IFSIC
INSTITUT DE FORMATION SUPÉRIEURE en INFORMATIQUE et COMMUNICATION

UNIVERSITÉ DE RENNES 1

Rapport final du projet de Master 1 Informatique

« animation d'algorithmes »

AnimAlgo

E. Abgrall, L. Aissi, S. André, J. Audo, S. Bolo, E. Caruyer, D. Hardy, T. Houdayer, K. Huguenin, N. Vigot et F. Wang

Encadré par O. Ridoux

An algorithm must be seen to be believed.

Donald Ervin Knuth

Table des matières

Préface

Un programme n'est qu'un texte, mais le processus qui l'exécute réalise par là la fonction qui est la sémantique du programme. Ces trois objets, le programme, le processus et la fonction, sont au centre de la pensée informatique, mais ils ne sont pas toujours pensés ensemble :

- le processus est le plus souvent pensé à la conception du langage de programmation et se matérialise sous la forme de machines virtuelles, de compilateurs, ou d'interpréteurs dont le programmeur standard ne connaît pas forcément le détail. Par contre celui-ci doit connaître la sémantique opérationnelle du langage de programmation qu'il utilise.
- le programme est l'unique objectif du programmeur débutant, qui a du mal à le distinguer des deux autres objets.
- la fonction qui est la sémantique du programme est souvent juste « ce que fait le programme » alors que si on la pense avant d'écrire le programme elle devient une spécification de « ce que doit faire le programme ».

L'objet des études d'informatique est pour ce qui concerne l'apprentissage de la programmation, de clarifier les relations entre programme, fonction et processus d'exécution. Le programmeur averti, qui commence par la spécification, aime voir fonctionner ces trois objets sous la forme de tests. Il en tire de l'information sous un mode expérimental.

L'animation d'algorithmes propose une réponse à ces deux types de programmeurs. Au débutant, elle offre la vision du programme (évidemment) et du processus, mais aussi de la fonction si elle est définie avant le programme. A l'expert, elle offre ce qu'offre la simulation en général : le fonctionnement en milieu contrôlé, la sécurité, la possibilité de brancher des observateurs.

Il existe de nombreux exemples où un algorithme est animé pour en expliquer le fonctionnement, mais souvent le système n'est pas un animateur d'algorithmes, mais l'animateur d'un algorithme. Parce qu'il sont assez spectaculaires, les algorithmes de tri ont souvent été traités de cette façon (exemple : interstice). Un aspect saillant de ce projet est qu'il s'agit vraiment de réaliser un système d'animation d'algorithmes. Ainsi, l'algorithme est une entrée du système, que l'utilisateur peut modifier, pour voir, pour tester des hypothèses, pour mieux comprendre ce qui se passe si... ?

Il existe aussi des systèmes d'animation d'algorithmes qui sont en fait des systèmes d'insertion d'ordres d'animation dans des algorithmes écrits dans un langage de programmation standard. Un autre aspect saillant de ce projet est que l'algorithme doit être instrumenté pour animation d'une façon qui peut former une expression enregistrable indépendamment du programme. L'utilisation par ces systèmes d'un langage d'expressions d'algorithmes qui est en fait un langage de programmation concret fait que la sémantique du langage et de l'animation reste difficilement accessible pour le programmeur débutant. Il vaut mieux utiliser un langage idéalisé dont la sémantique est simple, et dont il est facile de se convaincre que la mise en œuvre est correcte.

Olivier Ridoux

Introduction

Dans un but pédagogique, on a souvent recours à des exemples illustrés, et de façon générale, la représentation des objets est souvent une aide précieuse pour en comprendre le fonctionnement. L'algorithmique connaît aussi ce besoin et l'animation d'algorithme est la méthode qui permet de visualiser les étapes d'un calcul : variables, structures de données ... C'est ce genre d'outil que nous avons développé dans un projet de Master 1re année. Ce projet était proposé par M. Ridoux qui s'est donné le rôle de « client ».

Par conséquent, notre application est un outil complémentaire à l'enseignement de l'algorithmique, en fournissant une représentation graphique des objets manipulés par un algorithme lors de chaque pas de son exécution. Notre application peut être vue également comme un outil d'expérimentation et de mise au point d'algorithmes.

Un état de l'art nous a permis de définir avec le client les objectifs du projet, en particulier de choisir les structures de données à implémenter et sous quelle forme les représenter, ainsi que les structures de contrôle de notre langage algorithmique. Nous nous sommes basés sur une recherche des outils d'animation d'algorithmes existants, ainsi que sur une étude plus personnelle des structures de données mises en oeuvre en algorithmique.

Parallèlement à ces décisions sur la finalité de notre projet, nous avons spécifié les différentes parties de l'application et leurs interactions afin de réaliser au mieux les attentes du client. Pour une version 0 de ce système nous avons décidé de limiter les structures de contrôle du langage d'algorithme à des choses extrêmement simples (affectation, séquence, itération et conditionnelles) afin de développer de façon plus avancée l'animation et le protocole d'utilisation. En contrepartie de la simplicité des structures de contrôle, nous avons développé des structures de données plus complexes que ce que l'on trouve d'ordinaire dans les langages de programmation.

L'architecture logicielle de ce système repose sur une utilisation intense de patrons de conception qui permettent de le rendre facilement modifiable. De façon délibérée, certains services manquent dans ce qui n'est qu'une version 0. L'architecture logicielle proposée permet d'envisager sérieusement ses développements futures.

Nous avons opté pour une méthode de développement agile permettant au client de piloter le projet au cours des intégrations successives, en définissant les fonctionnalités prioritaires. Nous avons utilisé au maximum des outils et technologies standards afin de faciliter la reprise du projet.

Le premier chapitre est consacré à l'analyse des besoins selon les attentes du client. Le second chapitre dresse l'ensemble des fonctionnalités offertes à l'utilisateur. Le troisième chapitre présente une spécification plus précise de l'application ainsi qu'un historique de développement. Enfin le dernier chapitre porte sur la validation de notre réalisation.

Chapitre 1

Analyse des besoins

Dans cette partie, nous aborderons les spécifications du client, ce qu'il attend de nouveau en termes d'animation d'algorithmes, sa vision de l'application, puis nous expliquerons les premiers choix technologiques qui en découlent. Nous verrons ensuite quelques exemples représentatifs de ce qui existe déjà dans le domaine de l'animation d'algorithmes, ce qui nous a permis de mieux définir les fonctionnalités de notre application.

1.1 Les attentes du client

Le client souhaite un concept nouveau autour de l'animation. L'utilisateur final ne doit pas être passif devant une animation mais acteur. Il doit pouvoir faire évoluer à son goût l'animation, modifier les données et leurs visualisations. D'autre part, il doit pouvoir saisir ses propres algorithmes pour illustrer leur fonctionnement et échanger des algorithmes avec d'autres utilisateurs. Le client a proposé comme challenge de traiter dans l'application d'animation une sélection d'algorithmes tirés d'un livre informatique. Nous détaillerons cet objectif, puis nous verrons les attentes du client qui amèneront à quelques choix technologiques.

1.1.1 Le challenge

The New Turing Omnibus de A.K Dewdney [1] est le principal composant de notre challenge. Ce livre présente une multitude d'algorithmes : il passe en effet en revue des algorithmes numériques, comme le calcul du plus grand diviseur commun par la méthode d'Euclide ou la recherche de racines, des algorithmes graphiques ou la réprésentation des données à un rôle important (Tours de Hanoï, Le Jeu de la Vie) ou encore des algorithmes manipulant des structures de données classiques telles que les graphes et les arbres. Il se présente donc comme un horizon autant pour les structures de données à implémenter que pour l'expressivité du langage d'algorithme ; l'objectif que nous nous fixons est de réaliser une application capable d'animer un maximum d'algorithmes présents dans ce livre.

1.1.2 Les attentes et spécifications

Le client souhaite une application aux capacités d'animation importantes. Elle doit être en mesure d'animer des objets de types simples (entier, booléen...), ainsi que des structures de données classiques (paire, liste, tableau) ou plus élaborées (ensembles par exemple). Pour chaque type de données, plusieurs possibilités d'animation doivent être offertes à l'utilisateur. En contrepartie, les structures de contrôles seront délibérément simples, réduites à la conditionnelle, la séquence et l'itération.

L'utilisateur doit être en mesure de faire différentes expériences avec un même algorithme pour mieux en comprendre ou illustrer son fonctionnement. Pour ce faire, il doit saisir un algorithme à l'aide d'un langage algorithmique simple, sélectionner les éléments qu'il souhaite visualiser et leur

représentation au cours de l'animation et enfin choisir le jeu de données d'entrée avant l'activation de l'animation.

Pendant l'animation, il doit avoir la possibilité de faire évoluer l'algorithme de façon continue ou pas-à-pas, faire une pause, revenir en arrière et évidement l'arrêter tout en gardant la possibilité de la relancer du début, après avoir effectué les modifications qu'il souhaite. Dans toutes ces opérations, l'algorithme est un paramètre du système saisi par l'utilisateur : c'est en cela que l'application projetée diffère des systèmes d'animation d'algorithmes existants.

Le client souhaite que son logiciel soit disponible sur internet sous la forme d'une application Web. Elle doit utiliser au mieux les ressources de l'utilisateur en s'exécutant au maximum chez celui-ci. Enfin, elle doit offrir la possibilité de sauvegarde et d'échange d'algorithmes entre les différents utilisateurs via internet.

Le client souhaite également que son application soit évolutive au niveau des structures de données proposées et des algorithmes qu'elle peut animer. Par ailleurs, elle doit pouvoir offrir un accès direct à une animation sauvegardée, dans le but pédagogique d'exhiber un cas particulier pour un algorithme par exemple, cet outil jouant ainsi un rôle de « polycopié dynamique ».

1.1.3 Quelques décisions technologiques

Pour répondre au mieux aux spécifications du client, nous avons choisi de monter un serveur fonctionnant sous Linux utilisant la plateforme de développement Struts (que nous détaillerons par la suite) et mettant à disposition une base de données pour la sauvegarde et l'échange des algorithmes. Pour utiliser au mieux les ressources des utilisateurs, l'application est une *Applet* Java. Cette solution impose une contrainte au niveau de l'utilisateur final car il doit avoir préalablement installé une *Java Runtime Environment* (JRE) pour exécuter l'*Applet* dans son navigateur internet. Nous utilisons la version 1.5 qui nous permet de profiter des fonctionnalités offertes par Java5 (généricité, types énumérés...) Nous utilisons également des bibliothèques standard qui simplifient l'écriture du code (par exemple : utilisation de Swing pour la représentation graphique, JDBC pour la base de données...)

1.2 État de l'art

Nous présentons dans cette partie quelques outils d'animation d'algorithmes, en nous attachant à souligner les points intéressants de chacun, et à détecter les parties pour lesquelles on cherchera à apporter une meilleure solution. On peut ainsi proposer au client des solutions originales et mieux cerner avec lui ce que dénote « animation d'algorithmes ».

1.2.1 AGAT

AGAT (pour *Another Graphic Animation Tool*) [2] est une librairie permettant de visualiser, pendant son exécution, les données d'un programme écrit en C ou en FORTRAN. Elle fournit des fonctions (par exemple `agatSendDouble` ou `agatSendInt`) pour communiquer avec le serveur d'animation d'AGAT.

La description de l'animation se fait dans un fichier distinct du fichier programme, écrit dans un langage dédié. Il est possible de choisir la manière dont une valeur sera affichée, parmi des représentations sous formes graphiques (diagrammes, nuages de points, etc.). On peut également afficher la valeur d'une donnée, la moyenne des valeurs prises par une variable depuis le début, ou encore afficher les valeurs successives d'une variable, pour voir son évolution. AGAT utilise la notion de flux pour décrire les données à afficher ; l'utilisateur associe un flux à chaque variable qu'il veut visualiser, et il y a une fenêtre graphique par flux au moment de l'exécution.

En ce qui concerne l'animation d'algorithmes, on peut regretter qu'AGAT ne permette d'animer que des objets de type numérique (entiers ou réels) ; par ailleurs, la séparation entre la spécification de l'animation et la programmation de l'algorithme est somme toute relative, le code du programme devant être modifié pour obtenir une nouvelle animation.

1.2.2 JAWAA

JAWAA (pour *JAva and Web based Algorithm Animation*) [3] est un outil d'animation d'algorithmes, ou plus précisément d'animation de structures de données. C'est une application Web qui permet d'interpréter une animation décrite dans un langage de script, pour représenter des objets de types très variés. On peut par exemple représenter un graphe, et montrer un parcours de celui-ci très aisément.

Au niveau technologique, JAWAA est programmé en Java, et génère une *Applet* pour chaque animation demandée. La démarche pour animer un algorithme est alors la suivante : on exécute l'algorithme (dans lequel on a ajouté les primitives pour générer le script d'animation JAWAA), on transmet ce script à l'interpréteur, qui génère une *Applet* qui permet de visualiser l'animation correspondante. En ce sens, l'animation et l'exécution du programme sont désynchronisées, ce qui rend l'interaction limitée au cours de l'animation.

Cependant, on notera que l'interface offre une grande abstraction de la représentation, en particulier pour les structures de données complexes : l'utilisateur n'a pas à se soucier des détails de représentation (emplacement dans l'*Applet* d'animation, type d'animation...)

1.2.3)i(interstices

Interstices (interstices.info) est un site qui propose des articles de découverte dans le domaine des Sciences et Technologies de l'Information et de la Communication (STIC). Ce n'est pas à proprement parler un outil d'animation d'algorithmes, mais il contient quelques pages dédiées à la description d'algorithmes spécifiques illustrés par des animations.

Ainsi il existe une page qui présente différents algorithmes de tri animés dans une *Applet*. On a accès à quelques options sur cet exemple, comme le choix de la rapidité de l'animation, ou d'une exécution pas-à-pas. L'animation s'adapte à l'algorithme de tri sélectionné, par exemple le tri a bulles insiste sur la comparaison de deux éléments consécutifs, alors que le tri par insertion met en évidence la recherche de l'endroit où insérer un élément.

Un autre article sur le site présente deux algorithmes classiques sur les graphes : l'algorithme de recherche du plus court chemin à origine unique de Dijsktra, ainsi que l'algorithme de calcul de la fermeture transitive de Roy-Warshall. Ils sont illustrés chacun par une animation Flash, qui a été programmée une fois pour toutes.

Ces animations ont le point commun d'être graphiquement très soignées, et chacune est bien adaptée à l'algorithme qu'elle illustre. En revanche, elles n'offrent pas de modularité, et proposent peu d'interaction avec l'utilisateur : il n'est pas possible par exemple de modifier un algorithme pour voir les différents comportements. Si un nouvel algorithme doit être animé, il est nécessaire de reprendre le développement de l'animation à zéro.

Enfin, on peut remarquer que ce n'est pas un hasard de trouver des animations d'algorithmes sur un site qui a une visée didactique : cet aspect pédagogique est naturellement une des motivations majeures de notre projet.

1.2.4 OGRE

OGRE (pour *Object-oriented GRaphicalEnvironment*) est un logiciel de visualisation en trois dimensions. Il sert de support à des étudiants en informatique et permet d'observer l'état d'un programme lors de son exécution. L'accent est mis sur la visualisation de l'état de la mémoire, mais on peut tout de même observer les variables. Le programme écrit en C++ doit être court et rester assez simple (pas d'utilisation de processus ou de bibliothèques)

Le logiciel est écrit en Java et utilise la bibliothèque graphique Java3D. L'utilisateur charge ou écrit un programme en C++. Le code est analysé syntaxiquement et l'environnement graphique représentant la mémoire est affiché. On peut alors naviguer dans cet environnement a l'aide de la souris ou du clavier. Les objets apparaissent, disparaissent et se modifient au fur et à mesure que l'utilisateur exécute le programme. Il est possible d'exécuter le programme pas à pas, de revenir à l'état précédent...

Parmi les applications que nous avons recensé, Ogre est certainement l'application qui se rapproche le plus de ce que l'on souhaite développer. Il a été créé dans un but pédagogique et permet de visualiser l'état de la mémoire lors d'une exécution pas à pas. Il existe quand même quelques différences notables : il n'est pas possible de sélectionner les variables que l'on veut visualiser, ni la manière dont on veut les afficher. De plus la sémantique du langage est celle d'un langage particulièrement complexe.

Chapitre 2

Spécifications pour l'utilisateur

Dans ce chapitre nous présentons de manière exhaustive les possibilités que nous avons décidé d'offrir à l'utilisateur, en répondant aux questions suivantes : *Quel langage utilise-t-il pour saisir son algorithme ? Comment saisit-il les données d'entrée ainsi que les options d'animation ? Quel est l'aspect général de l'interface ? Comment utilise-t-il cette interface ? Que peut-il trouver comme informations sur le site Web ?*

2.1 Le langage

La saisie des algorithmes à animer se fait de manière textuelle. On définit dans cette partie un langage de spécification d'algorithmes à mi-chemin entre celui employé dans *The New Turing Omnibus* [1] (purement algorithmique), et C ou Java (impératifs) pour obtenir un langage proche du langage *While* [4].

2.1.1 Le langage d'algorithmes

La spécification du langage se fait à l'aide de la grammaire 1. Cependant, la grammaire utilisée pour l'implémentation de l'analyseur syntaxique, bien qu'elle génère le même langage, ne comporte pas rigoureusement les mêmes règles de dérivation pour des problèmes d'implémentation (celle-ci sera détaillée dans la partie 3.4.2, page 30).

Un algorithme peut être découpé en trois sections représentées par les non-terminaux *header*, *declarations* et *instructions*. Celles-ci correspondent respectivement à l'en-tête, spécifiant les paramètres ou données de l'algorithme, la déclaration des variables locales, de leurs types et de leurs valeurs initiales et enfin le corps contenant les instructions effectuées par l'algorithme.

En-tête Il contient le nom de l'algorithme ainsi que les paramètres (et leurs types) pris en entrée par celui-ci

Types Les types du langage algorithmique sont définis de manière inductive. Un type est soit un type de base (**int**, **bool**, **char**...) soit un type composé défini à l'aide de constructeur de type (**[ ]** pour les tableaux, **queue** pour les files, **stack** pour les piles, **list** pour les listes, et **set** pour les ensembles) avec une syntaxe proche de celle de Caml.

Expressions La grammaire du langage ne distingue pas les expressions suivant leur type. Cette vérification sera faite lors de l'analyse *sémantique* et sera détaillée dans la partie 3.4.2. Une expression peut se décliner sous trois formes : une constante, un appel de fonction ou une (respectivement deux) expression(s) composée(s) par un opérateur unaire (respectivement binaire).

Instructions Une instruction se décline également sous trois formes : une affectation, une structure de contrôle commandée par une ou plusieurs expressions ou un appel de procédure.

Grammaire 1 Langage Algorithmique

Algorithm	→	**algo** *header declarations instructions* **end**
header	→	**ident (** $(parameters)^{0/1}$ **)**
parameters	→	*parameter* $(,\ parameter)^*$
parameter	→	*type* **ident**
declarations	→	$(declaration)^*$
declaration	→	*type* **ident** $=$ *exp* $(,$ **ident** $= exp)^*$
instructions	→	$(\ instruction\ )^+$
instruction	→	**ident ([** *exp* **])*** $:=$ *exp* **;**
	|	**if** *exp* **{** *instructions* **}** $(\ $**else { ** *instructions* **}** $\ )^{0/1}$
	|	**while** *exp* **{** *instructions* **}**
	|	**for ident** $=$ *exp* **to** *exp* **step** *exp* **{** *instructions* **}**
	|	**for ident {***instructions* **}**
	|	**foreach ident in** *exp* **{** *instructions* **}**
	|	**ident (** *exp_list* **) ;**
type	→	*base_type* $(compose\_type)^*$
base_type	→	**int** | **bool** | **string** | **char**
compose_type	→	**bintree** | **set** | **list** | **stack** | **queue** | **[** *exp* **]**
exp	→	**(** *exp* **)**
	|	$exp + exp \mid exp - exp \mid exp * exp \mid exp\ /\ exp \mid exp\ \%\ exp \mid -\ exp$
	|	$exp < exp \mid exp > exp \mid exp <= exp \mid exp >= exp$
	|	$exp = exp \mid exp\ !=\ exp$
	|	*exp* **and** *exp* | *exp* **or** *exp* | **not** *exp*
	|	*value*
value	→	**integer** | **true** | **false** | **nil** | **"** **string** **"** | **'** **char** **'**
	|	$\{\ exp\_list\ \} \mid [exp\_list\ ] \mid <\mid exp\_list \mid > \mid [\![\ exp\_list\ ]\!]$
	|	**ident ([** *exp* **])***
	|	**ident (** *exp_list* **)**
exp_list	→	$(\ exp\ (,\ exp)^*\ )^{0/1}$

Priorité des opérateurs La priorité des opérateurs retenue est celle de Java et est définie dans le tableau 2.1. Celle-ci est classée de façon décroissante (le plus prioritaire en haut). Les opérateurs ayant la même priorité sont regroupés dans une même cellule. Tous les opérateurs ont une associativité gauche ($a + b + c$ est équivalent à $(a + b) + c$).

Passage par valeur Le objets retournés par les différentes fonctions retournent toujours de nouveaux objets. Par exemple push(s, 2) retourne une nouvelle pile, qui n'a aucun lien avec l'objet de départ s. L'affectation se fait toujours dans des variables et jamais directement dans le détail des structures de données.

