



VILLARD David

Licence Informatique Fondamentale troisième année

Année universitaire: 2011-2012

# DebateWEL: Un jeu de dialogue de persuasion utilisant des enthymèmes

Tuteur universitaire: Bannay Florence

Dates: TER du 09/04/2012 au 09/06/2012

## Sommaire

I- Contexte du TER.....	3
1. Déroulement du TER .....	3
2. Présentation du cadre .....	4
A) L'IRIT .....	4
B) L'équipe ADRIA .....	5
3. Fonctionnement des réunions de travail .....	6
II- Sujet .....	7
1. Résumé/Introduction .....	7
2. Détail des besoins utilisateurs .....	9
A) Interface .....	9
B) Éventail des coups .....	10
C) Fonctionnalités .....	11
III- Résolution du problème.....	12
1. Spécifications .....	12

A) Schéma de l'architecture .....	12
B) Détail des classes .....	13
2. SAToulouse .....	15
A) Présentation .....	15
B) Intérêts pour notre projet .....	17
C) Intégration/adaptation .....	18
3. Points techniques .....	21
A) Sauvegarde/Restauration .....	21
B) Affichage des informations.....	23
C) Option pour lancement rapide d'une partie.....	24
D) Difficultés rencontrées .....	24
IV- Bilan .....	26
1. Satisfactions .....	26
2. Regrets .....	27
A) L'intelligence artificielle .....	27
B) Le compte-rendu .....	27
V- Conclusion.....	28
VI- Glossaire .....	30
VII- Annexe .....	33

### ***1. Déroulement du TER***

Nous avons dans un premier temps «dépoussiérer» le sujet en imaginant tous les besoins que le logiciel devait prendre en compte.

Au début du projet, nous en avons trouvé un maximum même si nous savions déjà que certaines des fonctionnalités ne pourraient être explorées.

Mais comme les délais n'étaient pas fixés, il valait mieux en mettre trop que pas assez afin de ne pas pas se retrouver avec un logiciel terminé à un mois de la soutenance mais non modulable, non adaptable avec des fonctionnalités inventées sur le tas.

Après ces deux jours de réflexion, on a passé une semaine de spécifications à tenter de trouver l'architecture la plus adéquate pour répondre aux problèmes en utilisant des graphes UML.

Nous nous sommes réunis du lundi au vendredi pendant une heure avec notre référent et chaque lendemain on arrivait avec une évolution de ce qu'on avait proposé la veille mais qu'on avait convenu ensemble de modifier de telle ou telle manière.

Par la suite, la fréquence des réunions s'est réduite ; nous nous rencontrions hebdomadairement.

Nous nous sommes pas réparti le travail de manière très précise.

Lorsqu'un de nous avait terminé sa séance de travail et qu'il jugeait avoir fait avancer le projet, il soumettait une archive du dossier *src/* du projet à son camarade par mail (aussitôt suivi d'un sms) en lui faisant un bref rapport de ce qui avait été développé depuis la dernière archive reçu ou envoyé, des difficultés qu'il a rencontré et dont il n'a pas encore trouvé de solution ainsi que ce qu'il aimerait dont on s'occupe dans les jours suivants.

La contrainte principale étant de veiller à ce que l'autre ne code pas en même temps sur les mêmes parties, il était alors indispensable de prévenir son collègue à chaque fois que l'on allait programmer en lui précisant sur quoi on allait travailler et la durée estimée du travail qu'on s'apprêtait à faire.

Il est vrai et je le reconnais qu'au début, quand on n'avait pas encore le squelette du jeu, qu'on était amené à modifier de manière fine tant de classes et tant de fonctions, qu'il s'agissait plus d'un travail en alterné que d'un travail en parallèle.

Mais une fois que l'on a obtenu une version gérant tous les besoins primaires (sans système de restauration, sans gestion du score, deux sortes de coups), on s'est alors réparti le travail de manière plus intelligente (pendant qu'un gèrait l'affichage, l'autre codait un nouveau coup etc.)

### ***2. Présentation du cadre***

#### **A) L'IRIT**

L'institut de Recherche en Informatique de Toulouse représente un des plus forts potentiels de la recherche en informatique en France avec un effectif global de 600 personnes dont 250 chercheurs et enseignants-chercheurs, 244 doctorants, 14 Post-Doctorants et chercheurs contractuels ainsi que 43 ingénieurs et administratifs.

Les 19 équipes de recherche du laboratoire sont structurées en sept thèmes scientifiques qui couvrent l'ensemble des domaines de l'informatique actuelle :

- ✓ Analyse et synthèse de l'information
- ✓ Indexation et recherche d'informations
- ✓ Interaction, autonomie, dialogue et coopération
- ✓ Raisonnement et décision
- ✓ Modélisation, algorithmes et calcul haute performance
- ✓ Architecture, systèmes et réseaux
- ✓ Sécurité de développement du logiciel

## **B) L'équipe ADRIA**

**L'équipe ADRIA dans laquelle évolue notre tuteur** couvre le quatrième thème «Raisonnement et décision».

L'équipe se consacre à différents types de raisonnement :

- ✓ la révision d'informations tenues (plus ou moins certainement) pour vraies à l'arrivée de nouvelles informations
- ✓ la fusion d'informations partiellement contradictoires ou incertaines provenant de sources multiples
- ✓ le raisonnement abductif, qui permet de remonter aux causes plausibles d'une situation observée en diagnostic
- ✓ le raisonnement d'interpolation en logique floue
- ✓ le raisonnement en présence d'exceptions.

Ses travaux portent aussi sur les problématiques de décision, pour lesquelles l'équipe développe, avec un souci de calculabilité, des modèles originaux de représentation des préférences, ainsi que de nouveaux critères de décision qualitatifs de façon à être plus proches des manières dont l'humain peut appréhender ses choix.

