM et EURM était la même et nous avons ainsi regroupé le code en une seul class avec un type paramétré. Aussi, cela nous a permis de mieux définir un programme URM et EURM. Avant, nous considérions qu'un programme URM et EURM n'avait finalement aucune différence (en termes de code pur) avec cette factorisation nous avons compris que la principalement différence entre un programme URM et EURM n'est pas la façon dont il exécute les différentes interactions nécessaires pour qu'il fonctionne mais plutôt son étendu de commande disponible. En effet, un programme se définit avant tout par la liste des commandes qu'il est capable de comprendre et exécuter.

Enfin, grâce à cette pré-soutenance, nous avons pu corriger des problèmes dans nos constructeurs et certains manquements sur la vérification des paramètres en entrée.

# Bilan

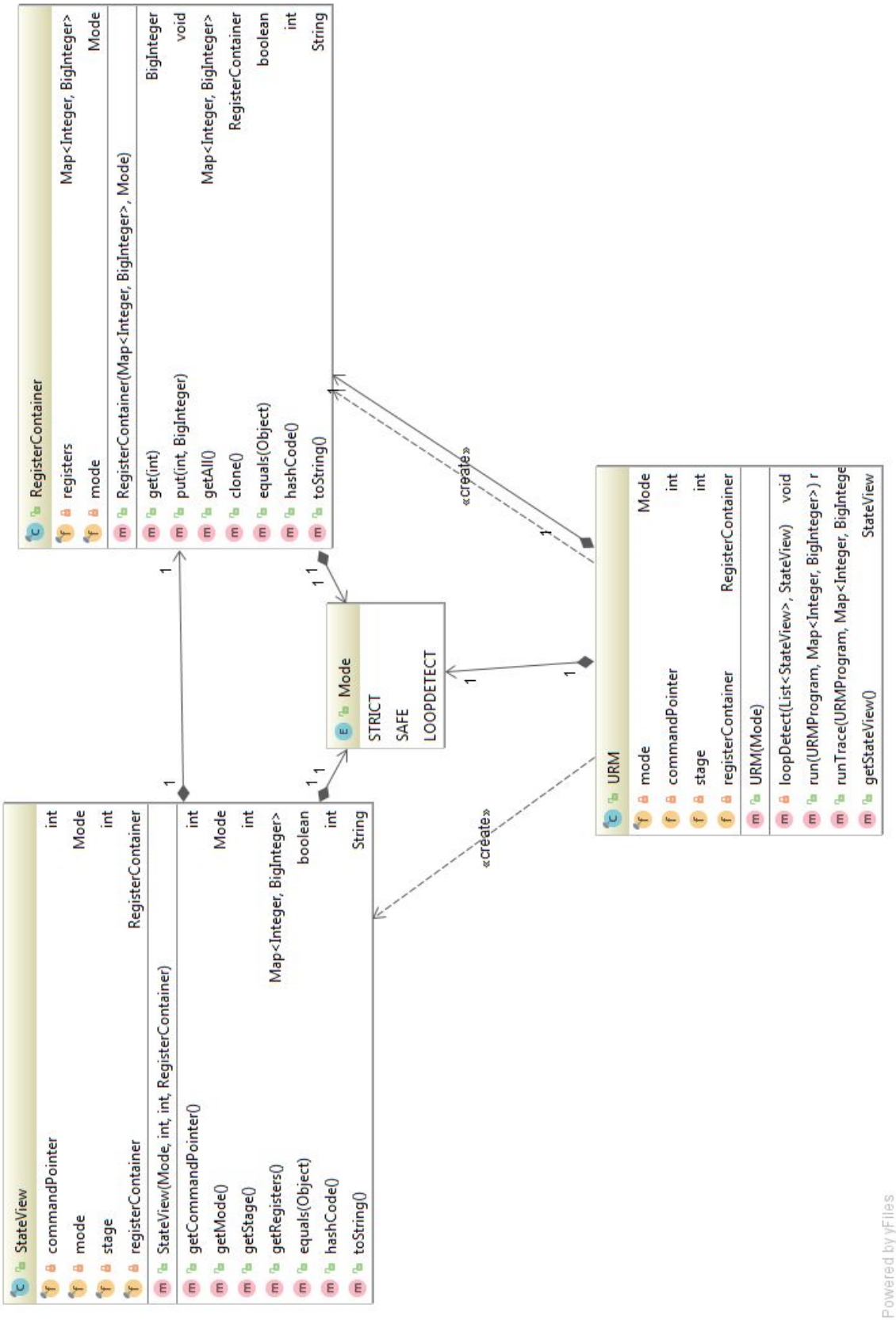
Au cours de ce projet, de nombreuses modifications ont dû être apportées au fil du temps. Nous avons donc dû rester actifs. Le travail en binôme a été extrêmement intéressant, il nous a poussés à argumenter pour défendre nos idées d'implémentations. Enfin, la soutenance bêta nous a permis d'améliorer la qualité générale de notre projet et nous a permis d'apprendre de nouvelles façons de concevoir un problème.