2.1.2 Primitives du langage

Le langage d'algorithmes choisi autorise l'utilisation de *fonctions* et de *procédures*, cependant il n'est pas possible à l'utilisateur d'en définir lui-même dans la version 0 du logiciel. Il s'agit donc uniquement de primitives du langage. On en élabore une liste pour chaque type de données en s'inspirant principalement des primitives utilisées dans *The New Turing Omnibus* [1].

On ne détaillera pas les éventuelles erreurs générées par ces fonctions et procédures. L'API

Opérateur	Description
( )	opérateurs primaires
−	moins unaire
not	non logique
*	multiplication
/	division
%	modulo
+	addition
-	soustraction
<	plus petit
<=	plus petit ou égal
>	plus grand
>=	plus grand ou égal
==	égalité
!=	différence
and	et logique
or	ou logique

TAB. 2.1 – Priorité des opérateurs

complète du langage est disponible sur le site hébergeant l'application dans un style proche de Javadoc.

entiers on définit principalement des fonctions de calcul :

- `max(int a, int b) : int` , renvoie $\max(a, b)$
- `min(int a, int b) : int` , renvoie $\min(a, b)$
- `power(int a, int b) : int` , renvoie a^b
- `random() : int`, renvoie un entier choisi alétoirement
- fonctions mathématiques usuelles...

caractères on définit des fonctions permettant de manipuler des caractères sans utiliser explicitement les codage :

- `compare(char c1, char c2) : int` , renvoie -1 si $c_1 < c_2$, 1 si $c_1 > c_2$ et 0 sinon
- `ascii(char c1) : int` , renvoie le code ascii correspondant au caractère c1
- `nextChar(char c1) : char` , renvoie le caractère suivant c1
- `previousChar(char c1): char` , renvoie le caractère précédant c1
- `...`

chaînes de caractères on définit les fonctions usuelles de composition et décomposition de chaînes :

- `compare(string s1, string s2) : int` , renvoie -1 si $s_1 < s_2$, 1 si $s_1 > s_2$ et 0 sinon (selon l'ordre lexicographique induit par l'ordre sur les caractères)
- `sconcat(string s1, string s2) : string` , renvoie la concaténation $s_1 @ s_2$
- `charat(string s, int i) : char` , renvoie le $i-$ème caractère de s
- `substring(string s, int i, int j) : string` , renvoie la sous-chaîne de s comprise entre les caractères i et j (inclus)
- `stringOfChar(char c) : string` , renvoie la chaîne formée du seul caractère c
- `length(string c) : int` , renvoie la longueur de la chaîne s
- `...`

ensembles on définit les opérations usuelles ensemblistes :

- `member('a set e, 'a x) : bool` , renvoie $x \in e$
- `size('a set e) : int` , renvoie le cardinal $|e|$ (commune à tous les types composés)
- `add('a set e, 'a x) : 'a set` , renvoie un set contitué de $e \cup \{x\}$
- `remove('a set e, 'a x) : 'a set` , renvoie un set contitué de $e \setminus \{x\}$

- `union('a set e1 , 'a set e2) : 'a set` , renvoie un set constitué de $e_1 \cup e_2$
- `inter('a set e1 , 'a set e2) : 'a set`, renvoie $e_1 \cap e_2$
- ...

arbres on définit les opérations usuelles sur les arbres :
- `cons('a bintree b1, 'a bintree b2): 'a bintree`, construit un arbre à partir de deux arbres b_1 et b_2
- `unit('a v): 'a bintree`, construit une feuille à partir d'une valeur v
- `car('a bintree b): 'a bintree`, retourne le sous-arbre gauche de b
- `cdr('a bintree b): 'a bintree`, retourne le sous-arbre droit de b
- ...

tableaux les éléments d'un tableau étant gérés comme des variables on ne définit que des constructeurs :
- `makeArray(int a,'b x) : 'b [a]` , renvoie un tableau de taille a dont tous les éléments sont égaux à x
- `makeRandomArray(int size, int max) : int [size]`, construit un tableau de taille *size* contenant des valeurs entières aléatoires comprieses entre 0 et $max - 1$
- initialisation à partir de fonctions...

listes on implémente les fonctions habituelles de manipulation de listes chaînées (constructeurs, extracteurs) :
- `head('a list l) : 'a` , renvoie la tête de liste
- `tail('a list l) : 'a list` , renvoie la queue de liste
- `cons('a list l, 'a x) : 'a list` , renvoie la liste l à laquelle on a ajouté x en tête
- `lconcat('a list l1, 'a list l2) : 'a list` , renvoie le résultat de la concaténation de l_1 et l_2
- `lmember ('a list l1, 'a x) : 'a list` , renvoie $x \in l1$
- ...

piles on implémente les *procédures* habituelles sur les piles :
- `top('a stack s) : 'a` , renvoie le sommet de pile
- `push('a stack s, 'a x) : 'a stack`, renvoie une pile égale à s à laquelle on a ajouté x
- `pop('a stack s) : 'a stack`, renvoie une pile égale à s à laquelle on a enlevé le sommet
- ...

files on implémente les *procédures* habituelles sur les files :
- `top('a queue q) : 'a` , renvoie le sommet de file
- `push('a queue s, 'a x) : 'a queue`, renvoie une file égale à q à laquelle on a ajouté x
- `pop('a queue q) : 'a queue`, renvoie une file égale à q à laquelle on a enlevé le sommet
- ...

Les programmes 1 et 2 donnent deux exemples de programmes simples écrient dans le langage algorithmique utilisant quelques unes des structures de données, structures de contrôles et primitives du langage.

2.1.3 Langage d'animation

L'interaction de l'utilisateur avec l'application ne se limite pas à la saisie de l'algorithme. En effet, l'utilisateur doit pouvoir spécifier les données de l'algorithme (valeurs des paramètres), ainsi que les paramètres de l'animation (quelles variables animer et comment).

La saisie des paramètres d'animation n'est pas faite sous forme textuelle mais à l'aide de composants graphiques (détaillés dans la partie 2.2.1) tels que des cases à cocher, menu déroulant... Pour être sauvegardés dans la base de données, les paramètres d'animation sont formatés en XML[1] (détaillé en 3.6.1). Cependant, la conversion sera effectuée par l'application et ne concerne donc pas directement l'utilisateur.

[1] format destiné à faciliter le partage de textes et d'informations structurées

Algorithme 1 Algorithme de tri à bulle

```
algo triABulle(int size, int [size] t)
int [size] tab = t
int temp = 1
bool permut = true
while permut {
    permut := false;
    for i := 0 to size(tab) - 2 step 1 {
        if (tab[i] > tab[i + 1]) {
            temp := tab[i];
            tab[i] := tab[i + 1];
            tab[i + 1] := temp;
            permut := true;
        }
    }
}
end
```

Algorithme 2 Algorithme de Roy-Warshall pour τ-minimalité

```
algo Roy(int set V, int [2] set E)
int [2] set E1 = E, E2 = E
  foreach p in V {
    foreach x in V {
      foreach y in V {
        if member( E1 , [| x , p |] ) and member( E1 , [| p , y |] ) {
          E1 := add(E1 , [| x , y |] );
          E2 := remove(E2 , [| x , y |] );
        }
      }
    }
  }
end
```

Enfin, la saisie des paramètres de l'algorithme est faite dans un langage proche de celui des expressions d'initialisation des variables locales. La seule différence étant l'absence d'identificateurs de variables et de paramètres.

2.2 De la définition de l'algorithme à son animation

On décrit dans cette section les différentes étapes de la saisie d'un algorithme (ou de la sélection dans la base de données d'un algorithme préalablement enregistré) à son animation, en passant par le choix des paramètres de l'algorithme et de l'animation. On va dans un premier temps lister ces étapes et donner leur contenu, en s'appuyant sur des captures d'écran de l'IHM (figures 2.2 à 2.6, situées en fin de chapitre). Nous verrons enfin quelles sont les contraintes de passage d'une étape à l'autre, sous la forme d'un graphe dont on expliquera la signification.

2.2.1 Animer en quatre étapes

L'animation d'un algorithme se fait en quatre étapes. On a choisi d'utiliser un onglet pour chacune de ces étapes, afin de permettre à l'utilisateur de pouvoir basculer facilement d'un point à l'autre. Un cinquième onglet correspond à une demande de sauvegarde dans une base de données de l'algorithme, du jeu de paramètres et des options d'animations.

Enfin, l'IHM comporte une zone visible à tout moment, la console, qui sert à envoyer des messages de l'application à l'utilisateur, l'informant d'une erreur (survenue lors de la compilation par exemple). Cette console peut se réduire en bas de l'*Applet*, afin de profiter de davantage d'espace pour la saisie de l'algorithme, l'animation, etc.

1re étape Saisie de l'algorithme (figure 2.2)

L'utilisateur peut dans cette étape, soit saisir un algorithme dans la zone de texte, soit en sélectionner un dans la base de données. Pour cela, il lui suffit de taper des mots clés dans le champ prévu à cet effet, et d'appuyer sur le bouton *Go*. Il a alors accès à une liste d'algorithmes correspondant à ces mots clés. Le fait de cliquer sur un algorithme de cette liste affiche la description de cet algorithme dans le champ *Description of the selected algorithm*, et le texte de l'algorithme dans la zone de saisie. Lorsque l'utilisateur est prêt, il clique sur le bouton *Compile and select Data*. Si la compilation de l'algorithme ne génère pas d'erreur, l'onglet de la deuxième étape s'affiche. Dans le cas contraire, le message d'erreur est indiqué dans la console.

2e étape Saisie des données d'entrée (figure 2.3)

Un algorithme peut éventuellement prendre des données en entrée (cf. la description du langage 2.1). L'utilisateur spécifie à cette étape les valeurs de ces paramètres. Si l'algorithme en cours d'étude est un algorithme issu de la base de données et non modifié, l'utilisateur peut également charger un jeu de données associé à cet algorithme. Pour valider et passer à la troisième étape, il doit cliquer sur le bouton *Select Options*.

3e étape Saisie des options d'animation (figure 2.4)

L'utilisateur peut sélectionner les variables qu'il souhaite observer, ainsi que la représentation graphique de celles-ci. L'algorithme est visible dans la partie gauche de l'*Applet*, correctement indenté, avec une seule instruction par ligne. L'utilisateur peut aussi sélectionner dans la base de données des options d'animation associées à cet algorithme, s'il en existe. Pour valider et passer à la quatrième étape, il doit cliquer sur le bouton *Anime it*.

4e étape Lancer l'animation (figure 2.5)

L'utilisateur peut alors lancer l'animation. Il a pour cela accès aux boutons *Play/Pause*, *Stop*, *Previous Step* et *Next Step*. Il peut aussi définir la vitesse d'exécution (en nombre de *step*/seconde). S'il est satisfait du résultat, il peut enregistrer les données de l'expérience dans la base de données. Ceci correspond à la dernière étape, à laquelle il accède en cliquant sur l'onglet *Save*.

5e étape Sauvegarde dans la base de données (figure 2.6)

L'utilisateur a la possibilité d'enregistrer tout ou partie de ce qu'il a édité, c'est-à-dire l'algorithme (si c'est un algorithme nouveau, ou s'il est issu de la base de données mais a été modifié), les options d'animation et les paramètres de l'algorithme. Il ne peut évidemment pas sauver les options d'un algorithme sans sauver cet algorithme, sauf s'il est déja présent dans la base de données. L'utilisateur peut aussi ajouter des commentaires sur l'algorithme, les options ou les données qu'il souhaite sauvegarder. Les commentaires permettront ensuite la recherche par mots-clés.

2.2.2 Définition des scenarii pour l'interface graphique

Le choix de la navigation par onglets nous conduit à imposer quelques restrictions à l'utilisateur ; en particulier, chaque onglet ne doit pas être disponible à tout moment. Par exemple, cela n'aurait aucun sens de permettre à l'utilisateur d'accéder à l'onglet « animation » si celui-ci n'a

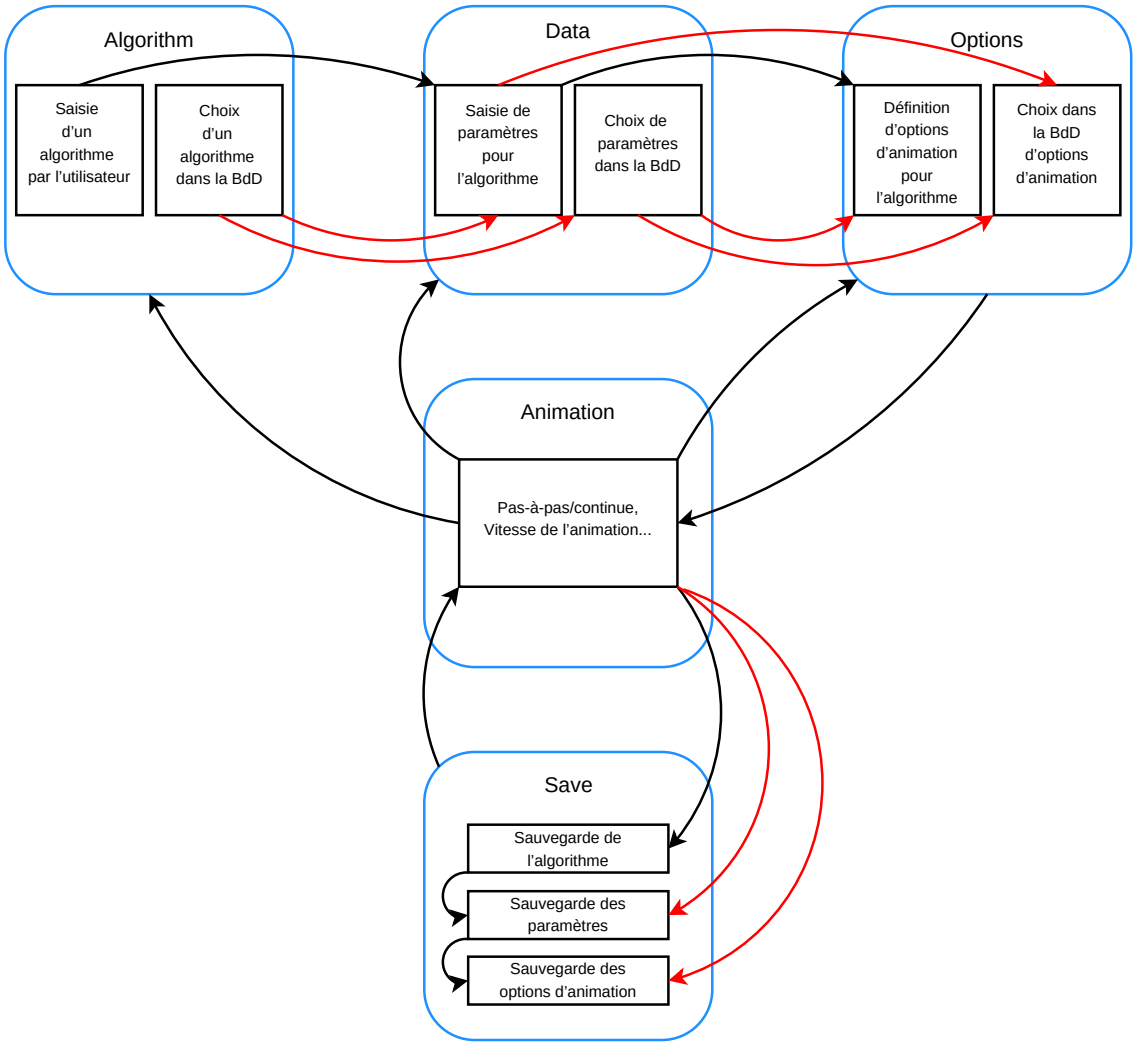


FIG. 2.1 – Ce schéma décrit les différents scénarii envisageables pour la navigation dans l'IHM ; l'utilisateur aura accès à une option i à partir d'une option j s'il existe une transition possible entre ces deux options (matérialisée par une flèche sur le schéma). Les flèches de couleur rouge imposent de plus que l'algorithme que l'on veut animer vienne de la base de données.

pas encore choisi d'algorithme à animer... Le schéma de la figure 2.1 spécifie ces contraintes de façon exhaustive, en les organisant sous la forme d'un graphe orienté (voir [5]). L'organisation de ce schéma est calquée sur la décomposition en onglets de l'IHM, et à la description des options disponibles à l'intérieur de ces onglets.

2.2.3 Représentation des types de données

Pour le rendu graphique des différents types de données, l'utilisateur aura le choix entre deux alternatives : une visualisation de l'aspect de la donnée et une visualisation plus détaillée. Nous présentons les animations possibles pour les types de bases puis pour les types composés.

Les types de bases sont les nombres entiers, les booléens, les caractères et les chaînes de caratères. Les animations pour ces types sont les suivantes :

les entiers la valeur du nombre, le signe du nombre, une barre verticale ou horizontale proportionnelle à la valeur, le caratère ASCII correspondant à la valeur du nombre.

les booléens la valeur (soit 0/1, soit true/false), un carré de couleur (soit noir/blanc).

les caractères le caractère, une chaine de caractère (consonne, voyelle, ou autre).

chaîne de caractère la chaîne elle-même (ou représentée en lettres majuscules ou minuscules) et la longueur de la chaîne.

De plus, notre langage met à disposition de l'utilisateur plusieurs types de données composées. On distingue deux types de représentations pour ces types : soit on s'intéresse à des caractéristiques globales (ex : cardinal), soit on les représente avec un grain de description plus fin. Par exemple, l'animation d'un tableau d'entiers pourra se résumer à l'animation de chacun des éléments du tableau (en invoquant une des méthodes d'animation de ses éléments).

Parmi ces structures, on distingue : **les ensembles, les tableaux, les listes, les piles et les files**. Les animations sont communes à toutes les variables de type composé. On peut :

- afficher le cardinal de la structure
- représenter la structure par une colonne (ou une ligne). La colonne (ligne) contient tous les éléments présents dans la structure en respectant l'ordre des éléments, sauf en ce qui concerne les ensembles, car ils ne sont pas ordonnés. Les élements peuvent être représentés conformément aux animations disponibles pour leur type.
- representer un tableaux à deux dimension sous forme de matrice
- representer un arbre binaire sous sa forme habituelle avec les valeurs des feuilles
- représenter la variable sous forme textuelle

2.3 Le site Web

L'application développée étant une *Applet*, il est indispensable d'avoir une page Web qui contienne cette application. Mais plus qu'une page, c'est un véritable site Web qu'il faut mettre en œuvre. Il est en effet nécessaire d'offrir à l'utilisateur une gamme de services et d'informations, afin qu'il puisse profiter au maximum de l'application.

2.3.1 Les sections principales

Dès la page d'accueil, l'utilisateur doit avoir un accès immédiat vers différentes pages, détaillées ci-dessous. Les liens vers chacune de ces sections seront regroupées dans un menu visible, situé en haut de chaque page.

Accueil Cette page est la première que le visiteur voit. Il doit pouvoir savoir en un regard en quoi consiste cette application. L'accueil doit présenter les dernières actualités, la possibilité de lancer l'application, ainsi qu'un lien vers toutes les autres pages.

Actualités Comme son nom l'indique, cette page présente toutes les dernières actualités concernant l'application développée. Dernière mise à jour, évolution, nouvelles possibilités, etc.

anim.algo C'est la page qui héberge l'application. En cliquant sur cette page, c'est l'*Applet* qui se lance. Hormis cela, seul le menu principal est visible.

Base d'algorithmes Cette page permet l'exploration des différents algorithmes déposés par les utilisateurs. Il peut y faire une recherche, ou les afficher tous.

Documentation Cette partie représente une étape indispensable pour le visiteur néophyte. Elle contient le mode d'emploi de l'application, la syntaxe du langage utilisé, les spécifications des fonctions définies, etc.

FAQ Toutes les questions fréquentes que les visiteurs poseront. Page qui devra être actualisée régulièrement afin de répondre aux attentes des utilisateurs.

À propos Cette page est faite pour ceux qui désirent en savoir un peu plus sur l'application. Qui l'a développée ? Dans quel cadre ? Comment participer au développement ?

Contact Les différents moyens de contacter l'équipe de développement, ou juste un membre. Adresse postale, mail, téléphone, forumulaire en ligne, etc.

2.3.2 Les pages annexes

Ces différentes pages contiennent des informations dites « annexes », du fait de leur intérêt moindre pour un utilisateur lambda. Elles contiennent néanmoins des services, informations pouvant être très utiles pour des utilisateurs spécifiques. Les liens pointant vers ces pages seront mis en évidence de façon plus discrète, pour ne pas perturber l'utilisateur. Ces liens seront probablement présents en bas de chaque page.

Plan du site Le visiteur pourra y trouver un plan du site, des différentes informations qui s'y trouvent, etc.

Rechercher L'utilisateur doit pouvoir faire une recherche sur le contenu du site.

Raccourcis clavier Cette page présente de manière claire les différents raccourcis claviers utilisables sur le site.

Administration Un accès vers l'espace d'administration, qui permet de modérer la base de donnée, de gérer les droits, etc.

Conformité Deux liens situés en bas de page permettent de vérifier la conformité des pages aux standards du W3C.