Une part importante des recherches actuelles de l'équipe concerne la formalisation de l'argumentation. Cette forme de raisonnement permet d'expliquer des conclusions et de gérer des contradictions. Elle joue aussi un rôle crucial dans les dialogues de négociation notamment. Plus récemment, elle a abordé aussi des questions d'apprentissage, en particulier à partir de données incertaines.

## ***3. Fonctionnement des réunions de travail***

Pendant les réunions de travail, nous nous installions tous les trois devant un ordinateur portable équipé de Netbeans.

Nous montrions à l'enseignante les points améliorés depuis la dernière rencontre.

Nous débattions de ce qu'il serait intéressant d'ajouter, de modifier et nous lui présentions ce qu'on pensait concevoir durant les jours à suivre.

Celle-ci nous fixait les priorités mais nous laissait une liberté conséquente.

Nous repartions de chaque réunion avec un compte-rendu faisant office de feuille de bord pour la semaine suivante.

Cette liste n'était pas trop exhaustive, alors une fois que ces points furent gérés on se fixait par e-mail la suite des événements.



### ***1. Résumé/Introduction***

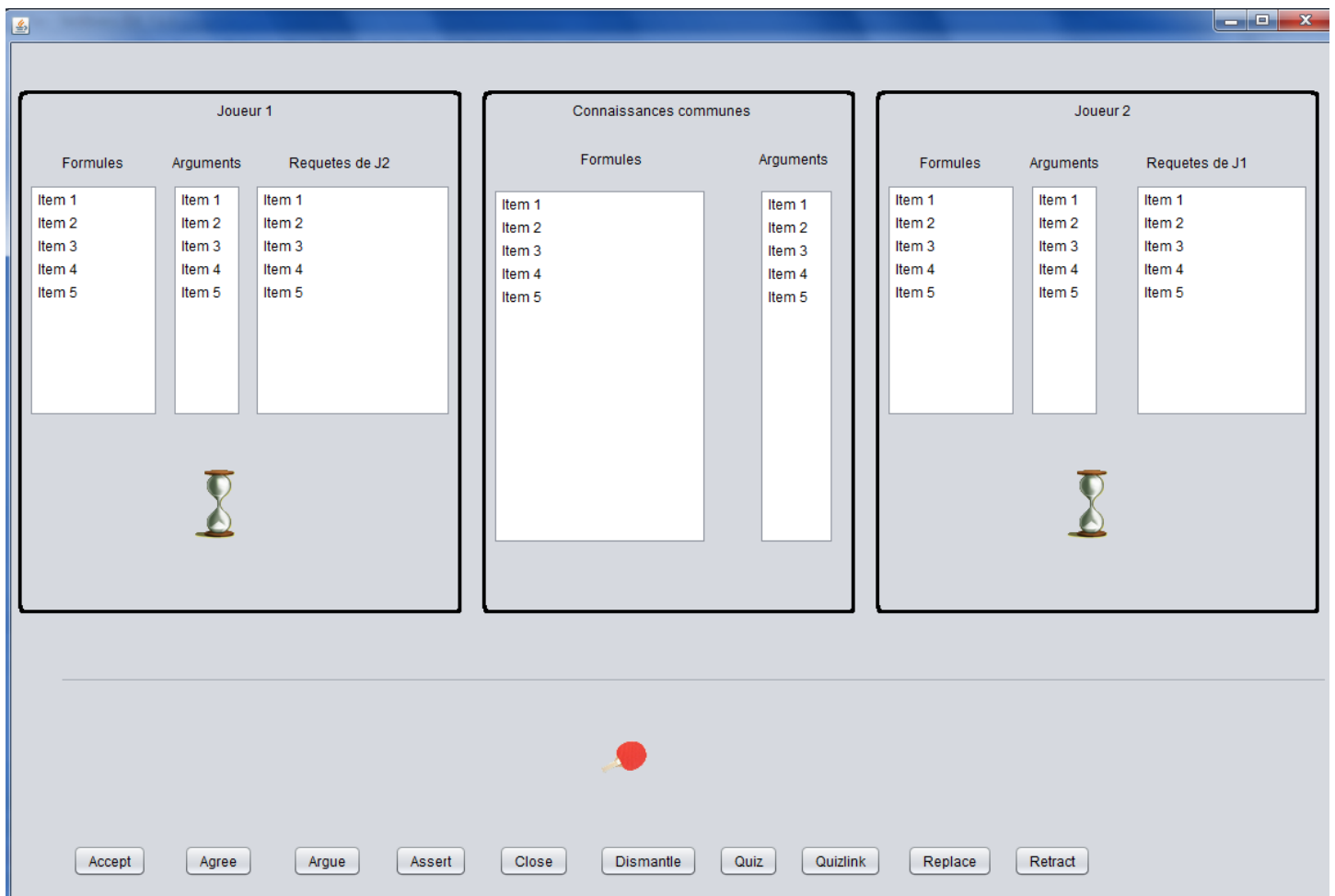
Plusieurs spécificités sont codées dans DebateWel:

- ✓ deux agents sont considérés (par soucis de simplicité)
- ✓ au commencement du dialogue, les agents se mettent d'accord sur un socle de connaissances communes sur lequel ils vont baser leurs prochaines déclarations.
- ✓ pas d'information concernant l'état d'esprit des agents, seulement ce qui sera prononcé publiquement sera pris en compte
- ✓ le contenu des coups est basé sur la logique propositionnelle classique
- ✓ les arguments échangés ne sont pas des entités abstraites mais des paires (S, c) où S est un ensemble de formules (appelée support) et c une formule constituant la conclusion.
- ✓ Par ailleurs, l'idée est d'autoriser la saisie de couples dans lesquels S n'est pas une preuve parfaite minimale de c.
  - ✗ Quand quelque chose manque pour prouver c depuis S, cette paire est appelé un enthymème.
  - ✗ L'utilisation d'enthymèmes est très commune dans les discussions entre individus car la preuve est souvent trop longue à énoncer mais évidemment admise.
- ✓ La connaissance commune pour les deux agents est composée de formules et enthymèmes (car ils peuvent convenir qu'une paire est une preuve imparfaite de sa conclusion bien que suffisamment convaincante).
- ✓ Cette connaissance commune peut seulement s'agrandir au cours du dialogue lorsque les deux agents sont d'accord avec l'adversaire sur une proposition ou un enthymème.
- ✓ Le protocole est totalement flexible, à savoir qu'un agent peut effectuer un coup quand il le souhaite, peut dire ce qu'il veut (à condition qu'il ait la main) tant qu'il n'enfreint pas les trois règles donnés ci-dessous.
  - ✗ l'auto-contradiction
  - ✗ la répétition
  - ✗ pour fermer le dialogue, l'agent doit remplir tous les engagements engendrés par les mouvements de l'autre agent.
- ✓ Le débat s'achève soit par la victoire d'un agent si l'autre agent est d'accord avec lui ou avec un échec si les agents n'ont pas réussi à remplir leurs engagements dans le temps imparti (soit exprimé en secondes ou en termes de nombre de symboles utilisés).
- ✓ L'idée du DebateWEL est d'encourager les agents à se retrouver sur un terrain d'entente plutôt que de parvenir à un échec du débat, d'où un score
  - ✗ élevé pour un agent victorieux
  - ✗ médian pour un agent défait
  - ✗ faible pour un dialogue échoué.

## 2. *Détail des besoins utilisateurs*

### A) Interface

Nous avons dès le départ conçu une interface sans handler pour gérer les événements. Ceci dans le but de valider auprès de notre tuteur les éléments de base que notre jeu devait contenir.



### B) Éventail des coups

- ✓ **Accept(phi)**: acceptation de la formule phi
- ✓ **Agree(arg, phi)**: acceptation de l'argument arg éventuellement incomplet qui justifie phi
- ✓ **Argue(arg, phi)**: annonce d'un argument éventuellement incomplet qui justifie phi
- ✓ **Assert(phi)**: affirmation de la formule phi
- ✓ **Challenge(phi)**: demande d'un argument justifiant phi
- ✓ **Close**: fermeture du dialogue
- ✓ **Dismantle<S, phi>**: retrait de l'argument <S, phi>
- ✓ **Quiz<S, phi>**: demande de complétion de l'argument <S, phi>
- ✓ **Quizlink<S, phi>**: demande d'une complétion dont la conclusion a un lien avec le dialogue
- ✓ **Replace (a; b)** : annonce d'un argument b qui complète a
- ✓ **Retract(phi)**: retrait de l'affirmation phi

## C) Fonctionnalités

- ✓ Base d'exemples
  - ✗ On doit avoir la possibilité de charger des fichiers dit thèmes afin d'alimenter les fenêtres de saisie de formules et d'arguments
  - ✗ L'idée étant d'inciter les joueurs à ne pas proposer d'atomes ne figurant pas dans l'ensemble de cette base.
  - ✗ Ceci permettrait également de ne pas avoir un débat trop ouvert mais bien un débat qui traite un sujet en profondeur.
- ✓ Compte rendu
  - ✗ Lorsque la partie s'achève, le logiciel calcule un compte rendu en remplaçant chaque variable propositionnelle par des affirmations extraites d'une base de donnée existante ou créée.
  - ✗ Mieux il pourra essayer de remplacer les inclusions par des mots conjonctions ou des adverbes (ainsi, donc), les et par des « ainsi que » les ou par des « mais aussi » etc.
- ✓ Mode



✗ Multijoueur: Joueur 1 vs Joueur 2

✗ Joueur vs IA

✓ Système de restauration/sauvegarde

✗ Enregistrement du répertoire de sauvegarde pour l'ouvrir aussitôt comme répertoire par défaut lors d'une restauration.