RASQUIER Vincent
ROUGIER Adrien

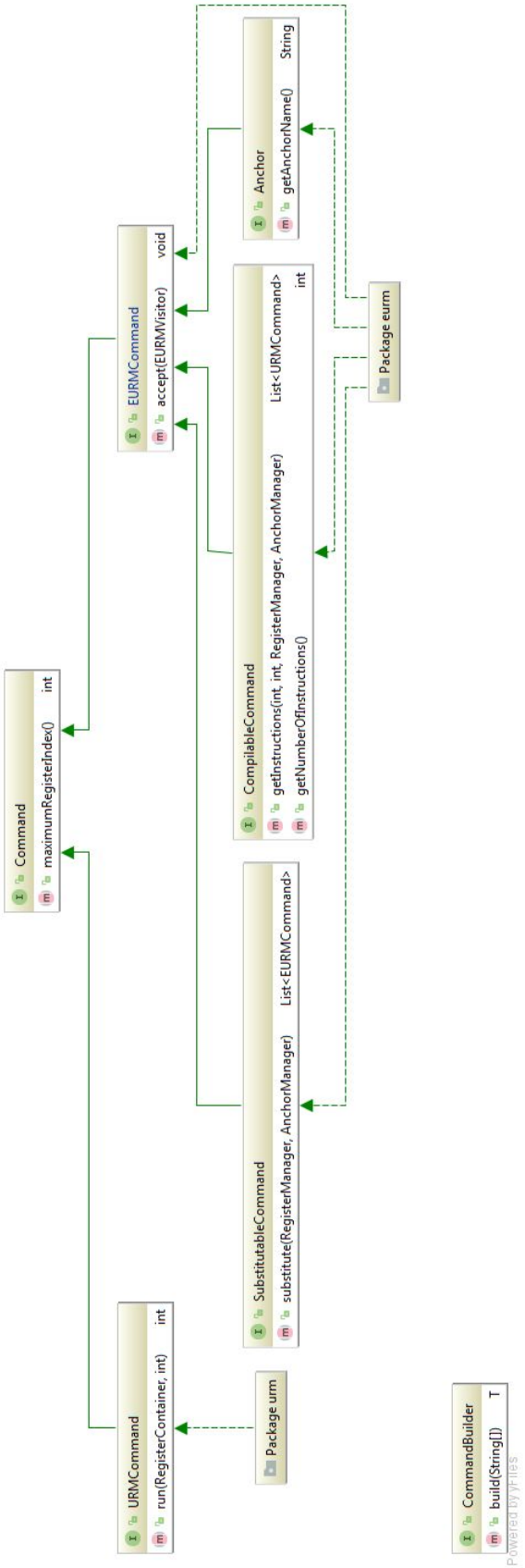


# UML

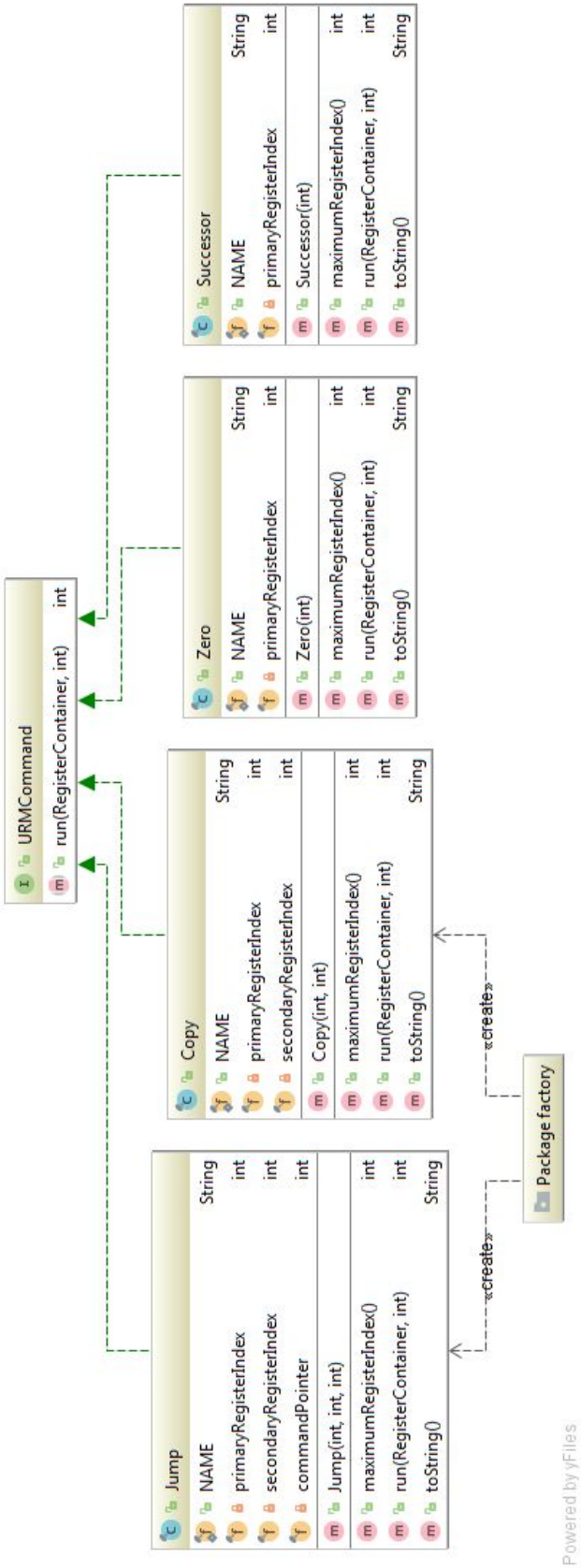
Les parties suivantes auront pour but de vous présenter nos diagrammes UML finaux.
Vous pourrez retrouver ces diagrammes sous forme png dans les dossiers des packages.


RegisterContainer
registers Map<Integer, BigInteger>
mode Mode
RegisterContainer(Map<Integer, BigInteger>, Mode)
get(int) BigInteger
put(int, BigInteger) void
getAll() Map<Integer, BigInteger>
clone() RegisterContainer
equals(Object) boolean
hashCode() int
toString() String
Mode
STRICT
SAFE
LOOPDETECT
StateView
commandPointer int
mode Mode
stage int
registerContainer RegisterContainer
StateView(Mode, int, int, RegisterContainer)
getCommandPointer() int
getMode() Mode
getStage() int
getRegisters() Map<Integer, BigInteger>
equals(Object) boolean
hashCode() int
toString() String
URM
mode Mode
commandPointer int
stage int
registerContainer RegisterContainer
URM(Mode)
loopDetect(List<StateView>, StateView) void
run(URMProgram, Map<Integer, BigInteger>) r
runTrace(URMProgram, Map<Integer, BigInteger
getStateView() StateView
«create»
«create»
1
1
1
1
1
1
1
1
Powered by yFiles