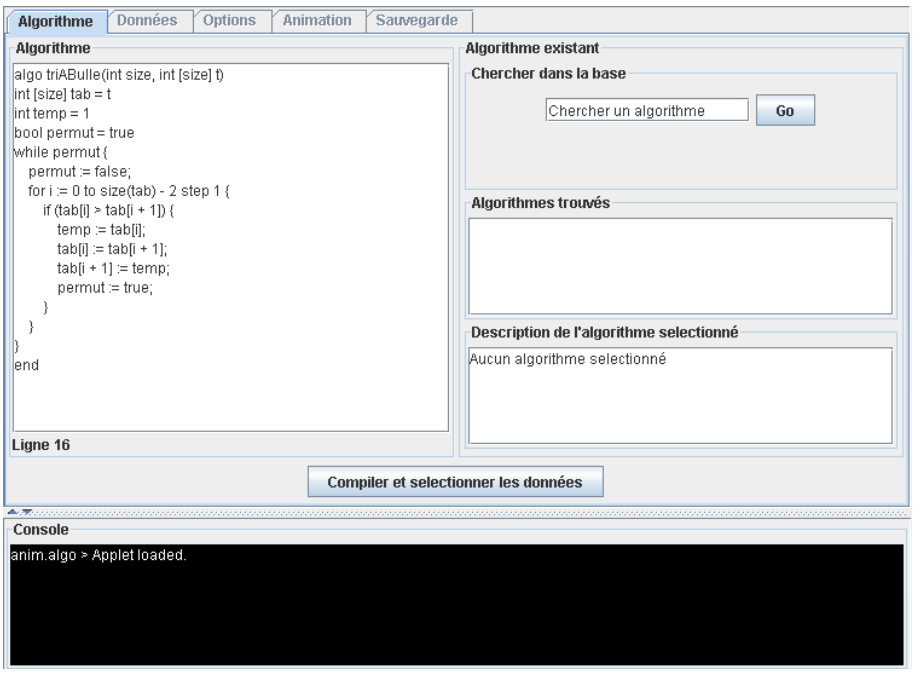


FIG. 2.2 – Capture d'écran de l'*Applet* : 1re étape

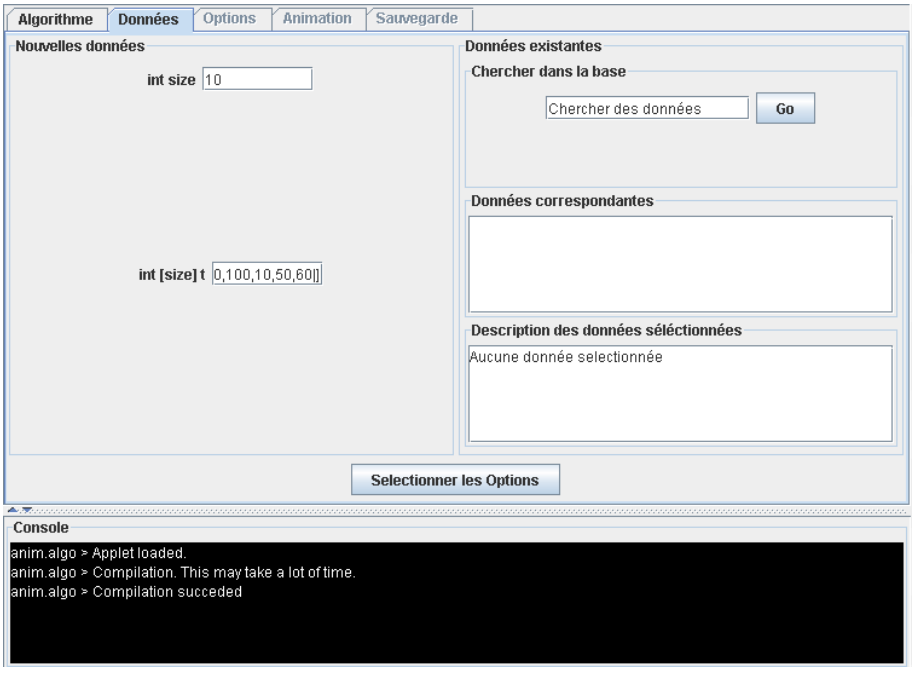


FIG. 2.3 – Capture d'écran de l'*Applet* : 2e étape

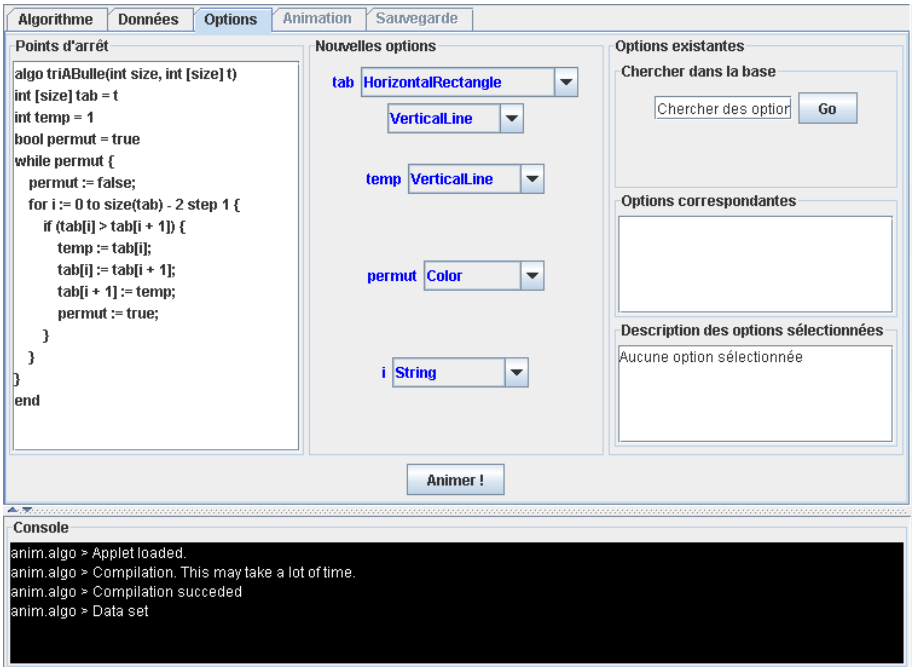


FIG. 2.4 – Capture d'écran de l'*Applet* : 3^{e}étape

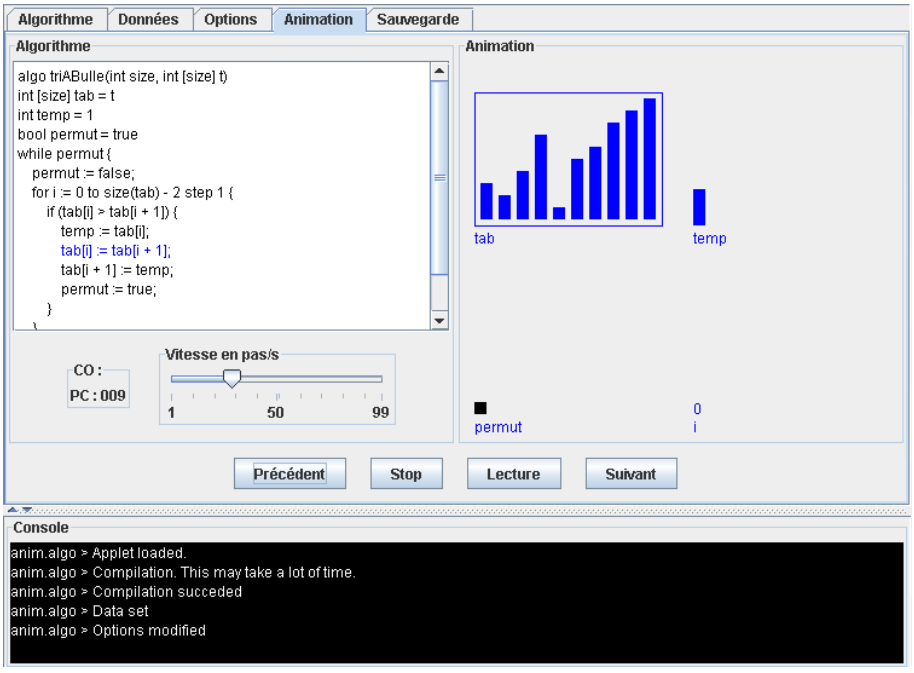


FIG. 2.5 – Capture d'écran de l'*Applet* : 4^{e}étape

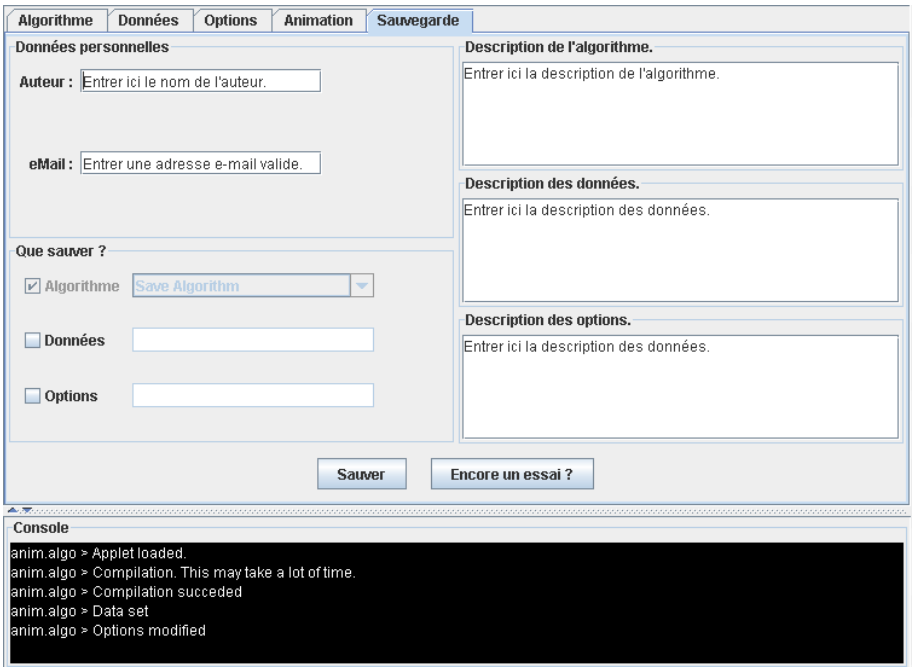


FIG. 2.6 – Capture d'écran de l'*Applet* : 5^{e}étape

Chapitre 3

Spécification de développement

On décrit dans cette partie la manière dont on répond aux spécifications présentées dans la partie 2. On présente dans un premier temps les choix technologiques et théoriques faits pour chacune des *briques* de l'application (au sens large, on ne considère pas uniquement l'*Applet*). L'utilisation des outils et l'implémentation des modèles utilisés seront détaillées dans les différentes parties de ce chapitre et à la fin de ce dernier nous verrons l'historique de développement.

3.1 Vue d'ensemble

La figure 3.1 donne une vue d'ensemble de notre application. Cela permet de voir les différents modules qui la composent et les relations (dépendance, communication ...) entre eux. On peut ainsi plus facilement voir les différentes parties de l'application. Cette vision globale sera détaillée par la suite.

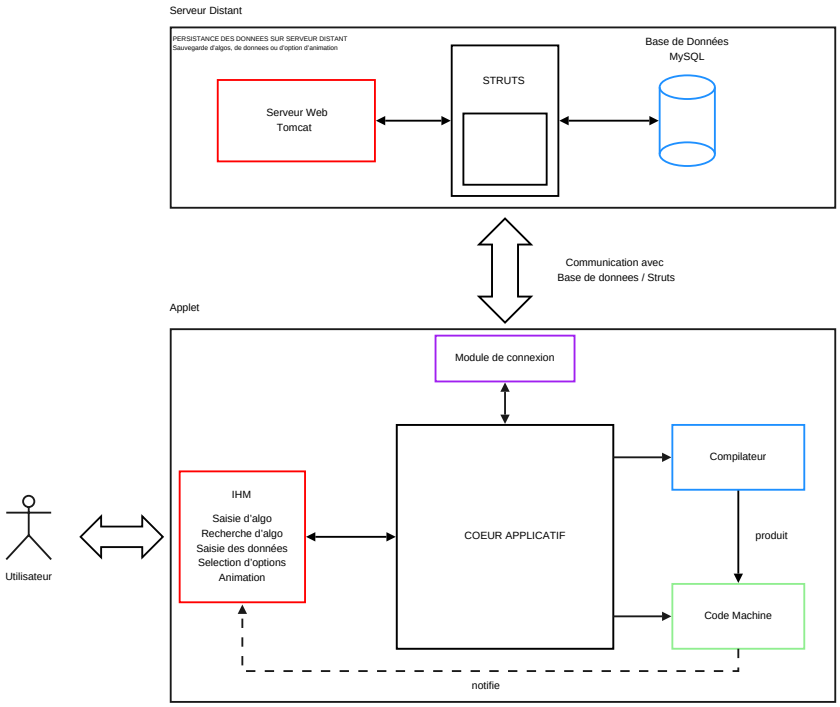


FIG. 3.1 – Vue d'ensemble de l'application

3.2 Principes du développement

Cette partie présente certains principes de développement que nous avons décidé d'appliquer. La première section présente les choix technologiques pour lesquels nous avons opté. La seconde mettra en avant la nécessité de la modularité. Enfin, la troisième section expliquera pourquoi nous tenterons d'éviter à tout prix les dépendances circulaires entre les *packages* de l'application.

3.2.1 Choix technologiques

Le cœur de l'application s'exécute côté client car la compilation est un processus lourd, et même optimisé, le serveur serait vite surchargé. De plus, les ressources serveur sont toujours plus coûteuses que les ressources de l'utilisateur. Enfin, ceci permettra d'économiser l'utilisation d'un code machine sous forme textuelle pour transmettre l'algorithme compilé du serveur à l'utilisateur.

Nous avons utilisé l'outil de création de compilateurs JavaCC, programmé en Java et intégrable dans Eclipse. Il génère des analyseurs syntaxiques programmés en Java reconnaissant les grammaires LL(k) et permet l'utilisation d'expressions régulières pour décrire la grammaire. Enfin son fonctionnement est similaire à YACC. Nous l'avons utilisé dans sa version 3.2 car c'était la dernière version stable au début de la phase de développement.

JUnit est un outil de test unitaire simple d'emploi intégré dans Eclipse. Sa rapidité de prise en main a été notre principal critère de choix compte tenu du temps imparti pour le développement de l'application.

Nous nous sommes orientés vers l'utilisation d'une base de données pour sauvegarder les algorithmes, les animations et les jeux de données, car elle présente plusieurs avantages. En effet, Java possède une interface standardisée pour accéder à une base de données et Struts gère la communication avec elle. Enfin, elle offre la possibilité de stocker des données sous forme de texte structuré. Nous utiliserons une base de données MySQL, à laquelle on accédera à l'aide de classes Java via Struts. L'utilisation de Struts repose sur un serveur d'exécution ou conteneur Web. Parmi les nombreux conteneurs Web existant, nous avons choisi d'utiliser Tomcat (Apache) car il est traditionnellement utilisé avec Struts et apporte toutes les fonctionnalités nécessaires sans grand ajout de complexité.

Notre décision d'utiliser Linux pour le serveur vient principalement de sa licence et du fait que la plupart des outils dont nous avons besoin sont disponibles pour ce système d'exploitation sous licence open-source. De plus, l'utilisation d'une distribution basée sur des paquets (Debian) simplifie l'installation du serveur.

De manière générale, les outils choisis l'ont été pour répondre efficacement et le plus simplement possible aux besoins de notre application. Enfin, nous nous sommes limités à l'utilisation de logiciels libres.

3.2.2 Modularité de l'application

Dans l'optique d'une évolution du code, on impose à l'application développée des propriétés de modularité. Celles-ci permettent de traiter séparément chacun des modules de l'application et ainsi de favoriser la répartition du travail, les tests d'intégration et l'évolution des modules pourvu qu'ils implémentent des interfaces bien définies.

La conception objet en elle-même introduit une certaine modularité dans l'application, de plus une factorisation régulière du code permet d'accroître cette modularité. D'autre part, l'emploi de patrons de conception tel qu'observateur ou MVC guide le découpage de l'application en modules cohérents et assure un découplage optimal, ainsi certaines parties de l'application pourront facilement être remplacées par de nouvelles versions. Par exemple, les observateur qui sont chargés de l'animation d'un type ont pu et pourront être ré-utilisés du moment que la variable associée est observable et que le type est représentable, et ce, peu importe la méthode employée par le moteur d'exécution pour dérouler l'algorithme. Enfin, l'utilisation judicieuse d'interfaces les plus globales possible a également permis de garantir un bon découpage de l'application.

3.2.3 Concepts avancés

Nous décrivons dans cette partie des technologies avancées, telles que la réflexivité Java ou le *multithreads*, que nous avons utilisées dans l'application.

La réflexivité Java

La réflexivité Java est un des mécanismes qui fait la souplesse de Java et qui est souvent utilisé dans les applications Java de haut niveau de par sa nature hautement dynamique. Ce mécanisme est notamment utilisé au sein de JUnit pour faciliter l'écriture des classes de test et éviter l'utilisation de scripts fastidieux.

Voici son fonctionnement : en Java, il y a des classes pour définir chaque entité du langage, en particulier la superclasse `Class`. Ainsi, au travers de cette dernière nous pouvons accéder aux méthodes d'une classe que nous avons définie. De plus, nous pouvons connaître les classes dont elle dérive ainsi que toutes les interfaces qu'elle implémente (sous la forme d'instances de la classe Interface). Enfin, summum du dynamisme, nous pouvons appeler et instancier le constructeur d'une classe dynamiquement à partir de son nom sous forme de chaîne de caractères.

La classe packageExplorer Le défaut de la réflexivité Java est qu'elle est incomplète. En effet, la classe `Package` décrivant un *package* Java au sein du langage ne permet pas, pour l'instant, de lister son contenu. Nous avons donc eu besoin de trouver une solution pour combler cette lacune. Voici son fonctionnement : Un *package* sur le disque se comporte comme un répertoire contenant les fichiers .class où sont définies les classes du *package*. Nous devons donc, dans un premier temps, ouvrir le répertoire et lister son contenu afin de connaître les classes définies dans ce *package*. Dans un second temps, nous instancions les classes trouvées et nous les filtrons par les interfaces qu'elles implémentent. Lorsque la bonne classe à instancier est sélectionnée, il suffit d'en générer une instance et de la caster selon le type polymorphe qui nous intéresse (`aAnimations` ou `aaFunction`).

Threads

Notre application est composée de différents *Threads*. La programmation *multithreads* implique de nombreux problèmes de synchronisation mais s'avère nécessaire. Java utilise en effet un *Thread* pour gérer l'IHM : le *event-dispatching Thread*. Ce *Thread* est créé automatiquement dès qu'une IHM est utilisée. La construction, la modification de l'IHM, le lancement et le traitement des événements ont lieu au sein de ce *Thread*. Si le développement de petites applications ne nécessitent pas de précautions particulières, des problèmes apparaissent dès que l'application possède des tâches assez longues.

Problèmes Une IHM en Java utilise les principes de programmation événementielle. Chaque action sur l'IHM (déplacement de la souris, clic sur un bouton...) déclenche le lancement d'un événement. On peut associer à un événement particulier une action à exécuter chaque fois qu'un tel événement est lancé. Par exemple, un clic sur le bouton *Compil* va lancer la compilation de l'algorithme saisi. Comme les événements, les actions vont s'exécuter dans le *event-dispatching Thread*. Or ce *Thread* doit aussi gérer l'affichage de l'IHM. Pendant le traitement d'un événement, le *Thread* ne peut exécuter aucune autre opération. L'IHM semble donc bloquée et des bugs d'affichage apparaissent (boutons restant enfoncés, zones mal dessinées...).

Pour résoudre ce problème, toutes nos tâches longues (à savoir la compilation et l'exécution d'un algorithme) sont chacune exécutées dans un *Thread* dédié. Le traitement des événements associés à ces tâches longues se résume à la création et le lancement d'un *Thread*. Ces opérations prennent un temps négligeable et le *Thread* de l'IHM reste disponible. Bien sûr, ces multiples *Threads* imposent une synchronisation.

Synchronisation Pendant la compilation, aucune action n'est possible ; l'IHM est donc bloquée. Ce blocage est volontaire et n'empêche pas le rafraîchissement de l'IHM. L'exécution et l'animation d'un algorithme représentent la plus grosse difficulté de synchronisation. L'exécution se déroule dans un *Thread* dédié et l'animation dans le *event-dispatcher Thread* (toutes les modifications graphiques doivent avoir lieu dans ce *Thread*). Nous avons donc synchronisé le patron de conception Observateur utilisé.

3.2.4 Diagramme de paquetages de l'*Applet*

Le découplage des modules d'une application se fait à l'aide de paquetages. Les paquetages regroupent des classes dans le but de fournir une fonctionnalité cohérente. Les relations entre classes à l'intérieur d'un paquetage ne doivent pas être prise en compte : on considère que chaque classe *voit* (*import* en Java) les autres classes du paquetage. Cependant, les relations de dépendance entre paquetages sont un bon critère de qualité pour une application. Il faut éviter à tout prix les dépendances circulaires. Une bonne organisation des blocs fonctionnels d'une application augmente la modularité et facilite le développement. La figure 3.2 donne le diagramme de paquetage de l'application globale (*Applet* Java). Il ne présente pas de dépendance circulaire et sa racine est le paquetage principal. Lors de l'exécution d'une *Applet*, la classe principale, moteur de l'application, réalise également l'interface avec l'utilisateur, cependant elle ne sera que le conteneur des éléments graphiques qui font partie du paquetage IHM.