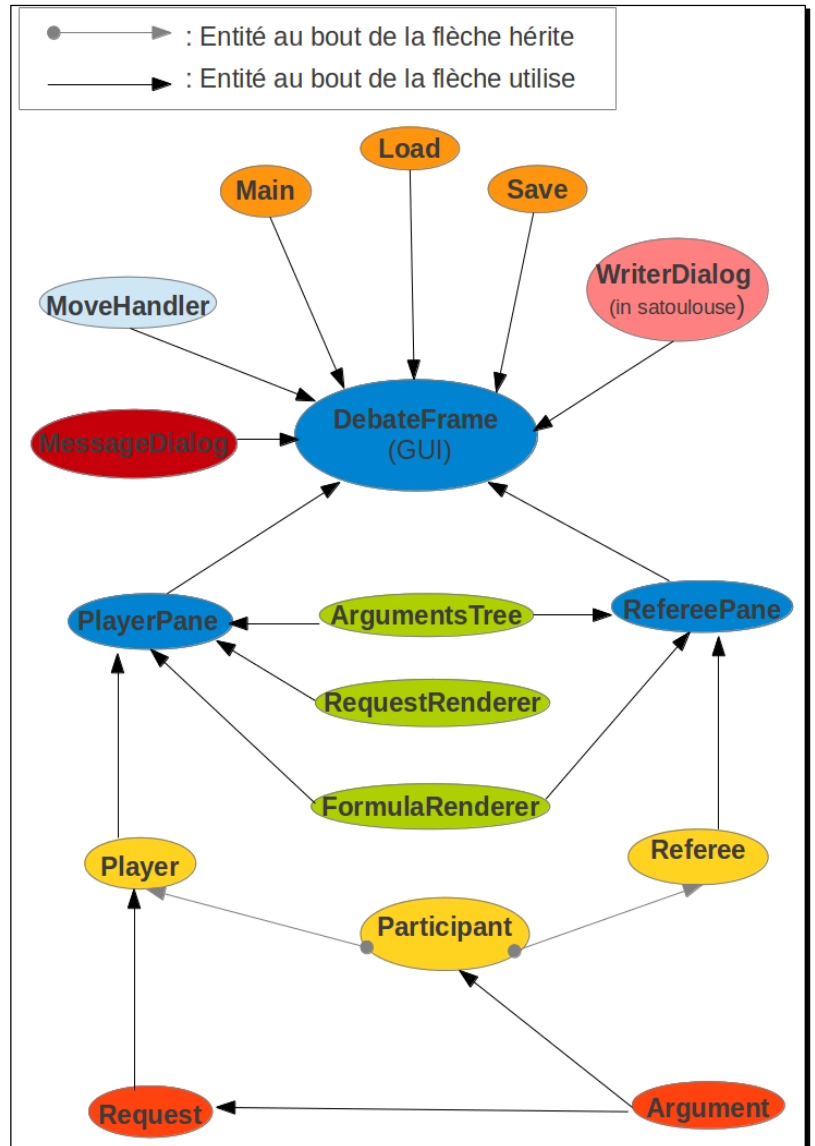
✓ Applet JAVA

## 1. Spécifications



































### A) Schéma de l'architecture



### B) Détail des



## classes

- ▼  debate.agents
  -  Participant.java
  -  Player.java
  -  Referee.java
- ▼  debate.gui
  -  DebateFrame.java
  -  MovesPane.java
  -  PlayerPane.java
  -  RefereePane.java
- ▼  debate.gui.renderer
  -  CellListRendererForFormula.java
  -  CellListRendererForRequest.java
  -  CompoundIcon.java
  -  RequestForListPanel.java
  -  TreeCellRendererForArgument.java
- ▼  debate.logical
  -  Argument.java
  -  Request.java
- ▼  debate.menu
  -  Main.java
  -  SaveConfiguration.java
- ▼  debate.move
  -  MoveEnum.java
  -  MoveHandler.java
- ▼  debate.tools
  -  CongratulationsDialog.java
  -  EndOfGameEnum.java
  -  ErrorEnum.java
  -  HandStatus.java
  -  MessageDialog.java
  -  PlaySound.java
  -  ScoreEnum.java
  -  SoundEnum.java
  -  SuccessEnum.java

### Vue du navigateur de Projet sous Netbeans

Participant	Liste des formules: List Liste des argument: List Solver : SATSolverSAT4J
Player (extends Participant)	Liste des requêtes: List Compteur de formules acceptés: int Pseudonyme: string Temps de parole: int
Referee (extends Participant)	Activation du son: boolean Situation (qui a la main): HandStatus

	Liste des thèmes chargés:List Compteur des coups acceptés: int.
DebateFrame	Interface principale: <i>GUI</i> . Traite les événements générés par l'utilisateur.
MovesPane	Panneau comportant les onze coups réalisables.
PlayerPane	Panneau propre aux informations du joueur. Passerelle entre la GUI et le joueur qu'elle contient
RefereePane	Panneau propre aux information de l'arbitre. Passerelle entre la GUI et l'arbitre qu'elle contient
ListCellRenderForFormula	Rendu pour afficher une formule en <b>LaTEX</b> (explication plus bas)
ListCellRenderForRequest	Rendu pour afficher une requête en <b>LaTEX</b> avec puce coloré selon le type de la requête.
TreeCellRenderForArgument	Rendu d'un argument.
Argument	support (liste de formule) conclusion (formule)
Request	<ul style="list-style-type: none"> <li>– formule</li> <li>– argument</li> <li>– type de coup</li> </ul>
Main	Lance la <i>GUI</i> DebateFrame en installant le <i>LookAndFeel</i> au préalable.
SaveConfiguration	Enregistre les infos dans un fichier texte (voir plus bas pour la manière utilisée).
MoveEnum	Type de coup (Assert, Accept, Agree etc).
MoveHandler	Vérifie redondance d'une formule ou d'un argument par rapport à la liste du joueur (self redundancy) ou de l'arbitre (global redundancy).
CongratulationsDialog	Fenêtre de félicitations.
EndOfGameEnum	Type de déclenchement de la fin de partie.
ErrorEnum	Type d'erreur à envoyer à MatDialog.

HandStatus	<u>Type de situation</u> : personne n'a la main, joueur 1 ou joueur 2.
MessageDialog	Message de succès ou d'erreur.
PlaySound	Lance le son selon le SoundEnum lui étant passé en paramètre.
ScoreEnum	Type de comptage du score.
SoundEnum	Type du son à jouer.
SuccessEnum	Type de succès à envoyer à MessageDialog.

## 2. SAToulouse

### A) Présentation

Ce logiciel permet de s'initier à la programmation logique à l'aide de la logique propositionnelle.

L'utilisateur programme la résolution d'un problème en le décrivant à l'aide de la logique propositionnelle. Contrairement à la programmation impérative (Java, C, etc.), où on donne les instructions qui devront être exécutées pour résoudre un problème, la programmation logique est descriptive.

Programmer revient à décrire les règles du problème que l'on souhaite résoudre.