fr.umlv.urm



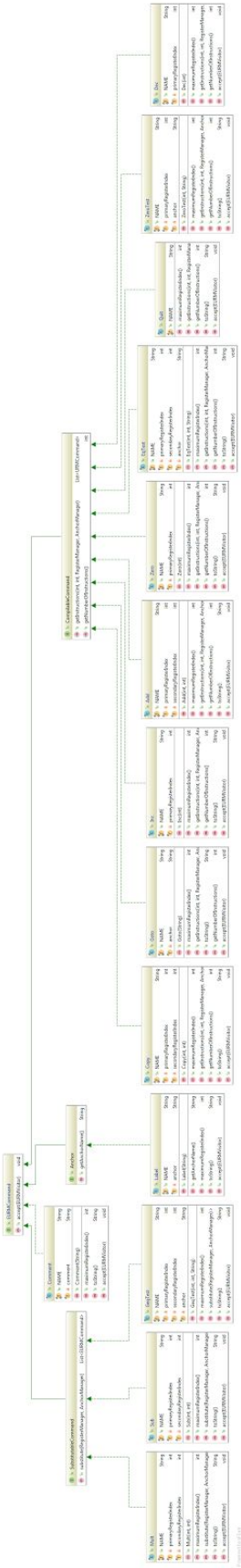

fr.umlv.urm.command




fr.umlv.urm.command.urm

RASQUIER Vincent
ROUGIER Adrien

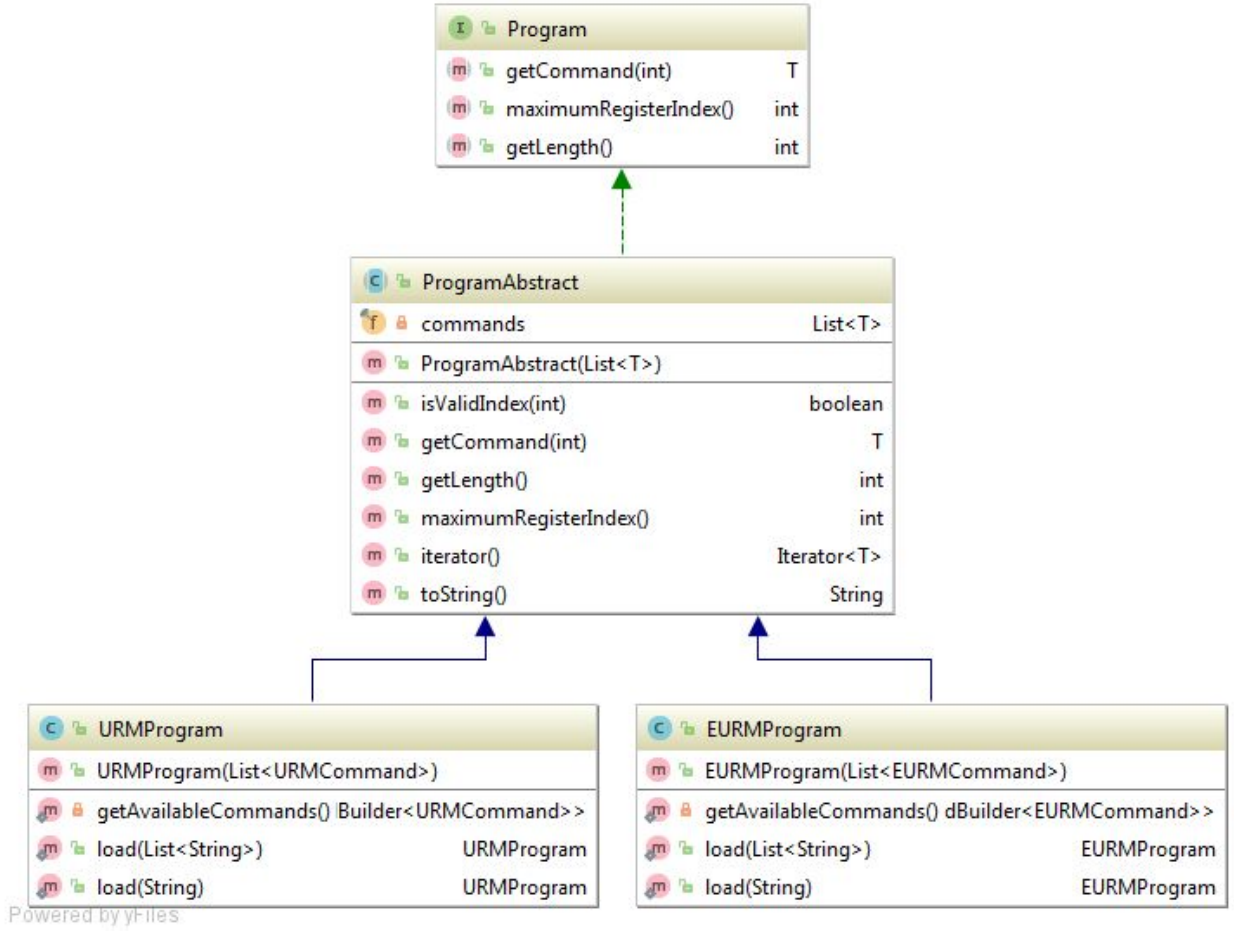




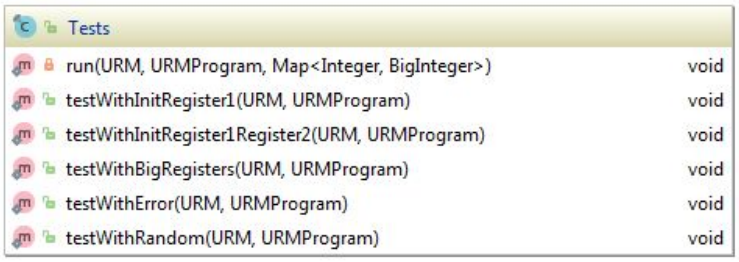
fr.umlv.urm.command.eurm

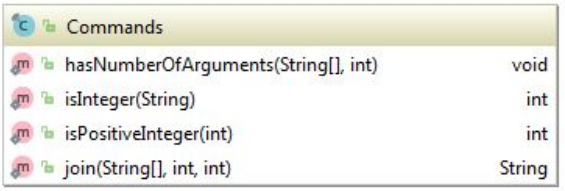
ESIPE MLV

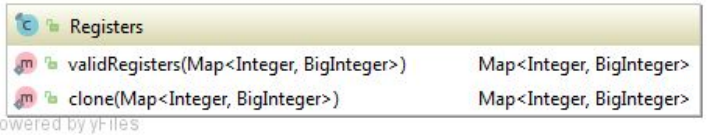
Enseignants : DURIS Etienne
VIALETTE Stéphane