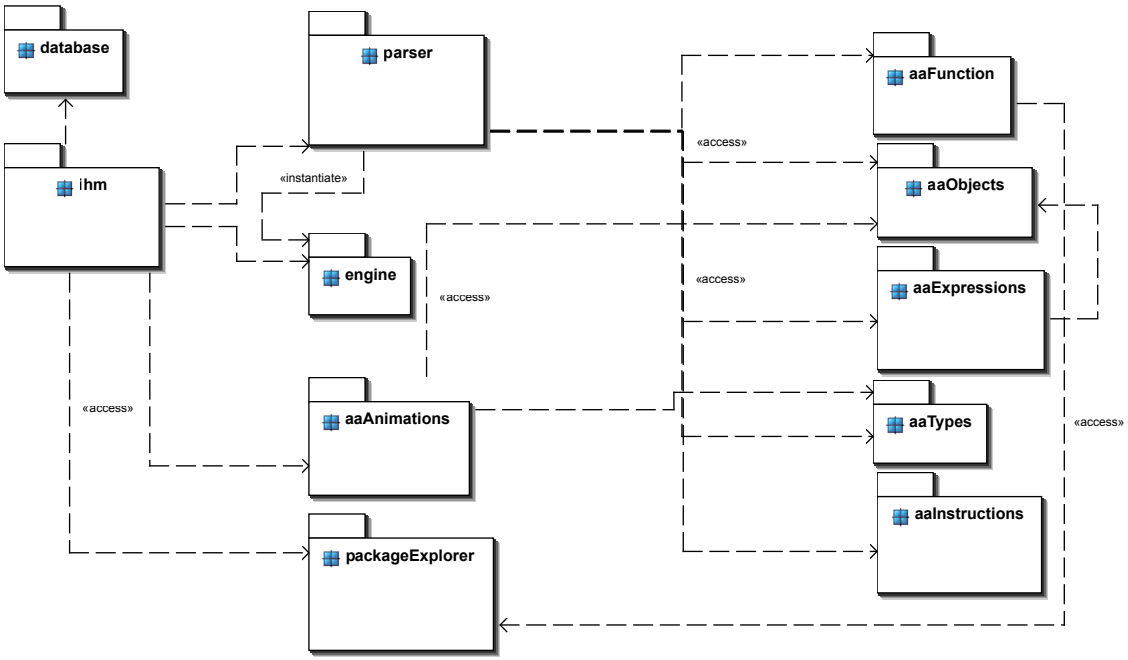


FIG. 3.2 – Diagramme de paquetages de l'*Applet*

3.3 L'IHM

L'*interface homme-machine* (IHM) a pour rôle d'assurer la communication entre l'utilisateur et l'application en fournissant des services ergonomiques. En effet, elle permet à l'utilisateur d'accéder aux fonctions de notre application par l'intermédiaire d'interactions avec des objets graphiques, et réciproquement l'application se sert de l'interface pour envoyer divers messages à l'utilisateur (messages d'erreur, résultats...). Dans le cadre de notre projet, nous avons utilisé la librairie Swing

de Java (disponible depuis la version 1.3) qui donne de très bons résultats graphiques et AWT pour la gestion des agencements dans les fenêtres graphiques et des événements.

L'une de nos préoccupations a été de découpler l'IHM des objets métiers et des objets décrivant l'algorithme saisi :

- objets représentant les objets du langage,
- objets représentant les variables,
- objets recevant les commandes.

L'utilisation de patrons de conceptions nous a permis de modulariser l'application. Des modifications dans l'interface homme-machine sont donc possibles sans avoir à modifier les autres parties de l'application du moment que les interfaces spécifiant les patrons de conception restent les mêmes. Dans cette partie on décrit les patrons utilisés pour développer notre application de manière modulaire en fournissant les diagrammes faisant intervenir les classes de notre projet. Pour plus de détails sur les patrons de conceptions et leur utilisation, on se reportera à [6].

3.3.1 Modèle, Vue, Contrôleur (MVC)

Ce patron réalise une séparation claire entre les données et la partie graphique affichant les données. Pour ce faire, on sépare l'application en trois parties distinctes :

Le modèle représente les données sur lesquelles on peut effectuer des opérations. Il s'agit par exemple des objets représentant les variables, les listes d'animations possibles pour une variable ou encore les résultats d'une recherche dans la base de données. Enfin le modèle notifie la vue de ses changements.

La vue correspond à la représentation visuelle par l'IHM des données du modèle

Le contrôleur réagit aux actions et aux données entrées par l'utilisateur. Il effectue alors les actions nécessaires sur le modèle.

La modification éventuelle d'une partie de l'interface a peu d'impact sur les autres sections. On n'est ainsi pas obligé d'apporter beaucoup de modifications au modèle le cas échéant. La figure 3.3 donne le schéma des interactions entre la vue, le contrôleur et le modèle MVC utilisé par Swing. La notification et l'activation sont réalisées de manière asynchrone par le biais des *Listener* de Java.

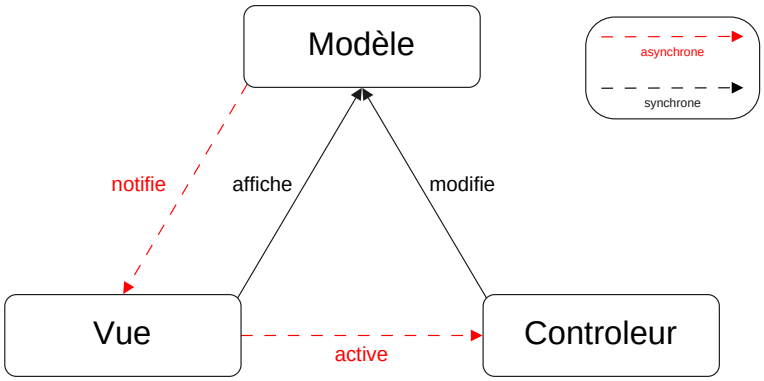


FIG. 3.3 – Schéma MVC de Swing

3.3.2 Observer

Le modèle Observateur définit une relation entre un objet à observer et un objet observateur. Il permet la mise a jour du panneau d'affichage d'une variable à chaque fois qu'elle est modifiée, tout en découplant l'IHM des objets manipulés par l'algorithme.

`Observable` classe Java qui fournit les méthodes pour ajouter et enlever les objets observateurs. La méthode *notifyObservers()* permet de signaler à tous les observateurs que la variable à été modifiée.

`AAVariable` est l'objet que l'on veut observer. Il hérite d'`Observable` via `AAMutable`.

`Observer` définit une interface pour les objets qui doivent être mis à jour à chaque modification de la variable

`AAAnimation` implémente l'interface `Observer`. La méthode *notify()* actualise le panneau en prenant en compte la nouvelle valeur de la variable

La figure 3.4 montre l'implémentation du patron de conception *Observer* au projet. Ce patron de conception est utilisé intensivement pour les animations.

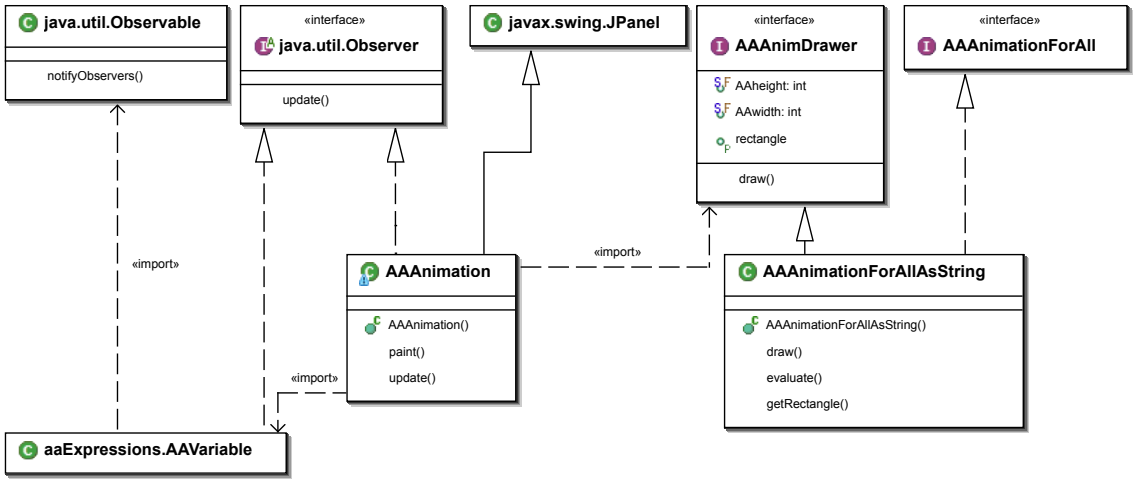


FIG. 3.4 – Patron de conception *observer* utilisé au sein des animations

3.4 Le langage

Le langage est un point important de notre application. Nous décrivons dans cette partie sa modélisation et sa mise en œuvre à l'aide de la sémantique opérationnelle structurée. Nous verrons ensuite la phase de compilation et la génération en un code machine exploitable et son exécution proprement dites.

3.4.1 Modélisation du langage

L'application étant exécutée sur une machine virtuelle Java, on peut exploiter cette machine comme moteur d'exécution dans notre application. D'autre part, le compilateur et le moteur d'exécution étant deux parties pouvant communiquer par le biais du module principal, il leur est possible de partager des données structurées sous forme de classes Java et donc de partager le code engendré par le premier sous la forme de données structurées.

Deux solutions se présentent donc pour gérer le moteur d'exécution et le code associé. La première consiste à produire du code natif Java (*bytecode*) et l'exécuter sur la machine virtuelle exécutant l'*Applet*. Une phase d'étude (notre principale référence étant [7]) a montré que cette solution est réalisable et efficace mais nécessite l'utilisation d'une plate-forme aidant à la production de code natif Java par l'intermédiaire de mnémoniques. Le projet KAWA fournit une telle plate-forme, cependant celle-ci nécessite beaucoup de temps pour être appréhendée. La seconde solution, que nous avons retenue pour sa simplicité et la rapidité de sa mise en œuvre, consiste à représenter le code machine et les objets manipulés sous la forme d'objets Java structurés (cette solution

étant également retenue par OGRE, voir partie 1.2.4). L'exécution d'un programme est donc une suite d'appels de méthodes Java sur ces objets. Ce mode d'exécution est dit *opérationnel*. On présente dans un premier temps cette méthode de description de programmes structurés puis son implémentation dans le cadre de notre projet, c'est-à-dire en Java et pour le langage algorithmique choisi.

Sémantique Opérationnelle Structurée

Le principe de la sémantique opérationnelle structurée (SOS) est d'associer à un programme un *sens* ou *sémantique* décrivant son exécution par une machine idéale qui reflète les structures du langage dans lequel le programme est écrit. Pour décrire de manière formelle la SOS, on travaille généralement sur des domaines décrivant les programmes (ou instructions), les valeurs, les variables, les expressions et les environnements (décrivant les valeurs des variables à un instant de l'exécution). Dans le cas de notre projet, nous n'avons pas utilisé d'environnement explicite mais celui de Java : les variables étant des objets Java, elles contiennent leur valeur et peuvent être mises à jour par appel de méthode et donc sans utiliser d'environnement explicite.

Dans le cadre de l'animation d'algorithmes, le domaine des valeurs décrit les objets manipulés par le programme et correspond donc aux types définis dans la partie 2.1. Les expressions sont définies inductivement de la même manière que dans la grammaire du langage. Sans détailler pour l'instant l'évaluation des expressions, on supposera que l'on dispose d'une méthode la réalisant pour décrire la sémantique des programmes écrit dans le langage algorithmique. On notera $[\![e]\!]$ le résultat de l'évaluation de l'expression e.

Le cas des primitives du langage n'est pas traité ici car leur description est faite dans la partie 2.1.2, cependant on détaille avec soin les *structures de contrôle* du langage car elles présentent l'idée de base d'une telle sémantique. Un programme est représenté de manière structurée comme une combinaison d'instructions de base telle que les primitives ou les affectations. Ces instructions de base sont organisées à l'aide de structures telle que le *séquencement* (décrivant l'enchaînement d'instructions), les *conditionnelles* (*if*) et les *itérations* (*for*, *foreach* et *while*). Donnons les règles d'exécution d'un *pas de calcul* pour les quatre types de structures évoqués où l'instruction Stop marque la fin d'une étape. On utilise pour cela des règles de calcul qui ont la forme suivante :

$$\frac{\texttt{étape de calcul}}{\langle \texttt{configuration avant} \rangle \xrightarrow{mem} \langle \texttt{configuration après} \rangle}\ \texttt{nom de la règle}$$

L'affectation évalue sa partie droite et lie la valeur obtenue $[\![\texttt{e}]\!]$ à la variable désignée en partie gauche. L'affectation constitue une étape de calcul élémentaire puisqu'elle n'est pas définie par une autre étape de calcul.

$$\frac{}{\langle \texttt{i :=e} \rangle \xrightarrow{i \leftarrow [\![\texttt{e}]\!]} \langle \texttt{Stop} \rangle}\ \texttt{Ax}$$

La séquence P ; Q demande de faire une étape de calcul dans P, puis continue. On commence à exécuter Q seulement lorsque P est terminé (Stop).

$$\frac{\langle \texttt{P} \rangle \longrightarrow \langle \texttt{Stop} \rangle}{\langle \texttt{P ; Q} \rangle \longrightarrow \langle \texttt{Q} \rangle}\ \texttt{Seq} \qquad \frac{\langle \texttt{P} \rangle \longrightarrow \langle \texttt{P'} \rangle}{\langle \texttt{P ; Q} \rangle \longrightarrow \langle \texttt{P' ; Q} \rangle}\ \texttt{Seq}$$

La conditionnelle if b P1 P2 evalue b ($[\![b]\!]$). Selon le résultat obtenu, P1 ou P2 est exécuté.

$$\frac{}{\langle \texttt{if b \{P1\} \{P2\}} \rangle \longrightarrow \begin{cases} \langle \texttt{P1} \rangle & \text{si } [\![b]\!] \text{ est vraie} \\ \langle \texttt{P2} \rangle & \text{si } [\![b]\!] \text{ est fausse} \end{cases}}\ \texttt{If}$$

La boucle while b P évalue b ($[\![b]\!]$) et exécute P puis relance une boucle si b vaut vrai, ou s'arrête si b vaut faux.

$$\frac{}{\langle \texttt{while b \{P\}} \rangle \longrightarrow \begin{cases} \langle \texttt{P ; while b \{P\}} \rangle & \text{si } [\![b]\!] \text{ est vraie} \\ \langle \texttt{Stop} \rangle & \text{si } [\![b]\!] \text{ est fausse} \end{cases}}\ \texttt{While}$$

Modélisation des instructions

Cette partie explique la modélisation objet de la SOS d'un programme et son exécution. On décrit donc dans un premier temps les classes implémentant les instructions ainsi que leur organisation puis dans un second temps on précise le corps de quelques méthodes *execute* spécifiées par l'interface `InterfaceAAInstruction`.

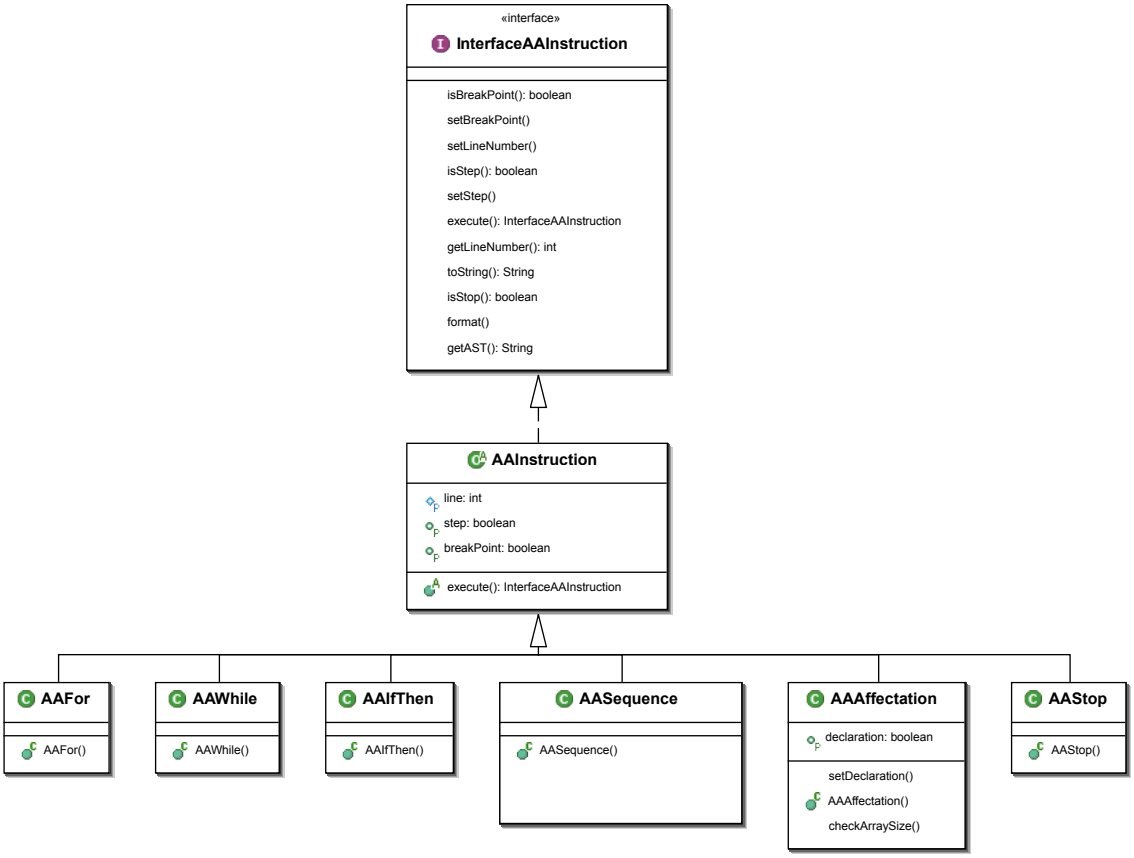


FIG. 3.5 – Diagramme de classe des instructions

La figure 3.5 présente l'organisation des instructions et de la sémantique d'un programme. La sémantique d'une instruction (on confondra désormais une instruction, sa sémantique et la classe qui l'implémente) doit donc implémenter une méthode *execute* exécutant un pas de calcul et une méthode *isStop* permettant de déterminer si l'instruction est terminée (Stop de la SOS). Une instruction est représentée sous forme arborescente et un programme est décrit par une instruction initiale (racine de l'arbre complet) et l'instruction courante (nœud interne de l'arbre). La sémantique utilisée étant structurée (absence de saut inconditionnel dans le langage), à chaque pas de calcul on peut abandonner un ou plusieurs sous-arbres de l'arbre du programme en s'*enfonçant* dans celui-ci.

Un programme ainsi décrit est généré à la compilation. Pour animer l'algorithme qu'il implémente, on doit l'exécuter. Dans ce cas, une partie du moteur d'exécution est incluse dans le programme lui même grâce aux méthodes *execute* des objets instructions. Cette méthode renvoie alors l'instructions à exécuter *ensuite*. La sémantique adoptée est dite *petits pas* pour les instructions en ceci que l'exécution d'un pas de programme « avance » d'une instruction atomique dans le programme. Dans le cas d'une structure conditionnelle, on exécute uniquement l'évaluation de la condition en un pas de calcul. Cependant l'évaluation des expressions est dite *grands pas*. On donne le code de la méthode *execute* du *if then else* pour mieux comprendre le mécanisme d'exécution.

```
If(InterfaceAAExpression e,InterfaceAAInstruction i1,InterfaceAAInstruction i2) {
        this.i1 = i1; this.i2 = i2; this.e = e;
}

public InterfaceAAInstruction execute() {
        AABoolean b = (AABoolean) e.evaluate();
        if(b.getValue())
                return i1;
        else
                return i2;
}
```

On voit donc que la construction d'une instruction *if* se fait à l'aide de l'expression décrivant la condition et des instructions à réaliser suivant le résultat de cette condition. D'autre part, on remarquera que le code produit est une traduction directe des règles d'exécution décrites dans la partie précédente. Enfin l'exécution d'un pas de programme revient à remplacer l'instruction courante par le résultat de l'appel de la méthode *execute* de cette instruction.

Modélisation des valeurs et expressions

Le but de cette partie est d'expliciter l'implémentation des valeurs et des expressions manipulées par les programmes qui sert de base à l'implémentation de la SOS. Pour représenter la structure des objets Java utilisés pour réaliser la modélisation, on utilise des diagrammes UML (de classes principalement).