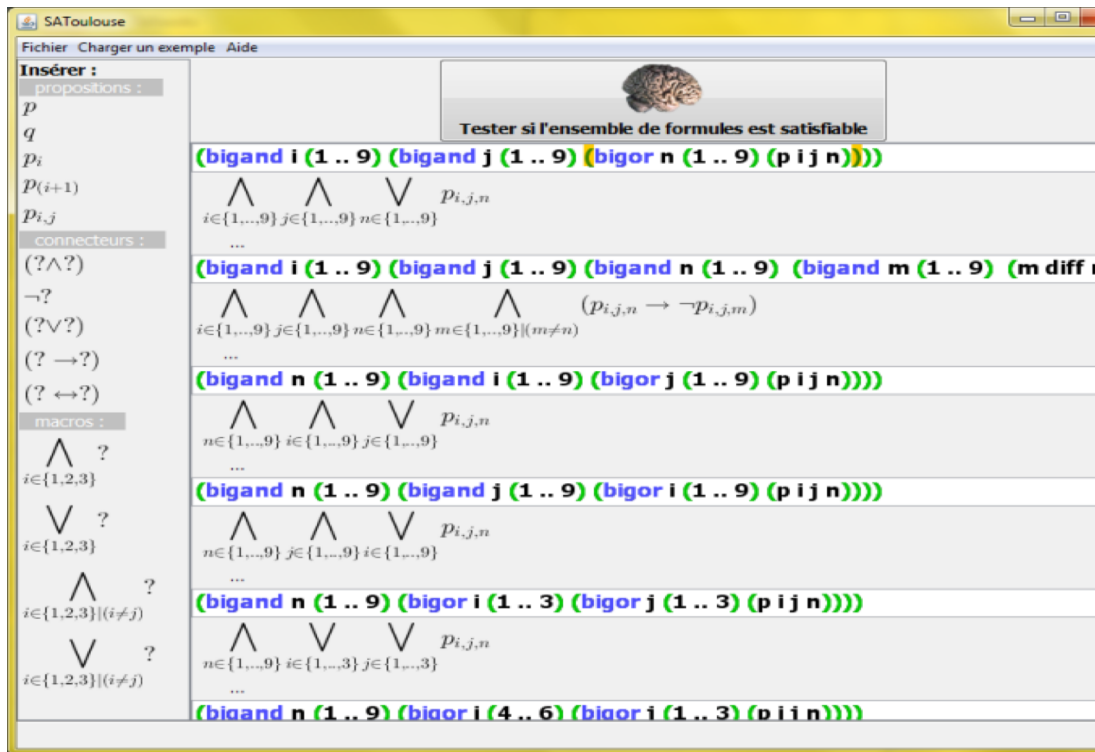
Ainsi, les problèmes traités sont formalisés en logique propositionnelle et ramenés au problème de « satisfaisabilité ».

Une fois un problème décrit par un ensemble de formules booléennes, le logiciel permet de vérifier si leur intégralité est recevable (*satisfiable*) ; en d'autres termes, de savoir si le problème possède une solution.

Dans le cas où le problème a une solution, le logiciel donne une « valuation » (aussi appelée « interprétation ») : la « valuation » correspond au résultat du problème.

Le logiciel permet de remplir les champs de formules avec un chargement d'un fichier texte contenant des formules écrites ligne par ligne.

L'ensemble des modèles des connecteurs valides sont disponibles dans une palette à gauche et peuvent être chargés en cliquant dessus.



Capture d'écran du logiciel SAToulouse

## B) Intérêts pour notre projet

Le projet est censé comme le rappelle le titre modéliser des débats logiques. Il était alors quasi-indispensable une fois ayant eu connaissance et eu à notre disposition le code de SAToulouse de l'utiliser pour résoudre la première grosse problématique à savoir comment on allait pouvoir vérifier la redondance, vérifier l'intégrité des formules, la consistance entre plusieurs d'entre elles etc.

Je me suis souvenu d'avoir utilisé en cours de Logique durant la deuxième année de licence ce dernier pour résoudre des sudokus. Je n'avais en revanche pas retenu que celui-ci était codé en JAVA et encore moins que sa licence était libre comme notre tuteur nous l'a rappelé.

Ne connaissant aucune bibliothèque particulière qu'elle soit de l'ordre de l'affichage, ou de l'ordre du calcul de satisfaisabilité, la tâche se serait annoncée particulièrement ardue voire très compromise.

La lecture et la compréhension de ce code source aura donc été la première longue séquence de travail.

Et au vu de toutes les bibliothèques jar attachées au dossier source j'ai compris que de tout coder soi-même était de toute manière voué à l'échec.

## C) Intégration/adaptation

### **a) Besoins supplémentaires par rapport à SAToulouse**

- ✓ extraction d'atomes à partir de formules
- ✓ ajout de formules ou atomes dans la palette à gauche
- ✓ bouton pour vider tous les champs
- ✓ bouton pour ajouter un champ de saisie de formule
- ✓ emplacement pour champ de saisie d'une conclusion (pour l'ArgumentWriter)
- ✓ verrouillage des champs de saisie de formule pré-remplies lors d'un replace

### **b) Éléments inutiles**

- ✓ fenêtre de résultat
- ✓ fenêtre d'aide
- ✓ chronomètre
- ✓ fenêtre d'attente de résultat
- ✓ fenêtre pour charger un Sudoku
- ✓ menu
- ✓ fenêtre « à propos »
- ✓ gestion des macros (bigand, bigorr)

### **c) Réalisation**

J'ai tout d'abord après avoir cerné les liens entre les classes essayé d'ébaucher une fenêtre assez simplifiée qui permette de saisir une seule formule.

J'ai enlevé les macros (bigands, bigorr) ainsi que le menu pour simplifier un maximum l'interface qui allait dans notre projet devenir de toute façon secondaire, appelé par la *GUI* pour certains coups (assert, argue et replace),

Pour rendre le paquet plus personnalisé et moins hostile à mon collègue qui m'avait laissé gérer cette partie, j'ai en plus de supprimer l'appel à certaines classes sans intérêts pour notre jeu, supprimé un bon nombre de classes (Chronomètre, WaitDialog, SATResults, HelpPanel, DialogHelp, DialogGrille2D, SAToulouseAboutBox).

Pour SATResult bien que supprimé, j'ai repris le comportement pour avertir de la non consistance d'une formule et d'un argument par la suite.

SAToulouse fonctionnait avec le duo de classes suivant : une extension de SingleFrameApplication lance une SingleView.