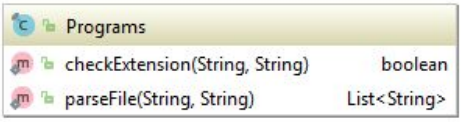



fr.umlv.urm.program

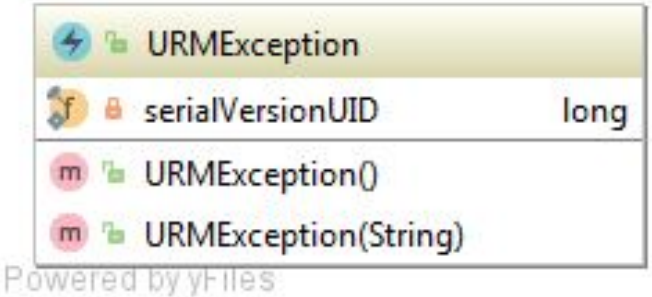


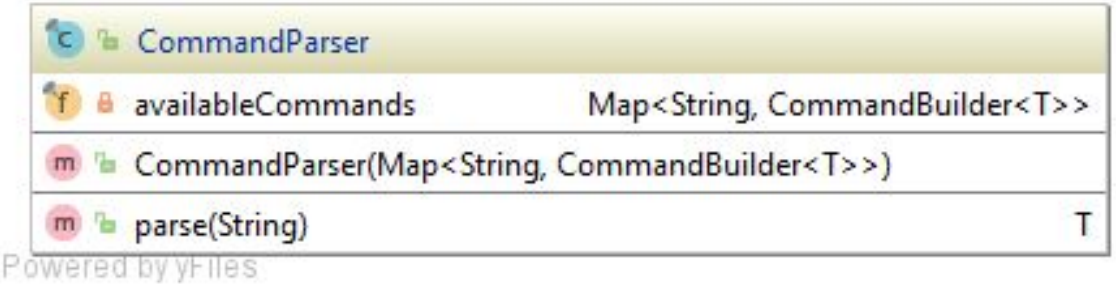



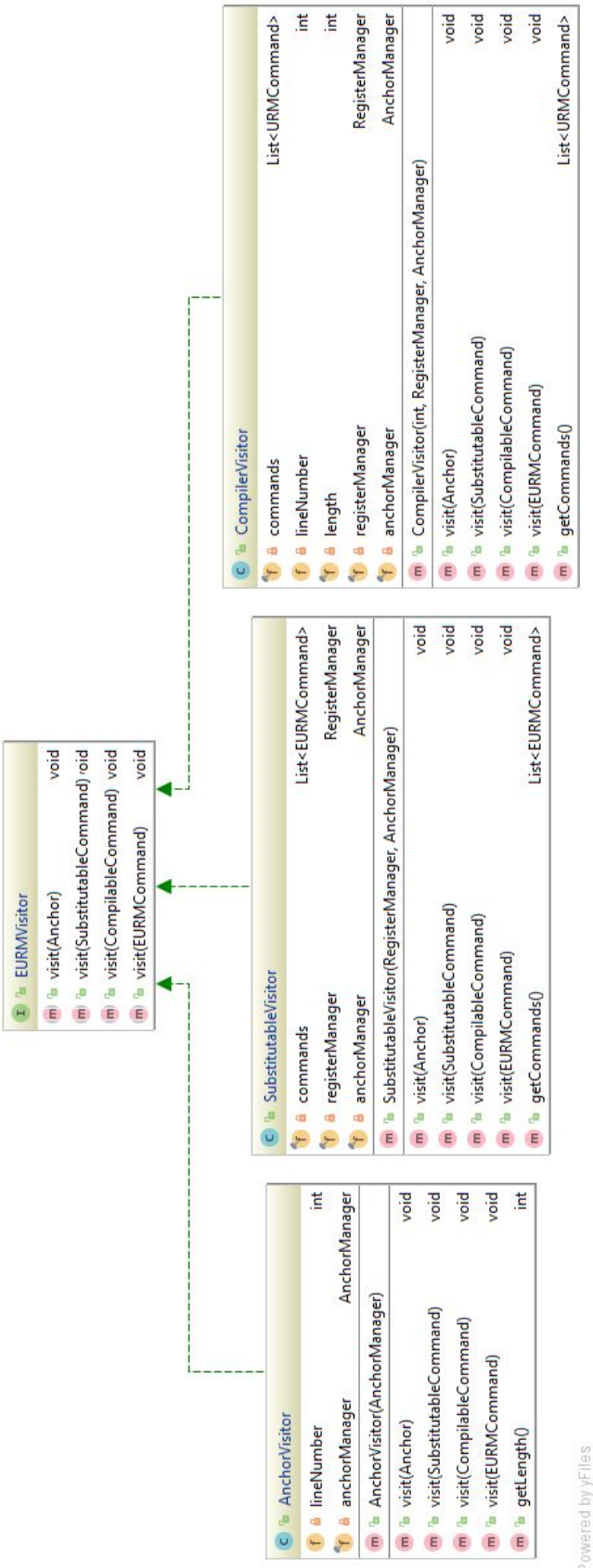

fr.umlv.urm.utilities