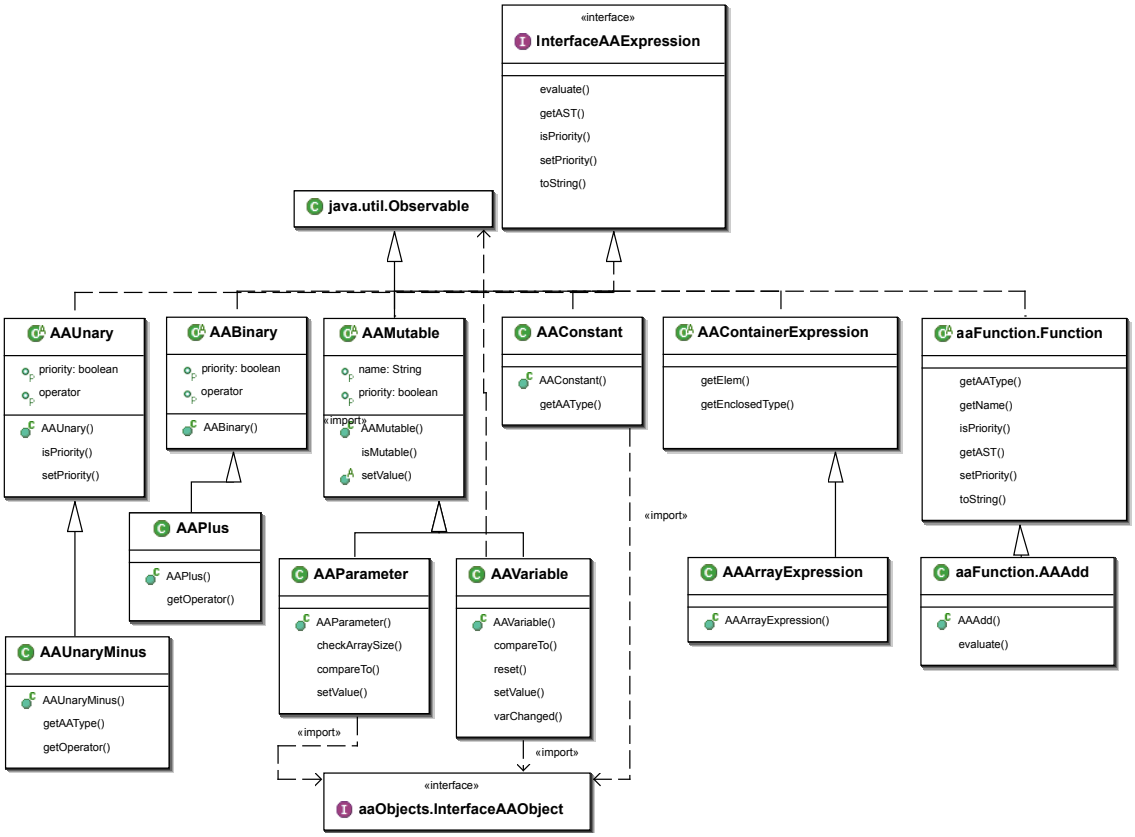


FIG. 3.6 – Diagramme de classe des expressions

La figure 3.6 représente la structure des classes modélisant les expressions du langage. La spécification des expressions est réalisée par l'interface `InterfaceAAExpression` imposant la méthode d'évaluation *evaluate*. La définition de classes abstraites pour les opérateurs unaires et binaires permet de `factoriser` les comportements similaires. D'autre part les expressions se déclinent principalement en trois formes : les constantes (`AAConstant`) et les représentations des types composés, les mutables (`AAVariable` et `AAParameter`) et les expressions composées (`AABinary`, `AAUnary` et les `AAFunction`). Pour des raisons de clarté, ce diagramme n'est évidemment pas exhaustif. Enfin, la représentation des expressions est arborescente et l'évaluation est effectuée par parcours en profondeur de cet arbre. Les feuilles de l'arbre sont les constantes et les mutables. L'évaluation fait donc *remonter* un objet de type `InterfaceAAObject` (interface modélisant les objets du langage algorithmique) à la racine.

Certaines opérations, telles que la division, peuvent provoquer des erreurs à l'évaluation d'une expression. Celles-ci sont gérées à l'aide d'exceptions Java et leur gestion sera détaillées dans la partie traitant du moteur d'exécution (voir partie 3.4.3).

Maintenant que la modélisation du langage est clairement définie, nous allons voir comment la phase de compilation transforme un algorithme textuel en un objet exploitable utilisant cette modélisation.

3.4.2 La compilation

Les algorithmes saisis par l'utilisateur sont fournis sous forme textuelle puis convertis sous forme d'*Arbre de Syntaxe Abstraite* (AST) par un *parser*. Pour être exécuté de manière continue ou pas à pas, cet AST doit être transformé en un *code machine* qui est exploitable par un *moteur d'exécution* intégré à l'*Applet* et donc programmé en Java. C'est la phase de compilation qui est chargée de produire ce code machine. Elle se décompose en trois phases : l'analyse lexicale qui transforme une chaîne de caractères en une suite d'éléments lexicaux (*token*), puis une analyse syntaxique qui construit l'AST à partir de ces éléments lexicaux et enfin l'analyse sémantique qui produit le code à partir de l'AST en détectant les éventuelles erreurs (dans la mesure du possible, les divisions par 0 par exemple ne peuvent pas être détectées à la compilation...). L'outil utilisé pour la génération du compilateur est JavaCC (Java Compiler Compiler).

Introduction à JavaCC

JavaCC produit un analyseur lexical et syntaxique reconnaissant une grammaire à partir de ses spécifications. Les analyseurs syntaxiques produits sont LL(1), codés en Java. Cependant pour régler les conflits dans certaines règles de la grammaire, on peut préciser le *lookahead* (taille du tampon de pré-lecture) : le *parser* devient alors LL(k) à l'intérieur de cette règle.

Les spécifications lexicales et grammaticales sont faites dans un même fichier (d'extension .jj), d'autre part, on peut ajouter des méthodes au *parser* pour faciliter par exemple l'analyse sémantique. La dernière version stable au début du développement était JavaCC 3.2. La version 4 est désormais stable et reste compatible avec la version précédente. Un *refactoring* pourra avoir lieu dans une version future en utilisant la généricité de Java 5. Pour cette raison le code du parser est écrit en Java 4 (en particulier, la programmation générique n'est pas utilisée).

JavaCC est d'une grande souplesse pour les spécifications de la grammaire. En effet il autorise l'utilisation de variables *locales* à une règle et d'expressions régulières sur les éléments lexicaux. On donne comme exemple l'axiome de la grammaire JavaCC (extraite de la grammaire du langage dans lequel les points de génération ont été supprimés pour des raisons de lisibilité).

```
InterfaceAAProgram start() throws AAException :
{
        String name;
        InterfaceAAInstruction decl;
        InterfaceAAInstruction instr;
        InterfaceAAInstruction result;
```

```
}
{
        (
        <PROG> name=header() decl=declarations() instr=instructions() <END>
        )
        {}
}
```

Enfin, la gestion des erreurs syntaxiques à la compilation est très bien prise en charge par JavaCC. En effet les classes générées fournissent une API complète pour générer des messages d'erreurs précis (éléments lexicaux attendus, ligne et colonne de l'erreur...) Ces erreurs sont gérées grâces aux exceptions Java.

Analyses lexicale et syntaxique

On définit dans un premier temps les éléments lexicaux du langage. Ils correspondent aux symboles terminaux de la grammaire du langage. Ces définitions sont faites directement dans le fichier JavaCC, qui permet de déclarer un certain nombre de *tokens*, composés de un ou plusieurs caractères (par exemple, l'opérateur <= est considéré comme un unique *token*).

La grammaire présentée dans la partie 2.1.1 ne peut pas être donnée telle quelle pour définir l'analyseur syntaxique. En effet, pour les expressions par exemple, la spécification de la grammaire ne tient pas compte des priorités de certains opérateurs sur d'autres ($\times$ plus prioritaire que $+$ par exemple). On gère les priorités *manuellement* en étageant la grammaire (c.-à-d. : en définissant plusieurs non-terminaux $expr_i$). Les opérateurs les plus loin de l'axiome (c.-à-d. : *expr*) sont les plus prioritaires. Pour les booléens par exemple les opérateurs sont séparés en trois niveaux de priorité (par ordre croissant) : { **or** }, { **and** }, { **not** }. Dans le cas d'opérateurs de même priorité, ils sont traités de gauche à droite (ces priorités sont décrites dans le tableau 2.1).

Gestion des primitives du langage

La liste des noms de fonctions ne peut évidemment pas être complètement énumérée dans la grammaire (il ne serait pas raisonnable de modifier la grammaire à chaque fois qu'on ajoute une nouvelle fonction au langage). Il était donc intéressant de trouver une méthode permettant de vérifier dynamiquement si une fonction existe, ainsi que la correction des paramètres passés lors de l'appel de la fonction. La réflexivité est une solution tout a fait adaptée à ce problème. En effet, nous définissons une classe pour chaque fonction. Dans le *parser*, il suffit donc de vérifier qu'une classe ayant le bon nom existe bien, auquel cas on instancie cette classe. Le constructeur se chargera ensuite de vérifier les types des paramètres.

Analyse sémantique et génération de code

L'analyse sémantique consiste à déterminer si un programme syntaxiquement correct *a du sens* et dans ce cas à générer le code machine correspondant. On entend par *avoir du sens*, qu'un programme ne réalise pas d'opération illicite telle qu'affecter un entier à une variable booléenne. On distingue ces erreurs des erreurs d'exécution telles que la division par zéro (problème indécidable à la compilation). Le principal souci de notre analyse sémantique sera la vérification de types. Pour ce faire on dispose d'une table des symboles pour déterminer les types des expressions et vérifier qu'ils coïncident. On traitera également la redéfinition de variables ou l'utilisation de variables non définies. Les erreurs sémantiques, tout comme les erreurs syntaxiques, sont gérées par des exceptions Java.

On donne un exemple de règle de grammaire décorée pour la génération de code, extraite du compilateur, pour un *étage* d'expression et une structure de contrôle simple.

```
InterfaceAAExpression exp() throws AAException :
```

```
{
        Token t;
        InterfaceAAExpression e1;
        InterfaceAAExpression result;
}
{
        (
                result=exp2()
                (
                        t=<OR> e1=exp()
                        {
                                result= new AAOr(result,e1,t.beginLine);
                        }
                )?
        )
        {
                return result;
        }
}

InterfaceAAInstruction instruction() throws AAException :
{
        InterfaceAAExpression e1;
        InterfaceAAInstruction i1;
}
{
        t=<WHILE> e1 = exp() <OBRACE> i1 = instructions() <EBRACE>
        {
                result = new AAWhile(e1,i1,t.beginLine);
        }
        {
                return result;
        }
}
```

Pour chaque création d'objet, on passe au constructeur la valeur `t.beginLine` (cette valeur contient la ligne débutant le *token* t). En effet ce sont les constructeurs qui effectuent les contrôles de types, et ils ont donc besoin de connaître la ligne concernée pour pouvoir signaler à l'utilisateur à quelle ligne a lieu l'éventuelle erreur de type.

Contrôle de types

Une part importante du travail à faire sur le langage a été la vérification de types. Celle-ci ne peut évidemment pas être totalement exprimée dans la seule grammaire. Malgré cela, il est important de voir que la vérification de type reste totalement statique ; une erreur de type n'arrive donc jamais en cours d'exécution du programme, mais uniquement lors de la compilation. Les types des expressions sont divisés en deux familles :

Les types simples (entier, caractère...) Ces types disposent d'une méthode *getAAType()* permettant de connaître le type d'une expression. La gestion de ces types est très simple, car par exemple tous les entiers sont de même type (on verra que ce n'est pas le cas avec les types composés). L'objet décrivant le type entier (retourné par *getAAType()*) peut donc être partagé par tous les entiers. Nous utilisons ici le patron de conception *singleton*, qui assure qu'une seule instance de chaque type sera créée. Quand on demande le type d'un entier, c'est toujours la même instance qui est retournée.

Les types composés (ensembles, listes...) Contrairement aux types simples, on se rend compte que le nombre de types composés n'est pas fini : à chaque fois qu'on a une expression e de type t, on peut toujours construire par exemple l'expression e, qui est de type ensemble de t (t *set*). On ne peut donc pas envisager d'utiliser le singleton ici, car cela voudrait dire qu'on crée une classe par type, ce qui est impossible. Chaque expression instancie donc un nouvel objet pour décrire son type. Bien entendu cette instanciation n'est faite qu'une seule fois, au moment de la compilation de l'expression.

On est amené à définir un type spécial, considéré comme un type générique. En effet, il n'est pas possible par exemple de typer l'ensemble vide, car il peut s'agir de l'ensemble vide des entiers ou de celui des caractères, et on n'a aucun moyen de le savoir au moment de l'analyse de l'expression. Ce raisonnement vaut pour tous les types de données composés (liste vide, pile vide...) D'un point de vue sémantique, ce type est égal à n'importe quel type ensemble, c'est ainsi qu'on autorise d'affecter l'ensemble vide à un ensemble d'entiers.

L'analyse de certaines expressions de type composé donne lieu à une déduction de type, comparable à celle qu'on observe dans certains langages de programmation fonctionnelle, même si elle est ici relativement simple. La présence d'éléments vides (ensemble, liste, ...) introduit parfois une certaine incertitude : par exemple quand on analyse l'expression $\{\{\}, \{2\}\}$, on ne connaît pas son type tant qu'on n'a pas analysé $\{2\}$. On ne connaît pas non plus le type de l'ensemble vide. Par contre on est capable de le déduire à la fin, car $\{2\}$ est un ensemble d'entiers, l'expression globale est donc un ensemble d'ensemble d'entiers. Ce cas est relativement simple, mais on peut imaginer des cas plus compliqués ou le nombre d'imbrications serait plus grand. La méthode que nous utilisons pour déterminer le type d'une expression est de parcourir toute l'expression, puis de chercher quel est le type le plus spécifique ($\{2\}$ dans l'exemple précédent). On déduit ainsi de manière le plus précise possible le type de l'expression.

3.4.3 Moteur d'exécution

La phase de compilation traduit un programme source en code machine. A ce stade, le code machine n'est pas directement exploitable. En effet, il ne dispose que d'une primitive *execute()* sur chacune des instructions. Il est donc nécessaire d'avoir recours à un moteur d'exécution offrant des primitives plus élaborées (*play*, *pause*, *previous*...) Le moteur utilise le code machine pour réaliser les actions demandées. Ces actions sont invoquées par l'IHM.

Pour ne pas bloquer l'IHM pendant l'exécution (éventuellement infinie) du programme, on gère les actions du moteur d'exécution comme des tâches parallèles (*Thread* Java, voir partie 3.2.3). Ainsi, les commandes d'exécution (*stop*, *pause*...) répondent pendant l'exécution et peuvent ainsi l'interrompre. Cette exécution en parallèle est également nécessaire pour permettre aux de se mettre à jour. Ce point sera détaillé dans la partie suivante. Enfin, elle offre la possibilité à l'utilisateur de varier la vitesse d'exécution du programme en cours. Cette fonctionnalité est mise en œuvre par un temps d'attente plus ou moins long entre chaque instruction (un pas au sens de la SOS).

Le moteur d'exécution offre différentes fonctionnalités à l'utilisateur. Nous allons présenter brièvement leur mise en œuvre. La fonction *play* consiste en une succession d'appels à la méthode *execute* de l'instruction courante (celle-ci évoluant au cours du temps). La fonction *pause* (respectivement *stop*) permet d'arrêter le *Thread* relatif à la méthode *play* sans modifier l'état courant (respectivement en initialisant le programme à sa configuration de départ). La fonction *previous* permet de reculer d'un pas l'exécution courante. Pour cette fonction, nous faisons une ré-exécution du programme depuis sa configuration d'origine en exécutant une instruction de moins, elle est donc d'autant plus coûteuse que l'exécution du programme a commencé il y a longtemps. Une solution envisageable est un cache d'états de l'exécution permettant de ne pas tout ré-exécuter. Cependant, lors de nos expérimentations, nous n'avons pas constaté de temps de calcul insupportable dûs à cette méthode.

Le dernier rôle du moteur d'exécution est de gérer les erreurs d'exécution (telles que la division par 0). Cela consiste juste à attraper des exceptions générées par l'exécution d'un pas. Les exceptions de type `AAException` sont celles que l'on a lancées (par exemple dans l'instruction de division avec un dénominateur nul). Quand on attrape une exception de ce type, le traitement consiste à arrêter l'exécution du programme et afficher un message décrivant l'erreur à l'utilisateur. On envisage aussi d'attraper les autres erreurs (on ne sait pas a priori pour quelle raison elles sont lancées). On effectue donc le même traitement, mais avec un message d'erreur plus général, spécifiant seulement qu'il s'agit d'une erreur d'exécution.

3.5 Les animations

La phase d'animation s'effectue en deux temps, la sélection puis l'animation en elle-même. La sélection peut être recherchée dans la base de données ou être choisie directement à travers les listes d'animations possibles pour chaque variable. Il existe des animations pour tous les types de variables, certaines sont spécifiques aux types simples ou aux types composés et enfin des animations sont créées spécialement pour un type donné.

Grâce a l'utilisation d'animations pour les types de donnée simples (entiers, booléens...) comme pour les types complexes (tableaux, ensembles...) et l'imbrication de ces derniers dans les premiers permet d'offrir une combinaison d'animation qui n'est limitée que par l'imagination de l'utilisateur et sa surface d'affichage.

3.5.1 La technologie utilisée

Nous utilisons Java2D pour composer la représentation graphique des variables de l'algorithmes. Java2D étant compatible avec `AWT.graphics`, n'importe qui ayant des notions de dessin avec AWT peut créer une animation compatible avec notre application. Chaque représentation de variable est composée au sein d'une boîte `JPanel` (composant de Swing).

De plus, le *patron Observateur*, utilisé pour contrôler le changement d'état d'une variable, a été implémenté en réutilisant directement la classe `Observable` et en implémentant l'interface `Observer` dans l'animation. De ce fait, les animations sont découplées des variables, elles peuvent ainsi être ré-écrites d'une toute autre façon, du moment qu'elles implémentent l'interface `Observer` de Java.

3.5.2 La sélection et la réflexivité

Dans un langage n'utilisant pas la réflexivité, la sélection d'animation aurait impliqué l'inscription dans le code source de l'application de la liste de toutes les animations disponibles pour un type spécifique de variable. Le problème aurait été de maintenir à jour cette liste au fil du développement avec le risque de ne pas être cohérent avec les classes d'animations réellement disponibles pour l'utilisateur.

Face à la richesse, présente et à venir, des structures de données à animer, il devenait intéressant de se doter d'un mécanisme de sélection d'animation beaucoup plus souple.

Après des recherches documentaires, la réflexivité semblait apporter une solution souple et beaucoup moins contraignante, permettant à l'équipe de se concentrer sur le développement d'animation tout en ayant un mécanisme de sélection toujours au courant de toutes les animations disponibles.

La sélection d'animation se base sur deux mécanismes de filtrage :

- l'interface implémentée par l'animation qui permet de définir ainsi le type de variable à animer.
- une convention des noms des animations qui définit les propositions d'animation pour l'utilisateur.

Par exemple :

`AAAnimationForbool` est l'interface désignant les animations de booléens. Une classe implémentant cette interface est `AAAnimationForboolAsBit` et désigne une animation de booléen sous la forme d'un bit (0 ou 1).

3.5.3 L'animation

Chaque variable pouvant être représentée sous forme textuelle, des animations génériques ont pu être définies. Par exemple : `AAAnimationForAllAsString` est proposé à la sélection pour tout les types de variables, mais elle n'a été codée qu'une seule fois en implémentant l'interface `AAAnimationForAll`. Nous avons adapté ce principe pour les animations spécifiques aux types simples et aux types composés.

Pour l'animation des variables de type composé, nous évaluons la taille des animations de chaque élément de la variable puis nous lançons l'animation de chaque élément et nous terminons en réalisant l'animation de la variable.

Suite à une demande du « client », nous n'avons pas hésité à ajouter une fonctionnalité supplémentaire : la sélection de type double, comme les tableaux de tableaux à animer en tant que matrice. En fait, nous contrôlons le type des variables et dès qu'il y a des tableaux imbriqués, on ajoute une case à cocher devant le choix des animations de ces tableaux. Ainsi, l'utilisateur peut choisir d'animer ces tableaux séparément ou de les combiner. Dans ce dernier cas, la liste des animations disponibles est re-générée, l'utilisateur peut alors décider d'animer les tableaux en matrice.

Concrètement, nous avons du modifier légèrement la façon de nommer nos interfaces et nos classes d'animation. Par exemple :

- `AAAnimationForcombinedarrayarray` est l'interface désignant les animations de deux tableaux combinés.
- `AAAnimationForcombinedarrayarrayAsMatrix` est la classe qui désigne une animation de deux tableaux combinés sous la forme d'une matrice. Elle implémente l'interface précédente.

3.6 Le serveur

Le serveur tient un rôle primordial dans le bon fonctionnement de notre application. Nous avons installé sur notre machine un conteneur web, le framework Struts, ainsi qu'une base de donnée, afin que l'*Applet* Java que nous avons développé offre toutes les possibilités promises à l'utilisateur.

3.6.1 La base de données

La base de données est accessible à partir de l'*Applet* Java et permet à l'utilisateur de sauvegarder ses travaux et de les restaurer.

L'enregistrement s'effectue en trois étapes :

- sauvegarde de l'algorithme.
- sauvegarde des jeux de données.
- sauvegarde des paramètres d'animation.

La restauration résulte d'une recherche, à l'aide d'un ou plusieurs mots clés, dans la base de données. L'utilisateur peut alors choisir de charger un algorithme, des animations ainsi que des jeux de données correspondants.

Schéma de la base

La figure 3.7 donne un schéma précis de la base de données en spécifiant les objets qu'elle contient ainsi que le nom et le type de leurs attributs. D'autre part, il précise la gestion des clés étrangères.