En cours de programmation événementielle, nous avons uniquement utilisé la bibliothèque JavaSwing

donc pour éviter de rencontrer des problèmes par la suite et pensant qu'il était moins obsolète de fonctionner ainsi, j'ai essayé d'adapter en supprimant la *SingleFrameApplication* (sorte de main qui charge des ressources) et en faisant passer l'interface d'application (classe *FrameView*) indépendante à une *JFrame*.

J'ai pour cela repris les composants de la View un à un et j'ai recrée une *JFrame*.

Ceci dit la conversion posa un problème dans la mesure où certains paramètres étaient enregistrés dans la ressource du *SingleFrameApplication* qui devait disparaître donc j'ai repris les paramètres des fichiers (.properties) pour les prendre en compte via *Matisse*.

Cette *JFrame* allait passer en *JDialog* un peu plus tard quand les événements seraient gérés, mais rester en *JFrame* avait l'intérêt de pouvoir la lancer directement et gagner du temps précieux lors des tests.

J'ai personnalisé le comportement du bouton validation.

Il me fallait un outil qui puisse:

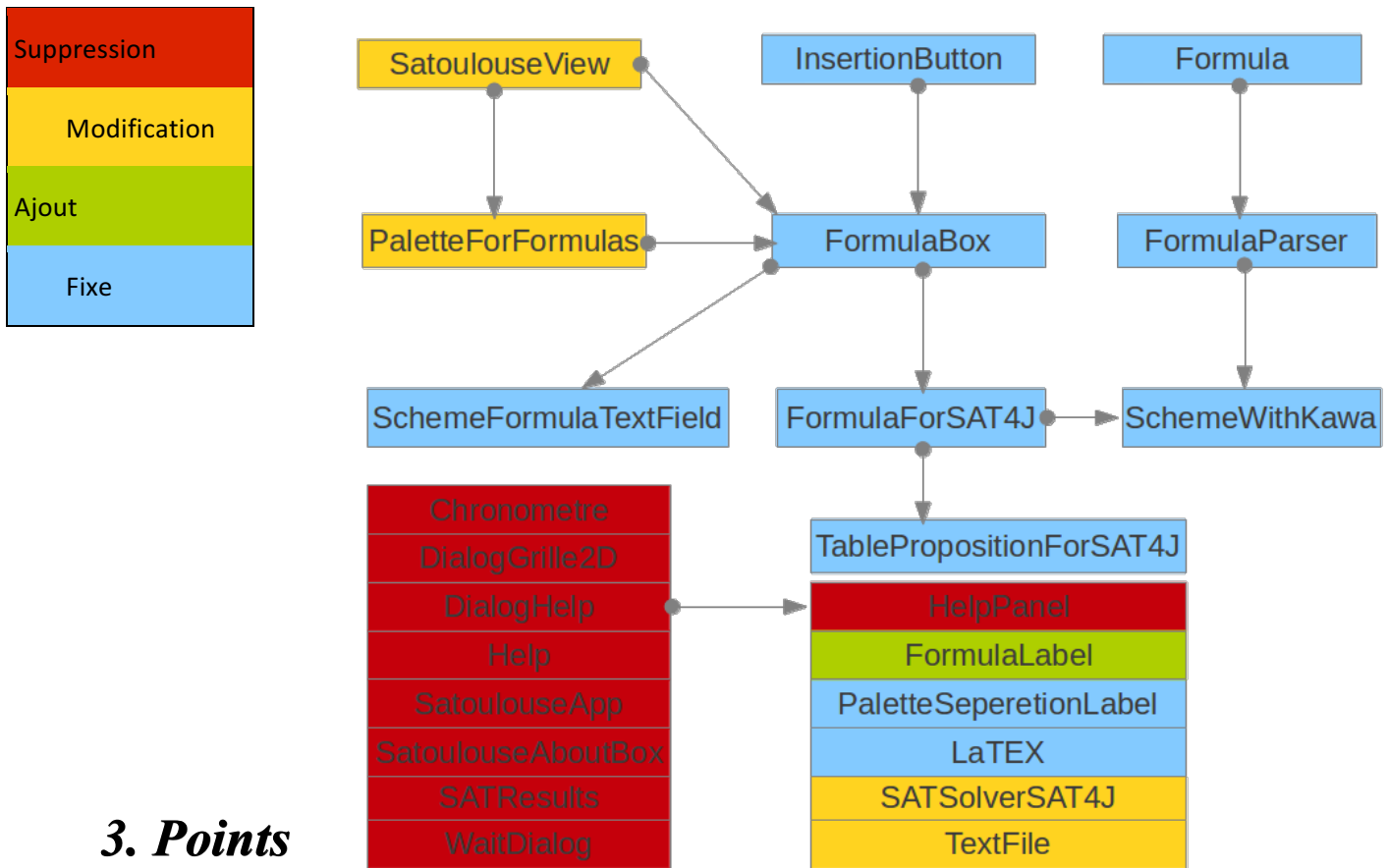
- ✓ charger les formules saisies dans la fenêtre courante des deux solvers.
- ✓ ajouter dans le premier l'ensemble des formules de l'arbitre
- ✓ ajouter dans le second celles déjà évoquées par le joueur courant
- ✓ soumettre les deux à des tests de consistance
- ✓ avertir le joueur en cas d'erreur de cohérence (avec soi même, avec les connaissances communes).

Une fois que ceci fonctionnait, j'ai voulu ajouter sur la palette flottante à gauche une aide à la saisie.

Pour extraire les atomes, j'ai modifié la classe *PaletteForFormula* et crée une fonction dans la classe *SATSolverSAT4J* (sur le modèle de la méthode déjà existante qui retournait les modèles de satisfaisabilité pour un problème).