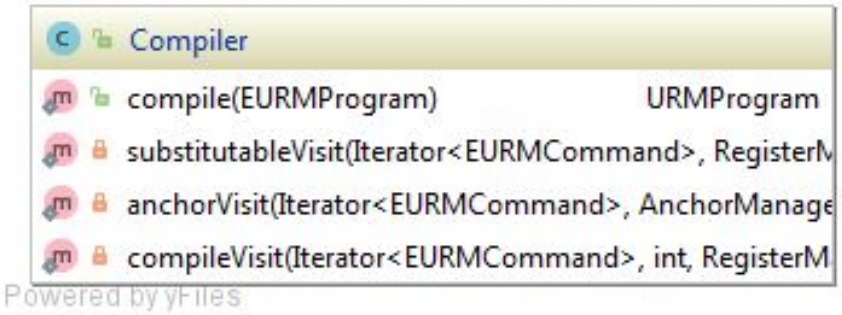

fr.umlv.urm.exception



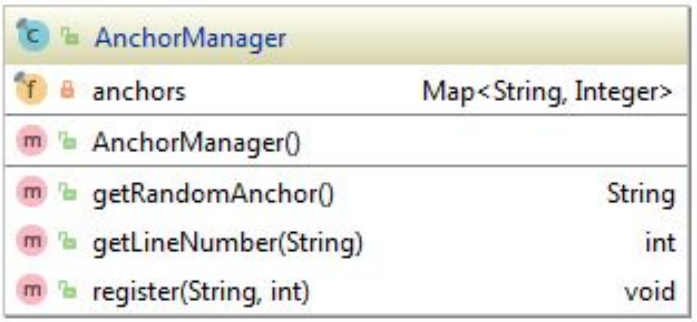

fr.umlv.urm.parser

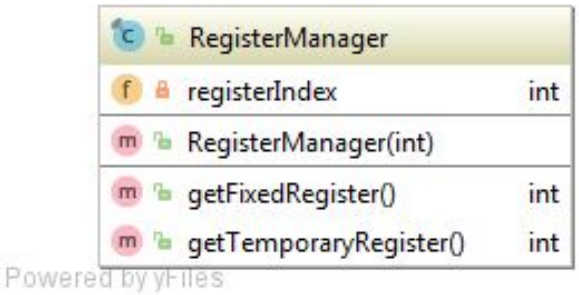



fr.umlv.urm.visitor




fr.umlv.urm.compiler




fr.umlv.urm.manager