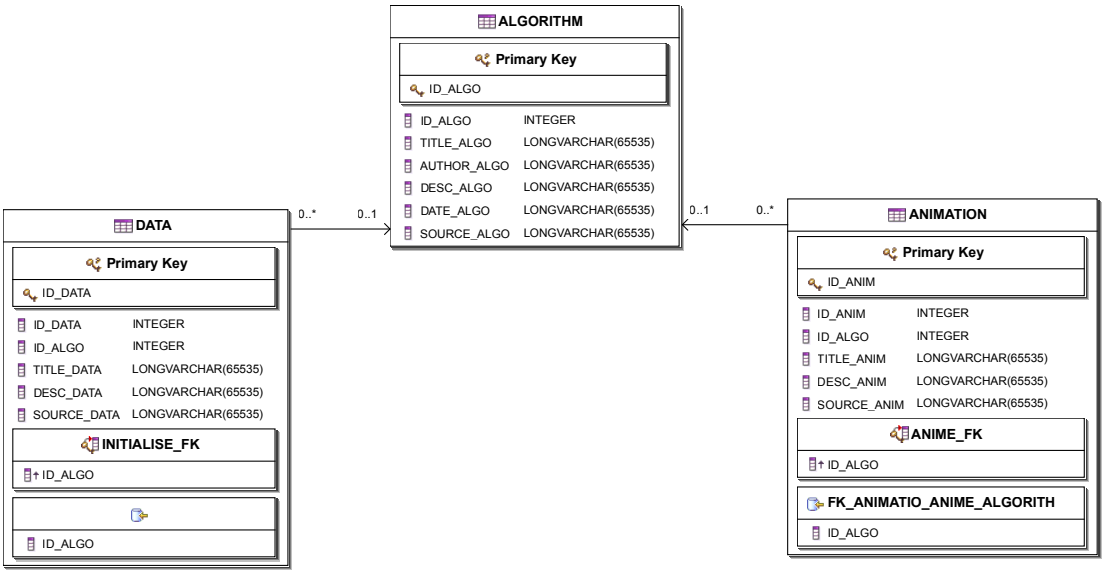


FIG. 3.7 – Schéma de la Base de données

3.6.2 Spécifications

La base de données est constituée de trois tables dîtes principales :

1. **ALGORITHM** table, contenant les informations sur les algorithmes, munie de 6 champs :
   - ID_ALGO, un entier qui s'incrémente automatiquement, clé primaire de la relation.
   - TITLE_ALGO, titre de l'algorithme.
   - AUTHOR_ALGO, nom de l'auteur de l'algorithme.
   - DESC_ALGO, spécifications de l'algorithme.
   - DATE_ALGO, date de sauvegarde de l'algorithme.
   - SOURCE_ALGO, un texte structuré contenant l'algorithme dans le langage du projet.
2. **DATA** table, contenant les informations sur les jeux de données, munie de 4 champs :
   - ID_DATA, un entier qui s'incrémente automatiquement, clé primaire de la relation.
   - ID_ALGO, clé étrangère du champ ID_ALGO de la table ALGORITHM.
   - TITLE_DATA, titre des jeux de données.
   - DESC_DATA, spécifications des jeux de données.
   - SOURCE_DATA, un texte structuré contenant les jeux de données.
3. **ANIMATION** table, contenant les informations sur les animations, munie de 4 champs :
   - ID_ANIM, un entier qui s'incrémente automatiquement, clé primaire de la relation.
   - ID_ALGO, clé étrangère du champ ID_ALGO de la table ALGORITHM.
   - TITLE_ANIM, titre de l'animation.
   - DESC_ANIM, spécifications de l'animation.
   - SOURCE_ANIM, un texte structuré contenant les paramètres de l'animation.

La base contient également trois autres tables dites temporaires, où sont stockées toutes les informations soumises par l'utilisateur afin de pouvoir réaliser une modération sur ce qui est sauvegardé. Ces tables sont structurées de la même manière que les principales, excepté un champ supplémentaire dans la table ALGORITHM_TEMP comportant l'adresse électronique de l'utilisateur déposant l'algorithme, afin qu'il soit prévenu au moment ou sont algorithme sera activé.

Les sources des données sont formatées en XML contenant le nom des variables, leurs valeurs et leurs types afin de réaliser un contrôle de type lors de la recherche des données pour un algorithme. Les sources des animations sont sous le même format mais contiennent le nom des variables, leurs

types et leurs paramètres d'animation pour savoir s'il y a ou non des animations combinées et pour connaître leurs types d'animation.

Plusieurs jeux de données et plusieurs animations peuvent correspondre à un seul algorithme. Cependant, un jeu de données ou une animation est spécifique à un seul algorithme. En particulier, il fait référence à des noms de variables de l'algorithme.

Nous utilisons le *trigger* ON DELETE CASCADE, sur les clés étrangères ID_ALGO des tables DATA et ANIMATION, afin de garantir l'intégrité référentielle avec la table ALGORITHM. Ainsi, si un algorithme est effacé, les jeux de données et les paramètres d'animation correspondant sont eux aussi supprimés.

Choix du SGBD

Nous travaillons sous Unix et comme nous sommes dans un cadre de projet universitaire, nous choisissons un SGBD existant sous une licence non commerciale. Nous avons donc le choix entre Borland, PostGreSQL et MySQL. Mais Borland présente l'inconvénient que certaines requêtes JDBC comme les ResultSet ne sont pas autorisées. Comme dit précédemment, on formate les données de la base en XML, or PostGreSQL ne prend pas en charge le XML, ni les services Web. En revanche, les seuls désavantages de MySQL sont qu'il ne supporte pas les *triggers*, ni les procédures stockées sauf pour des tables de type InnoDB.

Nous avons donc choisi MySQL, car nous ferons uniquement de la sauvegarde et de la restauration de données simples. De plus, nous utilisons des tables de type InnoDB afin de pouvoir incorporer la suppression en cascade. Enfin, nous avons la possibilité de gérer le chargement du *driver* et la connexion à la base de données dans le fichier de configuration de Struts, nous évitant ainsi de le réaliser dans chaque classe communiquant avec les données.

3.6.3 Struts

La plate-forme Struts implémente le modèle MVC II qui permet une construction rapide d'application Web facilement extensible et maintenable. Le modèle MVC II consiste à bien séparer les *objets métiers* (qui réalisent une tâche) des objets IHM (présentation et restitution d'informations à l'utilisateur (dans notre cas, il s'agit de l'*Applet*). Cela est mis en œuvre par le biais d'un *modèle*, d'une *vue* et d'un *contrôleur* (MVC). Le modèle correspond au noyau de fonctionnement de l'application et contient donc tous les objets métiers (classes, objets...) La vue est l'interface de communication entre l'utilisateur et l'application. Il est possible, à l'aide de tags JSP, d'écrire des pages dynamiques interprétées par le serveur d'exécution. Le contrôleur sert de liaison entre la vue et le modèle. Il gère le déroulement de la navigation. Il est constitué d'un fichier XML (`struts-config.xml`) qui décrit la correspondance entre les requêtes de l'utilisateur et les actions associées ; ainsi que des fichiers définissant les traitements effectués par ces actions. Enfin, Struts gère également la validation et facilite donc la gestion d'erreurs. Pour plus d'informations on se reportera au *Jakarta Struts pocket reference* [8].

Rôle de Struts

Nous avons délibérément choisi de réaliser les phases de compilation et d'exécution côté client. En effet, étant donné les possibles erreurs syntaxiques ou autres dans l'algorithme à tester et que la forme compilée ne doit pas persister, il ne nous a guère semblé opportun d'envoyer ces résultats intermédiaires au serveur ou de le surcharger avec ces tâches. De plus, nous tenons à limiter au maximum les échanges entre le client et le serveur afin de toujours garantir une bonne réactivité.

Notre utilisation de Struts s'est donc essentiellement restreinte à transférer les données de notre application à la base de données. La majorité des services rendus pas Struts consistent donc à sauvegarder des données ou des animations de l'*Applet* vers la base de données, ou à les restaurer depuis le serveur. En ce sens, Struts a un rôle d'interface entre l'utilisateur (l'*Applet*) et la base de données.

Par ailleurs la gestion de certaines pages du site Web nécessitant un accès à la base de données (pages d'administration, pages de recherche) sera basée sur Struts. Les rôles de Tomcat nous permettent de restreindre l'accès aux pages d'administration grâce à un mot de passe : on définit un rôle *admin*, qui sera le seul à pouvoir accéder à ces pages.

Enfin l'envoi de courrier électronique est aussi faite avec Struts, par l'intermédiaire de l'API *JavaMail*.

Communication avec la base de données et l'*Applet*

Struts intègre très bien la connexion à une base de données MySQL. Cela se fait grâce aux *pools de connexion DBCP*[1] de *Jakarta Commons*[2]. Les sources de données (ou *DataSource*) nous permettent d'obtenir facilement une connexion à la base de données sans avoir à écrire à chaque fois du code redondant. Dans notre cas, la base de données a une structure relativement simple, nous n'utiliserons donc qu'une seule source de données.

Toutefois, du fait que tous les accès au serveur se font depuis une *Applet*, notre utilisation de Struts ne reflète pas exactement le schéma classique d'une application Web utilisant Struts, qui consiste a faciliter le dialogue avec l'utilisateur à travers des formulaires HMTL. Si les communications de l'*Applet* vers le serveur sont relativement simples à gérer (ce sont de simples requêtes à construire), le contraire n'est pas aussi évident : l'*Applet* doit pouvoir comprendre les données retournées par le serveur. Pour cela nous avons dû adopter pour chaque requête un format de texte structuré, construit par le serveur, et compris par l'*Applet*.

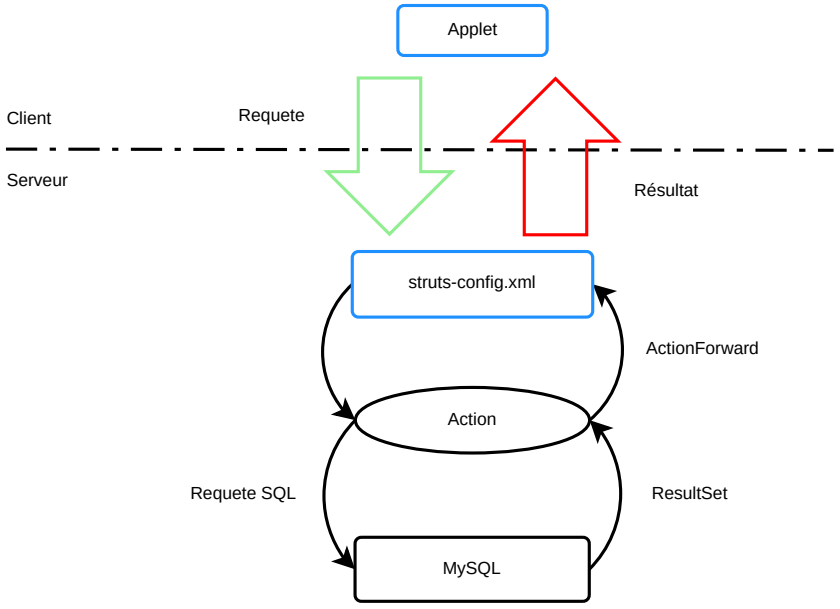


FIG. 3.8 – Communication avec l'*Applet* et la base de données

La figure 3.8 explique de manière un peu plus technique la façon dont ces communications seront réalisées. Quand le client effectue sa requête, une Action à exécuter est déterminée à partir du fichier `struts-config.xml`, qui contient une sorte de table associative entre des noms d'actions et des classes. C'est cette action qui exécute la requête à la base de données et récupère le résultat dans un objet de la classe `ResultSet` (en fait, certaines actions comme celles de mise à jour n'auront pas de résultat à récupérer).

[1] DBCP, DataBase Connection Pools, est une méthode de gestion des connexions à une base de données SQL offerte par Struts.

[2] Jakarta Commons est un sous-projet de Jakarta visant à aider à faire des composants Java réutilisables.

L'*ActionForward* (qui est utilisée par l'action pour dire quelle vue devra être sélectionnée) sera choisie selon le déroulement de l'exécution de l'action. Si une exception est déclenchée, on choisit par exemple une vue *error*, sinon on choisit une vue *success*. Cette vue est à nouveau propagée grâce au fichier `struts-config.xml` jusqu'à l'*Applet* pour obtenir le résultat final.

Enfin, l'*Applet*doit pouvoir distinguer une réponse positive d'une réponse d'erreur. Ce problème est résolu par la création d'un protocole de communication entre Struts et l'*Applet* permettant à l'*Applet* de distinguer sans ambiguïté les deux possibilités.

Gestion des erreurs

Nous n'avons pas utilisé pleinenemt la gestion d'erreur fournie par Struts. En effet, la compilation se faisant côté client, toutes les données envoyées au serveur doivent être valides. De plus, comme le code côté client s'exécute dans une *Applet*, la gestion des erreurs côté client est très simple à gérer. Les seules erreurs pouvant survenir côté serveur seront des erreurs liées à la base de données (connexion, mise à jours...) Par ailleurs, Struts supporte les exceptions Java, ce qui rend les gestion des erreurs très simple : à chaque fois qu'une action lance une exception, celle-ci est attrapée automatiquement, grâce à la définition d'exceptions globale. Il est possible de définir un *ExceptionHandler* (chargé de gérer l'exception) pour un type d'exception donné. Dans notre cas toutes les exception sont gérées de la même manière, on définit donc un seul *ExceptionHandler*, pour le type d'exceptions le plus général (`java.lang.Exception`). Cependant on peut très bien imaginer par la suite une gestion plus spécialisée des erreurs.

Le *package database*

Pour que l'*Applet* puisse communiquer avec le serveur, il est nécessaire de fournir un certain nombre de classes et méthodes qui seront chargées de faires les requêtes auprès du serveur et d'interpréter ses réponses. C'est le rôle du *package database*, qui a donc pour but d'abstraire l'accès au serveur, pour que les autres parties de l'application puissent facilement effectuer les opérations dont elles ont besoin, sans avoir à se soucier que la communication se fait avec un serveur distant (on peut d'ailleurs imaginer qu'un jour ce ne soit plus le cas). Ce *package* a ce seul rôle, il ne doit donc dépendre d'aucune autre partie de l'application.

Le *package database* fournit les classes encapsulant les données pour les trois types d'objets présents dans la base de données : les algorithmes, les jeux de données et les animations, ainsi qu'une fonction de recherche d'algorithme par mot-clé.

3.6.4 Mise en œuvre du site Web

Il est absolument nécessaire de passer par une étape d'analyse avant de commencer le développement d'une application. Il en va de même pour la création d'un site Web. Spécifier le choix des technologies, des formats, la manière de générer du contenu, l'architecture sont autant de choses qui vont dans ce sens.

Format des pages

Dans un but de maintenance évident, le fond et la forme du site sont séparés, grâce à l'utilisation des deux formats de documents suivants :

(X)HTML Il s'agit d'un méta-langage permettant la création de document, de manière organisée, logique, hiérarchisée. Successeur du HTML, le XHTML est à la fois compatible avec les navigateurs d'hier et avec le XML, sa rigueur en fait un puissant langage de structuration de contenu, laissant le soin de la présentation aux CSS. Nous avons opté pour la version 1.0 Strict, dont la Définition de Types de Documents (DTD) est la suivante :