#### d) Diagramme de classes



### **3. Points techniques**

#### **A) Sauvegarde/Restauration**

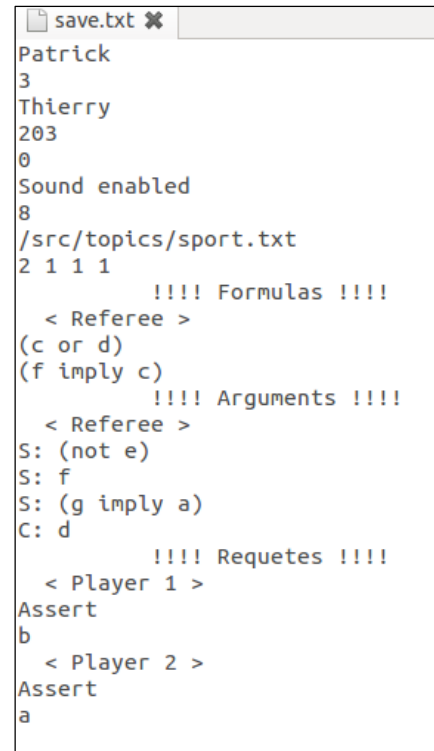
On utilise les fonctions prédéfinies de `satoulouse.TextFile` permettant de supprimer un fichier en mémoire, d'écrire une liste ou une ligne dans un fichier.

Le chemin aura été choisi via une fonction cette fois ci codée par nous même en utilisant un `JFileChooser`. J'ai modifié les fonctions d'écriture de façon à ce que l'on écrive à la suite et non écraser à chaque nouvelle écriture.

Format type de sauvegarde

Exemple

<nom du premier joueur: String>  
 <temps restant de parole  
 du premier joueur: int>  
 <activation du son: String>  
 <compteur de coups acceptés: int>  
 <thèmes (peut être vide)>  
 <nombre de formules de l'arbitre,  
 nombre d'arguments de l'arbitre,  
 nombre de requêtes en cours du joueur1,  
 nombre de requêtes en cours du joueur2>  
 <Formules de l'arbitre>  
 <Arguments de l'arbitre>  
 <Requêtes du joueur 1>  
 <Requêtes du joueur 2>



```

Patrick
3
Thierry
203
0
Sound enabled
8
/src/topics/sport.txt
2 1 1 1
      !!!! Formulas !!!!
    < Referee >
(c or d)
(f imply c)
      !!!! Arguments !!!!
    < Referee >
S: (not e)
S: f
S: (g imply a)
C: d
      !!!! Requetes !!!!
    < Player 1 >
Assert
b
    < Player 2 >
Assert
a
  
```

Au départ je comptais enregistrer formules et arguments des joueurs mais dans la mesure où chaque argument chez l'un correspond à une requête en cours Argue chez l'autre et où chaque formule correspond à un Assert chez l'autre, nous n'enregistrons pour les joueurs que leurs requêtes.

Pour restaurer, on va lire dans le fichier de sauvegarde sélectionné via la fonction de lecture également présente dans `satoulouse.TextFile` en récupérant une liste de String.

Cette liste sera séparée de son premier élément tout au long de la restauration (sera donc vide sauf erreur à la fin de celle ci).

## B) Affichage des informations

Nous avons dû ruser pour afficher les propositions logiques tel qu'on les aperçoit dans la fenêtre de saisie (symbole  $\wedge$  pour and,  $\rightarrow$  pour imply etc),

Or pour afficher en *LaTeX* nous ne pouvions faire passer de simple String à nos listes d'affichage, il nous fallait convertir ces chaînes en *LaTeX* et installer le résultat sous forme d'`ImageIcon` dans le label.

On a donc commencé par créer une classe `FormulaLabel` étendant la classe `JLabel` qui prend en paramètre un String `ch` (correspondant à l'écriture d'une formule) et qui effectue le protocole mentionné juste au dessus.

On a en second lieu créé notre propre rendu de cellule implémentant l'interface `ListCellRenderer`.

La méthode `getListCellRendererComponent` nous retourne donc un `FormulaLabel` à qui on a envoyé la valeur castée en String (correspond à `ch`) paramétré de manière à ce qu'il change de coloris après une sélection.

On installe ce rendu à l'instanciation de la liste de formules dans le `initComponents` des panels du joueur et de l'arbitre.

Mon collègue s'est servi de l'idée pour afficher les arguments et les requêtes (cf : son rapport)

## C) Option pour lancement rapide d'une partie

Nous avons dès le départ adapté notre code de manière à ce qu'un simple clic sur un bouton dans le menu nous lance automatiquement une partie avec des paramètres configurés par défaut :

- ✓ noms de joueurs : Player 1 et Player 2
- ✓ temps de paroles à 250 secondes
- ✓ son activé
- ✓ aucun thème chargé
- ✓ personne n'a encore la main

Le procédure normale consistait à inscrire chaque joueur en cliquant sur un bouton et à entrer un pseudonyme non vide puis cliquer sur NewGame de l'arbitre.

Cela faisait un nombre trop important d'étapes pour effectuer des tests fréquents.

Ceci peut paraître inutile une fois que la sauvegarde et restauration sont parfaitement fonctionnels dans la mesure où il est possible de charger une partie configurée et rédigée par soi-même dans un éditeur de texte.

Mais on s'est attaché à la restauration/sauvegarde assez tard du moins pas avant que le reste ne fonctionne.

## D) Difficultés rencontrées

### a) Clonage

Lorsqu'un joueur soumettait une formule ou un argument ne respectant pas la cohérence, il était impossible pour les deux agents par la suite d'en saisir un ou une de nouveau, tout étant refusé à tort.

Restant à cours d'idées pour corriger le problème, j'ai implémenter l'interface Cloneable dans SATSolverSaT4J, et dans le constructeur des fenêtres de saisie, on ne réalise pas une copie directe du paramètre (`this.solver=playersolver`) mais bien un clonage de ce paramètre (`this.solver=(SATSolverSAT4J)player.clone()`).