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
```

CSS Les feuilles de style en cascades, ou *cascading style sheet* s'adaptent à tout document XHTML pour mettre en forme le contenu. Elles permettent de proposer des mises en page différentes, que ce soit dans un soucis d'ergonomie ou pour adapter le contenu au support (écran d'ordinateur, navigateur alternatif, imprimante, etc.), sans modifier le contenu de la page.
Nous avons opté pour la version 2.0.

Pour une meilleure gestion du site, de son contenu, il est probable qu'il faille utiliser un langage exécuté côté serveur, afin de simplifier l'écriture des pages Web. Par exemple, afin d'assurer une maintenance correcte du site, les pages Web utilisent un système de *templates*. Nous avons opté pour la JSP[3].

Pour une meilleure gestion du site, et de son contenu, nous ne créerons pas de pages dites « statiques ». Les pages sont générée sur le serveur avant d'être envoyées chez le client. Pour cela nous utilisons les *Java Server Pages* (JSP).

Les JSP sont un standard permettant de développer des applications Web interactives. C'est-à-dire qu'une page Web JSP (repérable par l'extension .jsp) a un contenu pouvant changer selon certains paramètres (des informations stockées dans une base de données, les préférences de l'utilisateur...) Les JSP sont intégrables au sein d'une page Web en XHTML à l'aide de balises spéciales. Celles-ci permettent au serveur Web de savoir que le code compris à l'intérieur de ces balises doit être interprété, afin de renvoyer du code XHTML au navigateur du client.

Conformité

Afin de se plier aux recommandations du W3C[4] (*World Wide Web Consortium*), les pages générées ainsi que les feuilles de styles proposées passe avec succès les tests de validation présents sur le site du W3C. Pour cela, nous respectons la syntaxe recommandée par le W3C.

De plus, afin d'être en conformité avec les recommandations du W3C en matière d'accessibilité, nous nous efforcerons de les respecter au maximum, grâce à des mesures simples mais efficaces (attribut alternatif pour toute image, raccourcis claviers fonctionnels pour la navigation dans le site, respect d'une sémantique XHTML compréhensible par les navigateurs textuels, usage des DIV et non des tableaux pour la mise en forme, etc.)

Nom de domaine

Une adresse internet de la forme http ://bidibul.ifsic.univ-rennes1.fr/animalgo/web/en/index.jsp n'est pas facile à retenir, ou à communiquer, loin s'en faut. Nous avons donc crée une redirection de domaine. Nous avons acheté le nom de domaine www.animalgo.org, et l'avons redirigé vers l'adresse sus-citée. Ainsi, nous avons pu créer une adresse mail simple et facile à retenir : contact@animalgo.org. Adresse mail qui servira à contacter les personnes ayant proposé le dépôt d'un algorithme, d'options d'animation, mais aussi à être contacté par toute personne ayant des questions aux sujet de ce projet.

Choix des langues

Nous proposerons deux versions du site Web. Une en anglais, une en français. Le choix se fait dès la page d'accueil, mais aussi sur chacune des page du site, afin de faciliter la navigation. Ce choix nous a amené à réaliser deux versions différente du site Web. Ainsi, nous plaçons à la racine du site Web deux dossier, chacun correspondant à une langue différente. Ainsi, les liens www.animalgo.org/en/ et www.animalgo.org/fr/ dirigent respectivement le visiteur vers la version anglais et française.

[3] Pour plus d'information, consulter [9]

[4] Le World Wide Web Consortium, abrégé W3C, est un consortium fondé en octobre 1994 pour promouvoir la compatibilité des technologies du Web, telles que HTML, XHTML, XML, CSS, PNG...Le W3C n'émet pas des normes, mais des recommandations. Sa gestion est assurée conjointement par le Massachusetts Institute of Technology (MIT) aux États-Unis, le European Research Consortium for Informatics and Mathematics (ERCIM) en Europe et l'Université Keio au Japon.

Architecture du site

Afin de s'assurer une maintenance correcte du site, l'architecture même du site se doit d'être définie. La figure 3.9 montre comment la racine du serveur est organisée. Pour des raisons de lisibilité, un seul fichier JSP figure (index.jsp) par dossier. L'architecture du dossier *FR* est totalement identique à celle du dossier *EN*.

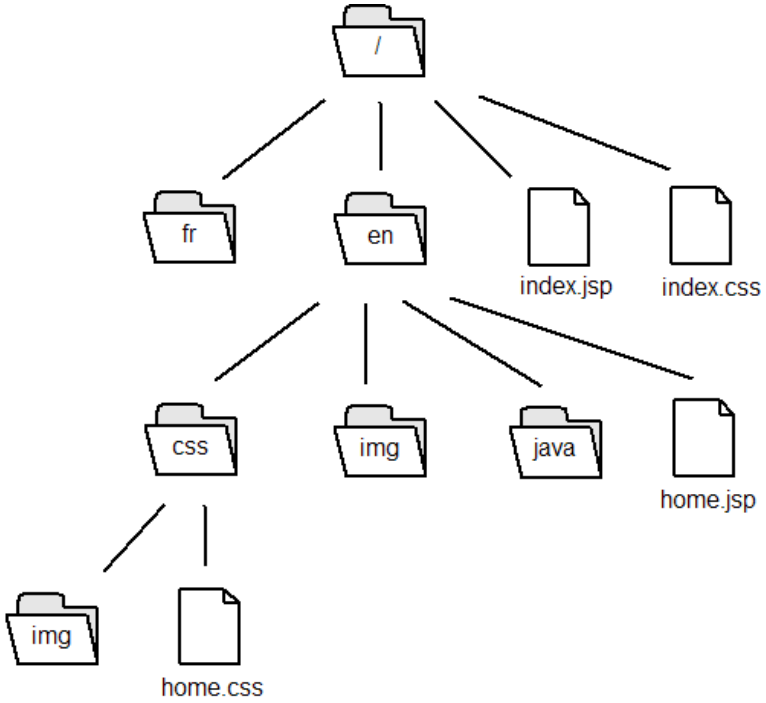


FIG. 3.9 – Architecture du site Web

3.7 Méthodes de développement et historique

Après l'ensemble des spécifications de l'application, nous allons présenter le développement en lui même. Même si pour des raisons de commodité le projet de Master 1 distingue une phase d'analyse et une phase de développement, nous avons adopté un cycle de vie en spirale qui a permis de commencer le développement pendant la phase d'analyse en particulier pour réaliser des expériences. Nous avons aussi adopter une organisation qui laisse une grande place au « client ». Les deux options nous font nous rapprocher des méthodes de développement dites agile. Nous allons présenter la méthode de développement appliquée, les rôles principaux au sein du projet et pour finir un historique des tâches effectuées par l'équipe de développement.

3.7.1 Organisation de l'équipe

La composition de l'équipe a été légèrement modifiée au début de la phase de développement. En effet, trois personnes ont quitté le projet à la fin de la phase d'analyse. Nous n'étions donc plus que huit durant la phase de développement. Il a fallu procéder à une redistribution des responsabilités. Quatre « postes » étaient à pourvoir :

Responsable du projet Il avait la responsabilité générale du projet. Il était par exemple chargé d'organiser, et de coordonner le travail, mais surtout de contrôler l'intégration des différents composants.

Administrateur machine Il était responsable de l'installation et de la maintenance des machines, du serveur et du *framework* Struts.

Administrateur Web Il avait la responsabilité générale du site Web, de son avancement et de son développement.

Responsable du rapport Il avait pour tâche de centraliser toute la documentation produite afin de créer le rapport au fur et à mesure.

3.7.2 Méthodes de travail

Développement agile Les méthodes de développement agile s'opposent aux méthodes de développement dites classiques (cycle en V). Ces méthodes sont basées sur des cycles de développements itératifs assez courts. On obtient ainsi rapidement une ébauche de l'application finale. Le client peut alors indiquer à la fin de chaque cycle si l'application répond ou non à ses besoins. Les programmeurs s'adaptent ainsi rapidement aux attentes du client. La satisfaction du client est un élément majeur dans la « philosophie » de ces méthodes.

Nous ne sommes pas une équipe de programmeurs expérimentés, Nous travaillons dans le cadre d'un projet universitaire. Il n'a donc pas été possible d'appliquer tous les principes de méthodes dites agiles. Cependant, nous avons tout de même pu retenir certains axiomes de méthodes telles que *eXtreme Programming* ou *Rational Unified Processus* qui nous ont permis d'avoir une organisation efficace :

Projet piloté par le client Le rôle du « client » a été joué par Mr Olivier Ridoux. Des réunions entre l'équipe de développement et le client étaient organisées toutes les semaines. Il indiquait à la fin de chaque itération s'il était satisfait ou non et décidait des modifications à apporter. Mr Ridoux a surtout eu un rôle de conseiller et nous indiquait les points sur lesquels il était préférable de se concentrer.

Travail en équipe La programmation s'est faite principalement en binôme. Il y avait 2 programmeurs en face du poste de travail. Pendant qu'une personne programmait, l'autre vérifiait et posait des questions si nécessaire. Le code produit ainsi est de bonne qualité et compris par au moins deux personnes. Afin de faciliter le travail en équipe et d'obtenir un résultat homogène, nous avons utilisé des standards de codage (exemples : normes de Sun, Javadoc...). La programmation en binôme a également permis de faire circuler les connaissances (une personne expérimentée dans un domaine particulier était en général avec une autre moins expérimentée).

Intégration permanente Les cycles de développement itératifs nous ont obligé à livrer des versions fonctionnelles régulièrement. Pour ce faire, nous avons intégré les différents composants de manière fréquente tout au long du développement. Cette intégration permanente a permis de toujours avoir une application qui fonctionne et nous a évité des mauvaises surprises que nous aurions pu rencontrer à la fin du projet si nous avions décidé d'intégrer tous les composants au dernier moment. Les erreurs détectées ont ainsi pu être corrigées assez rapidement.

Refactoring Qu'est ce que le *Refactoring* ? Il s'agit principalement de revoir le code source tout au long du développement afin d'en améliorer la qualité. Au fur et à mesure que l'on rajoute des fonctionnalités, il peut arriver que l'on se rende compte que certaines parties du code source ont besoin d'être modifiées car elles ne conviennent plus (découverte de doublons, de cycles de dépendances...). Il a donc fallu à certains moments modifier le code source, rajouter des interfaces et factoriser le code afin de garder une architecture simple et évolutive. Par exemple, lors de l'ajout des types composés, nous avons appliqué le patron de conception singleton et rajouté des interfaces.

Les quelques axiomes des méthodes de développement agile dont nous nous sommes inspirés ont surtout permis de définir un cadre méthodologique pour ce qui était pour de nombreuses personnes du groupe le premier projet de développement en équipe. Il n'a pas été difficile de s'adapter à ce mode de fonctionnement, cela demandait seulement un peu de rigueur supplémentaire. La bonne organisation de notre équipe a sûrement contribué au succès de notre projet.

Méthodologie de test Le développement agile peut se baser sur le développement piloté par les tests. En effet, ce dernier préconise de construire les classes de tests avant de développer les classes de l'itération en cours, forçant ainsi les développeurs à bien analyser les différentes parties de l'application. Le programmeur est ainsi mis au défi d'écrire une classe fonctionnelle répondant exactement aux spécifications de l'itération. Enfin cela apporte un gain de temps lors des phases d'intégration et de tests.

Il existe diverses méthodologies de test qui interviennent à divers degré dans le développement de l'application. On notera ici les principales méthodologies utilisées au cours du développement.

Test structurel (Test boîte blanche) Ce type de test nécessite que l'outil de test ait accès au code source. Il réalise une analyse structurée du code.

Test fonctionnel (Test boîte noire) Dans ce cas l'outil vérifie à l'aide d'un *oracle* de manière *externe* que les méthodes vérifient la spécification sur les cas généraux et pathologiques

Test unitaire Lors d'un test unitaire on se concentre sur l'élément de base de l'application : la *classe*.

Test d'intégration (Test boîte noire) Le test d'intégration est fortement basé sur le test unitaire. Il consiste à remplacer une partie des *bouchons de test* par d'autres classes déjà testées unitairement. Ainsi on contrôle l'interaction entre les classes. Le test d'intégration intervient généralement dans le dernier quart de l'itération.

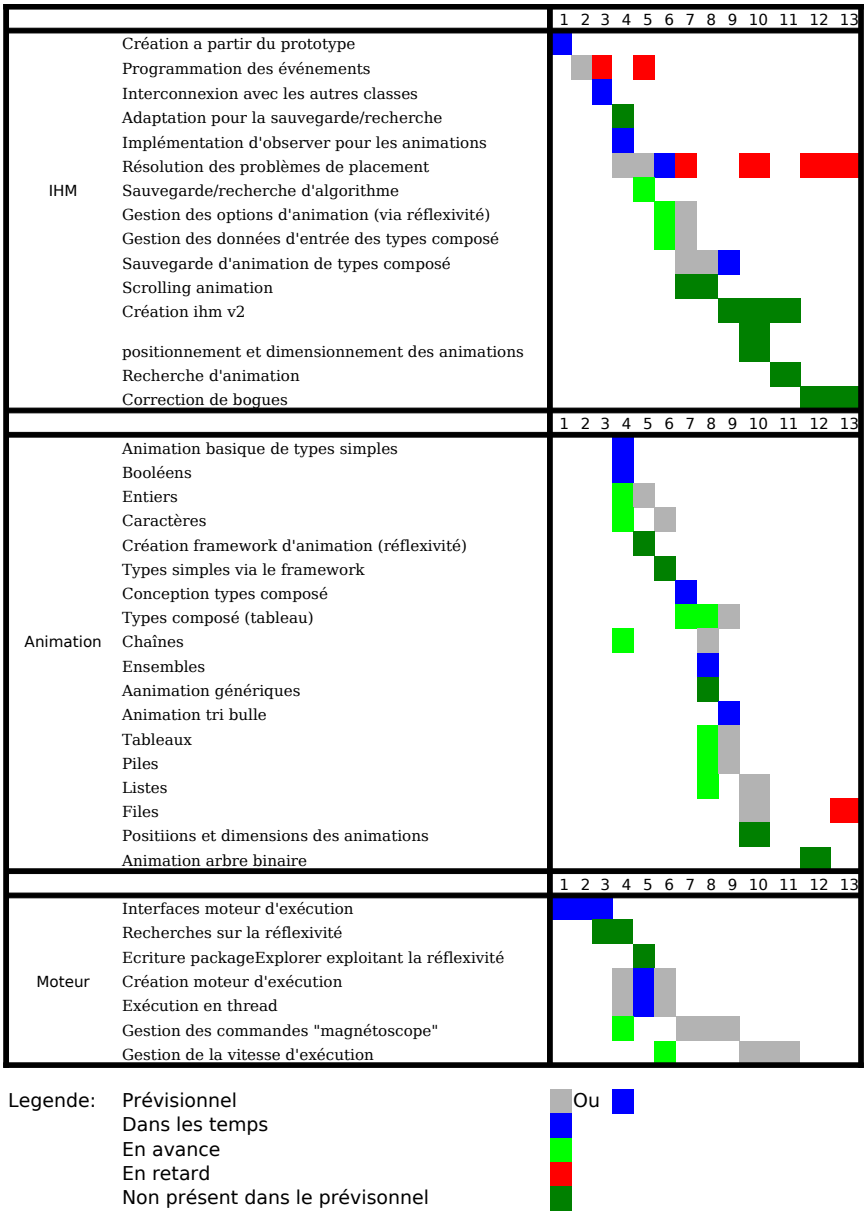
Test système (Test boîte noire) Les tests système ont pour but de vérifier le bon fonctionnement de l'application entière au cours de scénarii prédéfinis avec des jeux d'essais assurant une couverture maximale (voire complète)

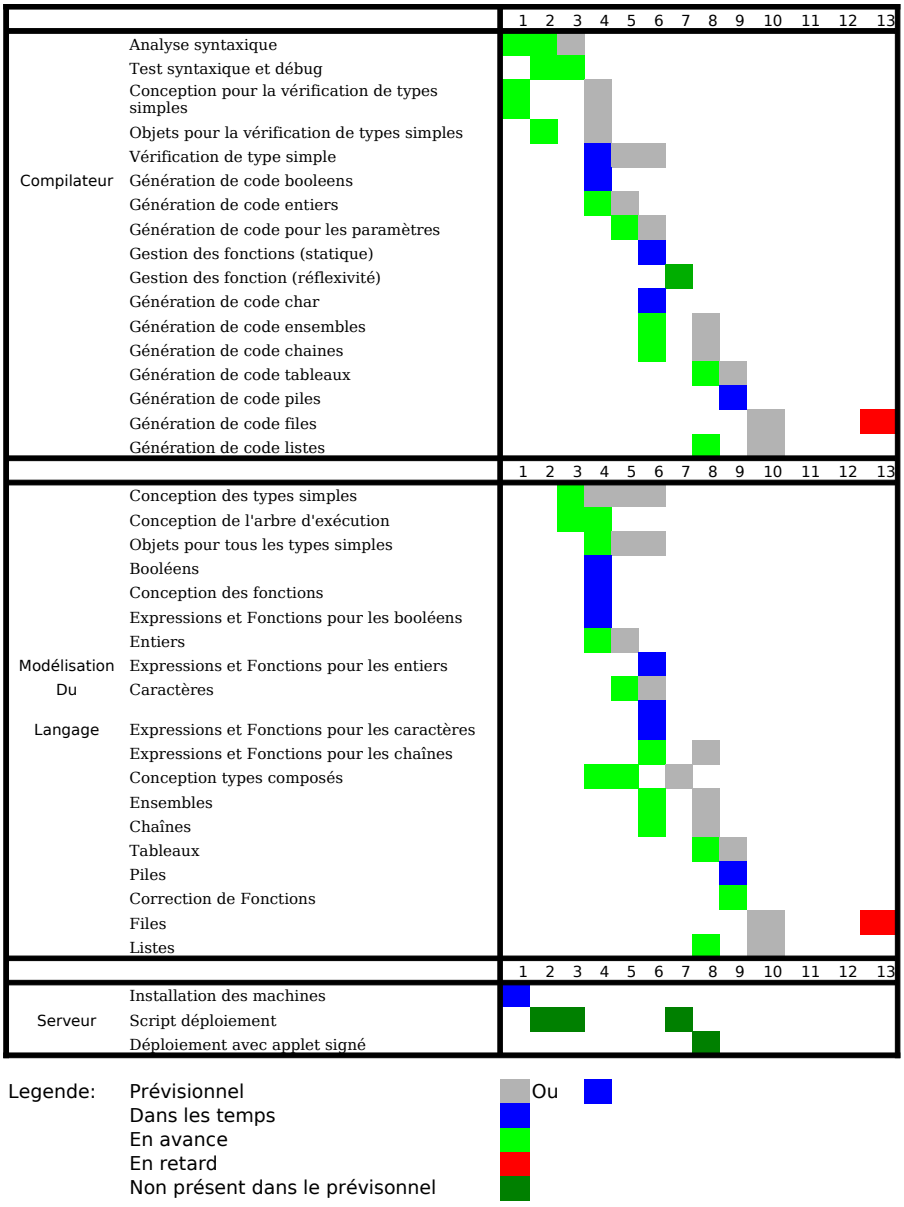
3.7.3 Historique de développement

Nous allons vous présenter ici le planning qui a été effectivement suivi par l'équipe au cours de la phase de développement. Il vous est présenté sous la forme d'un tableau synthétique (figure 3.10) qui reprend les grands axes de l'application (IHM, animation, compilation...) ainsi que les étapes de son développement. Est absent de ce tableau les tâches préliminaires et la réalisation du prototype.

Pour chaque partie nous avons fait un comparatif entre le planning prévisionnel et effectif afin de dégager d'un seul coup d'œuil ce qui a été fait en avance (en vert clair), en retard (en rouge), dans les temps (en bleu) et ce qui n'avais pas été prévu (en vert foncé). Lorsque les tâches ont été effectuées en avance ou en retard, vous retrouverez en gris le prévisionnel.

Dans l'ensemble, nous pouvons remarquer que la plupart des tâches on été faites soit dans les temps, soit en avance, et ce grâce au prototype réalisé pendant la phase d'analyse et aux efforts personnels de chacun. Néanmoins, certaines fonctionnalités ou concepts n'avaient pas été envisagées tel que l'utilisation de la réflexivité et la création d'un script de déploiement... De tels imprévus on nécessité de la part de l'équipe une forte recherche documentaire pour répondre aux problèmes et diverses demandes du client au cours des itérations. Cependant la bonne modularité de l'application nous a permis d'intégrer ces nouvelles fonctionnalités sans revoir l'architecture globale de l'application.


1 2 3 4 5 6 7 8 9 10 11 12 13
Création a partir du prototype
Programmation des événements
Interconnexion avec les autres classes
Adaptation pour la sauvegarde/recherche
Implémentation d'observer pour les animations
Résolution des problèmes de placement
IHM
Sauvegarde/recherche d'algorithme
Gestion des options d'animation (via réflexivité)
Gestion des données d'entrée des types composé
Sauvegarde d'animation de types composé
Scrolling animation
Création ihm v2
positionnement et dimensionnement des animations
Recherche d'animation
Correction de bogues
1 2 3 4 5 6 7 8 9 10 11 12 13
Animation basique de types simples
Booléens
Entiers
Caractères
Création framework d'animation (réflexivité)
Types simples via le framework
Conception types composé
Types composé (tableau)
Animation
Chaînes
Ensembles
Aanimation génériques
Animation tri bulle
Tableaux
Piles
Listes
Files
Positiions et dimensions des animations
Animation arbre binaire
1 2 3 4 5 6 7 8 9 10 11 12 13
Interfaces moteur d'exécution
Recherches sur la réflexivité
Ecriture packageExplorer exploitant la réflexivité
Moteur
Création moteur d'exécution
Exécution en thread
Gestion des commandes "magnétoscope"
Gestion de la vitesse d'exécution
Legende:
Prévisionnel
Ou
Dans les temps
En avance
En retard
Non présent dans le prévisonnel


1 2 3 4 5 6 7 8 9 10 11 12 13
Analyse syntaxique
Test syntaxique et débug
Conception pour la vérification de types simples
Objets pour la vérification de types simples
Vérification de type simple
Compilateur
Génération de code booleens
Génération de code entiers
Génération de code pour les paramètres
Gestion des fonctions (statique)
Gestion des fonction (réflexivité)
Génération de code char
Génération de code ensembles
Génération de code chaines
Génération de code tableaux
Génération de code piles
Génération de code files
Génération de code listes
1 2 3 4 5 6 7 8 9 10 11 12 13
Conception des types simples
Conception de l'arbre d'exécution
Objets pour tous les types simples
Booléens
Conception des fonctions
Expressions et Fonctions pour les booléens
Entiers
Modélisation
Expressions et Fonctions pour les entiers
Du
Caractères
Langage
Expressions et Fonctions pour les caractères
Expressions et Fonctions pour les chaînes
Conception types composés
Ensembles
Chaînes
Tableaux
Piles
Correction de Fonctions
Files
Listes
1 2 3 4 5 6 7 8 9 10 11 12 13
Installation des machines
Serveur
Script déploiement
Déploiement avec applet signé
Legende:
Prévisionnel
Ou
Dans les temps
En avance
En retard
Non présent dans le prévisonnel

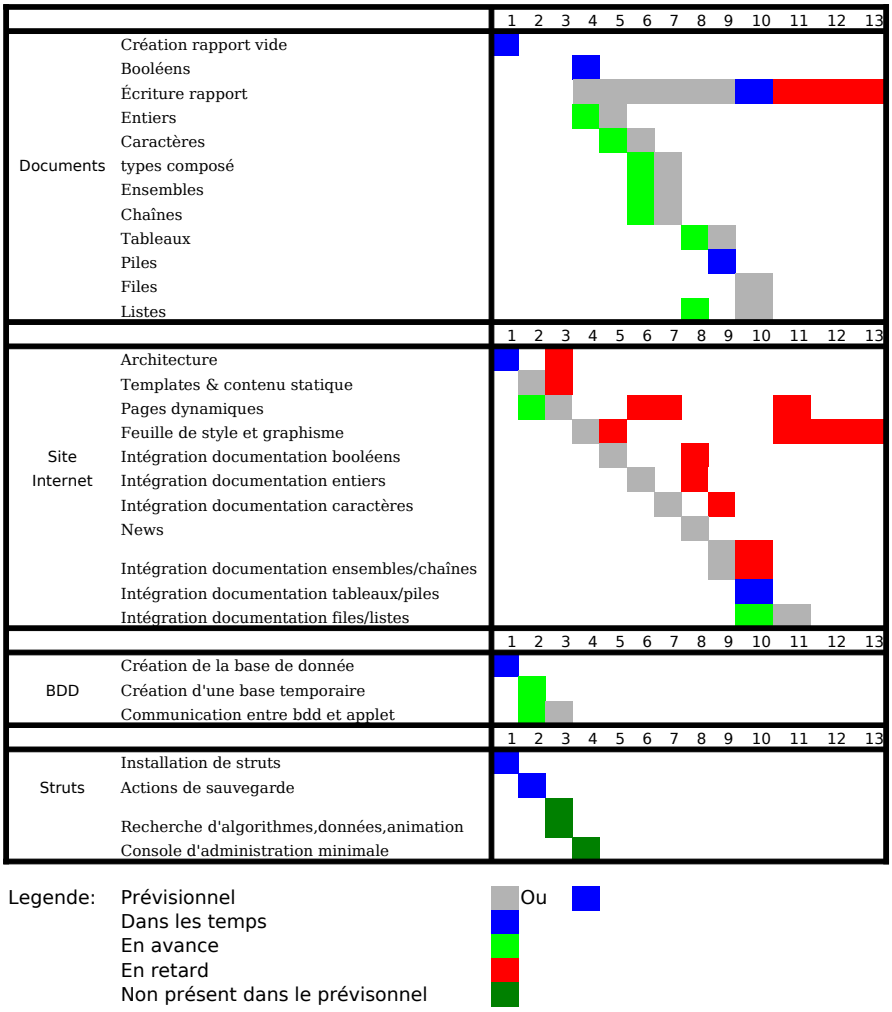


FIG. 3.10 – Historique de développement

Chapitre 4

Validation

La validation est une étape importante, sinon nécessaire dans le cadre d'un projet. Elle permet de confronter l'accomplissement du logiciel fini aux spécifications initiales. La méthode de développement agile noua a permis de rester proche de ce que souhaitait le client tout au long de ce projet. Nous allons présenter ici les aspects fonctionnels et les possibilités d'utilisation de notre application.

4.1 Fonctionnalités

4.1.1 Le langage

La syntaxe du langage L'ensemble du langage spécifié dans la grammaire 1 est fonctionnel. Il permet la saisie de la plus part des algorithmes retenu pour notre challenge et bien d'autres qui pourront être illustrés pour une meilleure compréhension algorithmique.

Un algorithme peut prendre en entrée des paramètres mais le langage n'autorise pas de modifications de ceux-ci lors de l'exécution ; c'est une distinction entre les paramètres et les variables souhaitée par le « client ».

Conformément aux spécifications, le langage n'autorise pas les algorithmes récursifs. Néanmoins, de tels algorithmes peuvent être dérecursivés notamment avec l'utilisation d'une pile qui est un type natif de notre langage.

Le langage ne permet pas une programmation multi processus, il ne possède pas de primitive de synchronisation. Ceci reste envisageable dans une version future et la principale notion à ajouter serait un ordonnanceur de tâches.

Les types de données Le langage offre à l'utilisateur un nombre important de types natifs.

Des types simples d'une part, tels que les booléens, les entiers ($\mathbb{Z}$ sans débordement), les caractères et les chaînes de caractères (ASCII). Les réels pourront être ajouté dans une version future en surchargeant les opérations manipulant les entiers.

D'autres part, des types composés limités à $2^{31} - 1$ éléments et pouvant s'imbriquer (exemple : int set set représente un ensemble d'ensembles d'entiers). L'utilisateur a à sa disposition les ensembles, les listes, les files, les piles et les arbres binaires ces derniers permettant de gérer des structures simples. Tous ces types ont des animations prédéfinies disponibles pour l'utilisateur. L'utilisateur peut définir et animer les structures de données de son choix en utilisant les constructeurs fournis mais, il ne peut pas composer de nouveaux constructeurs. La modularité de l'application permet l'ajout ultérieur de type de données et d'animations sans trop de difficultés.

Les primitives du langage La grande partie des primitives du langage sont des fonctions. Nous ne ferons pas une liste détaillée de celles-ci dans cette partie. Elles offrent à l'utilisateur une souplesse dans la programmation des algorithmes notamment sur la manipulation des types de

données et sur les opérations de calculs. Toutes ces fonctions retournent un résultat et sont des expressions. Elles ne font pas d'effet de bord.

4.1.2 Animation

La recherche d'animations dans la base de données permet de sélectionner automatiquement l'animation sauvegardée pour chaque variable dans la liste d'animations correspondante.

L'animation d'une variable de type composé prend en compte la longueur et la largeur des animations des éléments qu'elle contient.

La sélection d'animation des tableaux à plusieurs dimensions sous forme de matrice fonctionne très bien à l'aide de case à cocher. La technologie employée ne fonctionne que pour ce type spécifique de données, mais peut-être facilement étendue pour d'autres combinaisons de types.

L'ensemble des animations énoncées durant la phase d'analyse du projet ont été développées, nous avons même alimenté un peu plus la liste de ces animations. (ex : matrice, arbre...)

les techniques employés permettent d'envisager des animations bien plus poussées a l'aide de composant d'animation bien plus riches. chaque animation étant un composant graphique `JPanel` et la sélection d'animation n'imposant qu'une convention sur leur nom, une animation peut etre écrite a l'aide de `Java3D` et directement intégré au sein de l'application. ainsi on peut envisager des animations bien plus riches avec des commandes de manipulation graphique (rotation, zoom, déployer un noeud d'arbre ou le réduire). C'est dans ce cas précis que la conception modulaire que nous nous sommes imposé se révèle payante et la reprise du projet et son extension en est grandement facilité.

4.1.3 La base de données

Les tables de la base de données permettent de stocker les algorithmes, les jeux de données et les options d'animation ainsi que diverses informations décrivant ces objets (titre, description ...).

La sauvegarde des jeux de données et des animations est relativement précise car elle inclut les types. Il est donc possible de vérifier que les types correspondent bien au moment de restaurer ces données. En effet, puisqu'on autorise de modifier un algorithme existant, il se peut que certaines données ou animations deviennent invalides. Cette information est transmise à l'utilisateur dans le résultat de sa recherche : les données et animations valides sont suivies du mot *safe* et celle qui sont invalides sont suivies du mot *unsafe*.

Deux scripts ont aussi été prévus pour que nous puissions rapidement sauvegarder le contenu de la base de données ou le restaurer, en prévention d'éventuelles pannes.

4.2 Utilisabilité

4.2.1 La compilation d'un algorithme

La compilation peut dans certains cas prendre du temps notamment lors de la première compilation de l'algorithme. Lors de cette phase des erreurs de différentes natures peuvent être générées. La compilation s'arrête dès la première erreur rencontrée et affiche alors un message significatif dans la console. Le message contient le numéro de la ligne correspondante et la nature de l'erreur. Les principales erreurs signalées sont les erreurs de type (par exemple : affectation d'une valeur booléenne à une variable de type entier), les affectations de paramètres, les doubles déclarations de variables ou bien les identifiants non déclarés, de même qu'un identifiant ne faisant pas référence à une fonction offerte par le langage sur les types implémentés et bien évidemment les erreurs de syntaxe. Grâce au numéro de la ligne indiquant la position du curseur, l'utilisateur peut facilement retrouver son erreur dans le texte de l'algorithme.

4.2.2 Capacité d'exécution

Notre application pose certaines limitations à l'utilisateur. La taille de tous les types composés est limitée à $2^{31} - 1$, cette borne n'est pas vraiment limitative pour l'utilisateur dans le cadre de l'animation d'algorithme. De plus, certaines opérations peuvent prendre du temps, comme par exemple le calcul d'une puissance élevé d'un grand entier. Enfin, l'affichage de nombres très grands peut demander du temps selon le type d'animation sélectionné. Par exemple l'affichage sous forme de chaîne de caractères d'un grand entier prend quelques secondes. Cette estimation est relative car l'exécution est faite côté utilisateur et dépend donc de la machine de celui-ci.

4.2.3 Animation

Nous avons animé à l'aide de Java2D tout les types de données proposés sous diverses formes, mais la qualité de l'animation finale dépend grandement d'une sélection judicieuse de la part de l'utilisateur. Ce premier lot d'animations représente un peu plus que le minimum nécessaire pour couvrir les structures de données mises en œuvre. Cependant cette couverture minimale est loin d'être minime, car elle offre la possibilité d'animer de façon clair tout algorithme de tri de tableau, de calcul, a base d'opérations sur les ensembles, de calcul matriciel, etc. Bien entendu tout ne peut être animé sous une forme confortable, et c'est a l'utilisateur de jouer avec l'application afin de découvrir quelle forme d'animation est la plus adaptée a ses besoin. Ce dernier se doit d'être raisonnable quand a l'envergure des structures de données qu'il anime car dans le temps qui nous était imparti, cette couverture minimale était déjà un challenge en soit.

Les animations ont les défauts suivants : L'animation des entiers sous forme de ligne horizontale ou verticale est de longueur égale à la valeur de la variable en pixels, ce qui peut entraîner des débordements pour des entiers assez importants. Toutes les animations peuvent s'animer sous la forme d'une chaîne de caractère mais la longueur de cette chaîne n'est pas connue avant de s'afficher ce qui peut poser des problèmes d'affichage. Mais ces problèmes d'affichages sont minimes et leur résolution fut sacrifié pour atteindre la couverture minimale requise a l'accomplissement du challenge. de plus ces problèmes ne sont pas inhérent a l'architecture de l'application, mais a sa mise en oeuvre. de tels défauts pourrons être aisément corrigé dans l'avenir, soit par la ré-écriture des deux morceaux d'animation incriminé, soit par le développement d'une seconde génération d'animation dont l'intégration sera triviale grâce au mécanisme de sélection d'animation réflexif.

4.2.4 Polycopié dynamique

Le but de l'application étant de faciliter l'enseignement de l'algorithmique, elle accepte divers paramètres, ainsi il est possible de créer un lien au sein d'un PDF, ou d'une page HTML, renvoyant vers l'application qui charge l'algorithme et l'animation définie au sein des paramètres. De ce fait, l'enseignant en algorithmique peut illustrer son cours a l'aide d'animations de l'algorithme étudié, facilitant sa compréhension. Un tel outil est un allié précieux aussi bien pour l'étudiant que pour l'enseignant.

Grâce a la souplesse de création d'animations spécifique, le principe de polycopié dynamique ne se limite pas à l'algorithmique et peut être étendu a l'enseignement des techniques de traitement de l'image ou des problématiques d'ordonnancement temps réel. Soit en utilisant le panel d'animations actuellement présentes au sein de l'application, soit en créant des animations spécifiques.

4.3 Comparatif avec l'existant

Il existe d'autres *Applets* sur le Web permettant de visualiser des algorithmes.)i(interstices propose une visualisation pour des algorithmes de tri comme par exemple le tri par propagation (ou à bulles), écrit également dans notre langage. Nous nous proposons de comparer les deux.

Dans la version de)i(interstices (voir figure 4.2) il est possible de lancer l'algorithme de l'une de ces deux façons : « tri visuel » ou « tri temporel ». Le tri temporel montre le temps écoulé ainsi que les comparaisons et copies effectuées lors du tri de tableaux dont on peut préciser le

nombre ainsi que la taille. En revanche, dans le tri visuel qui s'approche de notre réalisation, il n'est pas possible de modifier les paramètres, on ne peut trier qu'un seul tableau de 16 éléments dont les éléments sont aléatoirement arrangés à chaque animation. Le texte de l'algorithme n'est pas affiché dans l'*Applet*, mais une explication est mise lors d'une action passée en mode « pas à pas » (comparaison d'éléments, permutation...), mais ce n'est plus le cas en exécution normale. On visualise uniquement l'état du tri (exemple : le plus grand élément se trouve à la position 16). Il y a deux variables permettant d'avoir une idée sur le coût de l'algorithme : le nombre de comparaisons et de copies effectuées. Le tableau d'entier à trier est représenté avec des rectangles pleins pour les valeurs et les indices de tableaux sont indiqués. La représentation graphique est très bien réalisée, cependant l'utilisateur est limité aux actions COMMENCER, PAS À PAS et RECOMMENCER, celui-ci ne peut ni voir ni modifier le texte de l'algorithme et ne peut choisir la représentation des variables.

Dans notre version (voir figure 4.1), il est possible de choisir une réprésentation similaire à)i(interstices sous forme de tableau de rectangles pleins. L'utilisateur a en plus d'autres représentations d'animations possibles pour chaque type. L'utilisateur peut lancer l'algorithme en continu ou en pas à pas et également revenir en arrière. Le texte de l'algorithme étant visible l'utilisateur sait également quelle action va être effectuée à chaque pas. Il est même possible de se faire une idée sur le coût de l'algorithme comme dans la version)i(interstices en modifiant le texte de l'algorithme et en ajoutant des variables comptabilisant le nombre de comparaisons et copies effectuées. Cela pourrait être automatisé dans une version ultérieure de notre système. L'utilisateur peut choisir la valeur de chaque paramètre afin de faire différentes expériences. Par exemple entre deux exécutions, il peut changer la taille du tableau et la valeur des éléments. Ceci marque la grande différence avec)i(interstices car dans notre application, l'utilisateur peut jouer avec l'algorithme, changer les conditions d'arrêt et autres pour mieux comprendre comment et pourquoi l'algorithme fonctionne.

En conclusion, grâce à une grande interactivité avec l'utilisateur, notre version permet de saisir de nombreux algorithmes et de visualiser ceux-ci avec un grand choix de représentations. Ce concept laisse une grande liberté à l'utilisateur du fait que les animations ne sont pas prédéfinies à la différence d'interstices. L'utilisateur à la possibilité de faire ces propres expériences en incorporant ou modifiant certaines parties de l'algorithme.

4.4 Le challenge

Du point de vue du client, le challenge est réussi. Nous avons visualisé le déroulement de divers algorithmes issus du livre *The New Turing Omnibus* de A.K Dewdney [1]. Des algorithmes numériques comme le calcul du plus grand diviseur commun par la méthode d'Euclide, des algorithmes probabilistes tel que la recherche de primalité d'un entier par la méthode de Michael O.Rabin [10] ou encore des algorithmes graphiques tels que les tours de Hanoï ou le jeu de la vie. La plupart des algorithmes retenus sont implémentables dans notre langage sauf certains en style récursif ou utilisant les réels. Néanmoins, ces derniers ne rentrent pas dans les spécifications de développement de la version 0 définis avec le client. Ils pourront être ajoutés dans une version future.

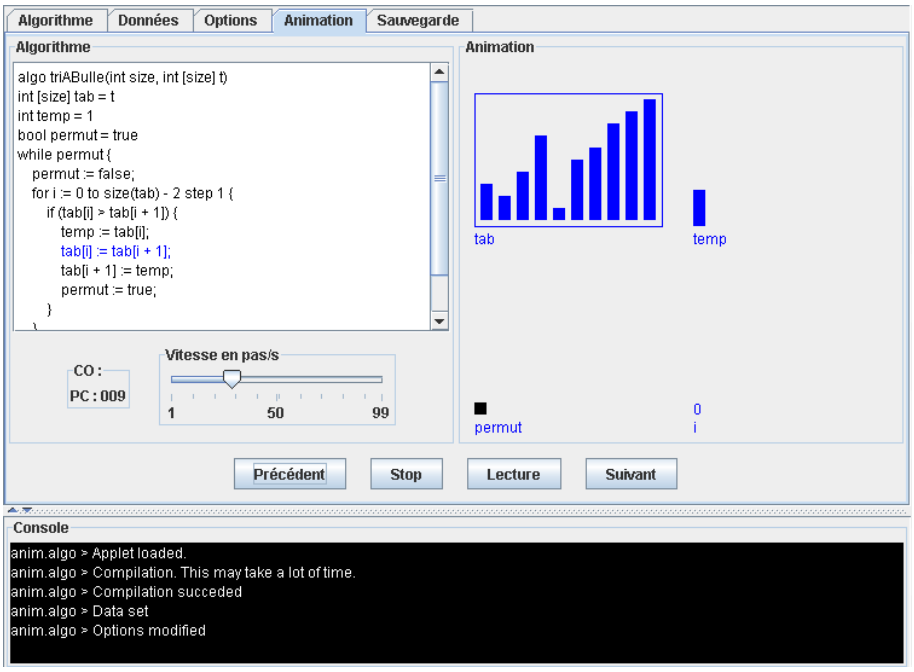


FIG. 4.1 – AnimAlgo

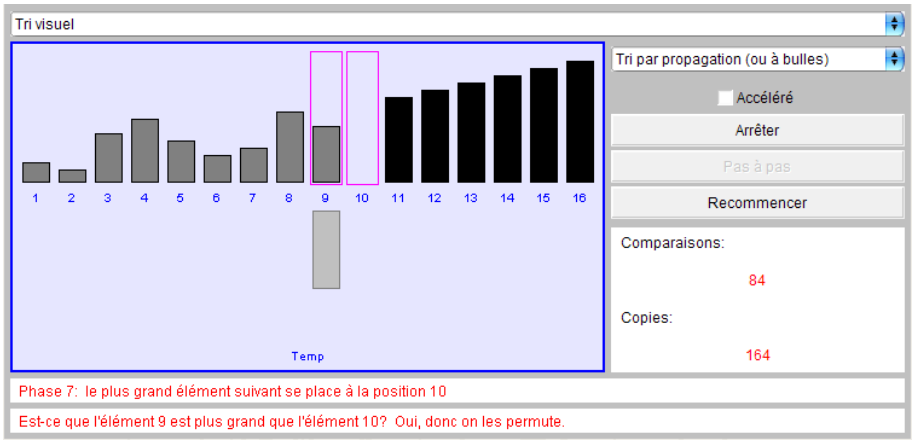


FIG. 4.2 – Interstices

Conclusion et Perspectives

Réaliser un logiciel d'animation d'algorithmes capable d'animer différents types d'algorithmes (numériques, probabilistes, graphiques ...). Tel était le challenge fixé par notre encadreur Mr Ridoux. Celui-ci avait à la fois un rôle d'utilisateur final et de « client ». Il a exprimé le besoin d'avoir une application disponible en ligne et a choisit les instructions et les types offerts par le langage. Nous avions carte blanche pour le reste. Pour mener à bien notre projet il a donc fallu trouver des réponses aux questions suivantes : *Quelles sont les fonctionnalités offertes par notre application ? Quels sont ses principes de fonctionnement ? De quelle manière et dans quelles conditions allons nous la développer ?*

Après avoir étudié l'art existant dans le domaine des animations d'algorithmes et à travers un travail d'étude et de recherche, nous avons défini ce que l'application AnimAlgo propose : saisir ou charger des algorithmes ainsi que les jeux de données qui y sont associés, sélectionner les variables de l'algorithme que l'on souhaite visualiser, animer ces algorithmes pas à pas, de manière continue et revenir en arrière, sauvegarder l'ensemble de ces étapes. Le langage algorithmique devait être simple en terme de structures de contrôles et riche en terme de structures de données afin de permettre la modélisation de nombreux problèmes algorithmiques. C'est ainsi que nous avons pu déterminer le fonctionnement globale de notre application.

Chaque semaine, l'équipe avait rendez vous avec le « client » pour faire le bilan sur le travail effectué et décider de la marche à suivre pour la semaine suivante en concordance avec l'ensemble des fonctionnalités initialement prévues. Nous nous sommes efforcés de suivre les méthodes de développement (cycle de développement en spirale, importance des tests ...). Ce cadre méthodologique a servi de garde fou et nous a contribué au bon déroulement de notre réalisation.

Tout au long de ce projet, nous avons pu acquérir (pour certains) ou approfondir (pour d'autres) un certain nombre de connaissances, particulièrement en ce qui concerne le travail en équipe, l'étude bibliographique d'un domaine, l'analyse d'un problème, l'élaboration d'une solution ainsi que sa mise en oeuvre.

L'application est fonctionnelle, il ne s'agit cependant que de la version 0. De nombreuses fonctionnalités peuvent encore être ajoutées lors d'une reprise future du projet. Sans être limitative, la liste ci dessous, permet de se faire une idée de ce qui pourrait être ajouté.

- *Langage d'algorithmes :* gestion des processus, création de fonctions par l'utilisateur...
- *Types de données et structures :* les réels, les matrices, les images...
- *Types d'animation et options d'animation :* gestion de la 3D, gestion de l'emplacement et de la taille des objets observés, implémentation du glisser-déposer dans l'interface...
- *Serveurs et sauvegarde :* création de serveur externe de sauvegarde, choix du serveur de sauvegarde, sauvegarde de l'animation en format externe (flash, pdf)...
- *Utilisation du cœur applicatif :* création d'applications utilisant notre application comme une brique de base.

Nous considérons que le challenge à été rempli et que nous sommes arrivé à l'objectif que nous nous étions fixé : Développer dans un temps impartit une première version de l'application. Ce concept novateur offre un logiciel pratique et utile permettant de nombreuses expérimentations sur les algorithmes.

Remerciements

Nous tenons à remercier du fond du coeur toutes les personnes qui ont pu nous aider durant la réalisation de ce projet.

Tout d'abord, un grand merci à Olivier Ridoux pour avoir imaginé un projet aussi intéressant. Mais aussi pour ses conseils avisés, sa pédagogie, et ses gâteaux délicieux... Et un grand coup de chapeau pour son incroyable disponibilité !

Merci aussi à l'ensemble des enseignants et personnels de l'IFSIC (nous ne pouvons tous les citer, de peur de rajouter une dizaine de page à ce rapport) pour leur aide et leurs conseils souvent précieux. Merci au personnel de la cafétéria et ses boissons chaudes au goût de café (sic), ainsi que sa bonne humeur permanente, qui nous ont permis de tenir le coup pendant certains après midis un peu longs... Merci aux imprimantes de l'IFSIC, que l'on a mis à rude épreuve lors de la rédaction des rapports (certains quotas ne s'en sont jamais remis).

Merci à nos parents, amis, compagnes, et à toutes les personnes qui ont pu nous supporter, nous encourager et nous motiver pendant ce projet.

Le projet *animalgo* en chiffres :

- 1 projet
- $8m^2$ d'encre
- 9 boites d'aspirine
- 12 ramettes de papier
- 23 installations d'Eclipse
- 25 crayons papier
- 63 kilos de pates
- 130 litres de café
- 183 heures de téléphone
- 12350 redémarrages du serveur
- et beaucoup de plaisir...

Bibliographie

[1] A. K. Dewdney. *New Turing Omnibus.* W. H. Freeman & Co., 1993.

[2] Olivier Arsac, Stéphane Dalmas, and Marc Gaëtano. Algorithm animation with AGAT. Technical report, INRIA Sophia Antipolis, 1997.

[3] Willard C. Pierson and Susan H. Rodger. Web-based animation of data structures using JAWAA. In *SIGCSE '98 : Proceedings of the twenty-ninth SIGCSE technical symposium on Computer science education*, pages 267–271. ACM Press, 1998.

[4] Neil Jones. *Computability and Complexity : From a Programming Perspective.* MIT Press, 1997.

[5] Christophe Kolski. *Interfaces Homme-Machine : application aux systèmes industriels complexes.* Hermes, 1997.

[6] Richard Helm, Ralph Johnson, John Vlissides, and Erich Gamma. *Design patterns.* Addison-Wesley Professional Computing Series. Addison-Wesley, 1995.

[7] Joshua Engel. *Programming for the Java Virtual Machine.* Addison-Wesley, Reading, MA, USA, 1999.

[8] Chuck Cavaness and Brian Keeton. *Jakarta Struts pocket reference.* O'Reilly & Associates, Inc., 2003.

[9] Eric Chaber. *Les JSP avec Struts, Eclipse et Tomcat.* Dunod, 2004.

[10] M. O. Rabin. *Algorithms and Complexity, New Directions and Recent Trends.* J. F. Traud, 1976.

[11] Chromatic. *Extreme Programming pocket guide.* O'Reilly & Associates, Inc., 2003.

[12] Gregor N. Purdy. *CVS Pocket Reference.* O'Reilly & Associates, Inc., 2000.

Annexes

Annexe A

Composition de l'équipe

Durant la phase d'analyse

- Erwan Abgrall
- Lionel Aissi
- Simon André
- Julien Audo
- Sebastien Bolo
- Emmanuel Caruyer
- Damien Hardy
- Teddy Houdayer
- Kévin Huguenin
- Nicolas Vigot
- François Wang

Durant la phase de développement

- Erwan Abgrall
- Lionel Aissi
- Simon André
- Julien Audo
- Damien Hardy
- Teddy Houdayer
- Nicolas Vigot
- François Wang

Annexe B

À propos de ce rapport

Réalisé avec LaTeX

Ce rapport a été réalisé sous LaTeX. Nous avons utilisé la classe *report*, ainsi que les *packages* suivant : *a4wide, fancyhdr, graphicx, float, listings, subfigure* et *prooftree*.

Où trouver ce document ?

Ce rapport peut être téléchargé, au format pdf, à l'adresse suivant :
http ://animalgo.free.fr/dl/rapport.pdf.

Dernière mise à jour

Ce document a été compilé pour la dernière fois le 5 mai 2006.