### b) Restauration dans la classe principale

L'idéal aurait été d'avoir comme pour la sauvegarde une méthode dans une classe délocalisée qui reçoive les trois agents (2 joueurs + arbitre) en paramètre et qui modifierait ceux ci en fonction des informations du fichier texte.

Or en fonctionnant de la sorte, des comportements inattendus apparaissaient.

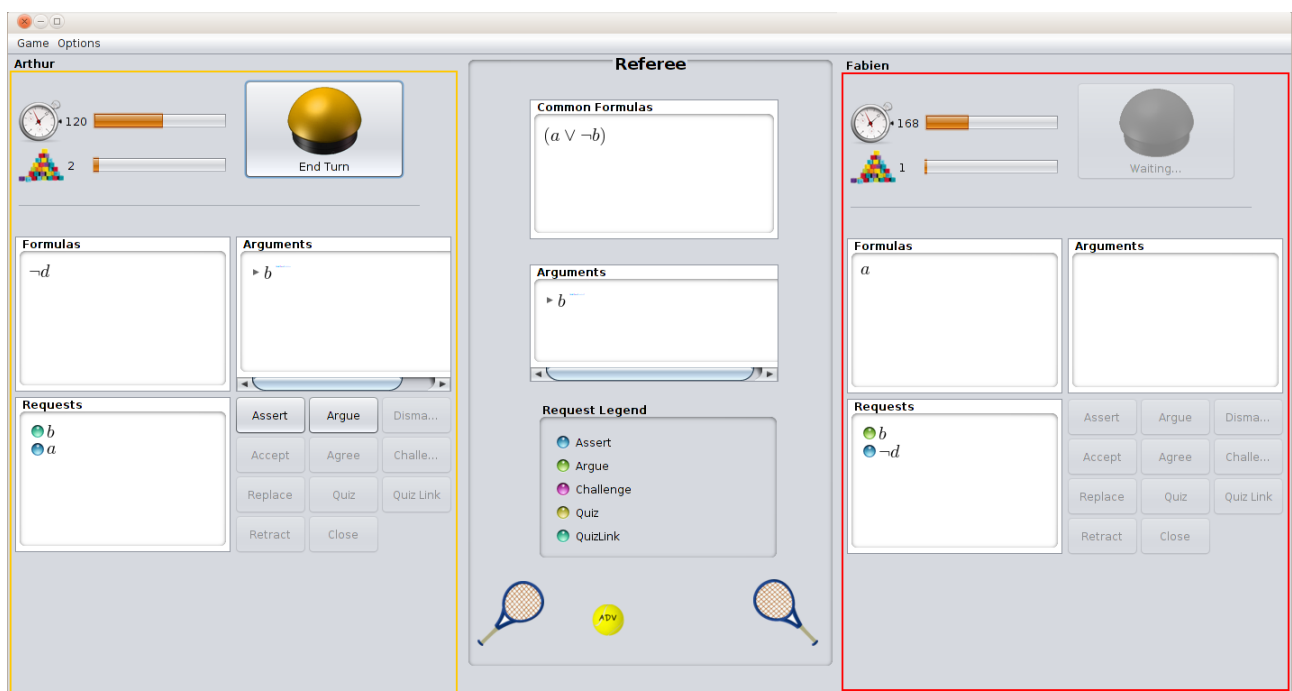
Malgré divers essais avec des *final*, des *static* ou des clones, le problème persistait et nous avons donc déplacer les fonctions propres à la restauration dans la classe principale DebateFrame même si notre schéma d'architecture ne le prévoyait pas comme cela.

### 1. *Satisfactions*

Nous sommes dans l'ensemble assez satisfaits du produit que nous avons conçu.

Nous avons pu instaurer des éléments que nous n'avions pas défini à la spécification et à la définition des besoins.

- ✓ affichage des arguments sous forme d'arbre déroulant
- ✓ association de chaque type de requête avec sa couleur (précisé dans une légende)
- ✓ affichage de l'évolution du score
- ✓ son



Copie d'écran de la GUI

### 2. *Regrets*

Ceci dit nous n'avons pas recouvert l'ensemble des besoins que l'on avait évoqué lors des spécifications comme l'applet JAVA et les deux éléments ci-dessous.

#### A) L'intelligence artificielle

Plusieurs raisons peuvent expliquer cela, on était assez juste en délai et le code n'est pas parfaitement adapté pour intégrer une IA rapidement.

Sans compter qu'elle aurait été trop perfectible:

- ✓ la génération aléatoire de formules étant particulièrement difficile, la génération de formules concises et cohérentes l'est d'autant plus.
- ✓ difficulté pour définir une stratégie d'argumentation ou de réplique

✓ besoin d'une fonction « decode » qui à partir d'un support calcule si la conclusion peut être prouvée.

On arrivait là à la limite de nos compétences.

## **B) Le compte-rendu**

Nous comptons au début enregistrer au fur et à mesure dans un fichier ce qui se passait en rendant le dialogue parfaitement audible et naturel.

Par exemple pour un « QuizLink » on aurait écrit « Player 2: Je ne vois pas le rapport avec le sujet du débat Mr » ou « Player 1: Je suis d'accord quand au fait que ... entraine ... ».

Mais le plus compliqué résidait dans la traduction de formules.

Nous n'avons pas trouvé de solutions fiables pour « parser » une formule.

Sans doute qu'une méthode récursive peut le permettre pour n'importe quelle profondeur mais je n'ai pas su la développer.

## V- Conclusion

Ce projet m'a permis d'exploiter au maximum les notions apprises dans l'UE d'ouverture du S6 «Programmation événementielle».

Sans avoir suivi cette option le sujet m'aurait de toute façon effrayé de par sa difficulté au niveau de la gestion des formules ne sachant pas encore que SAToulouse pouvait être utilisé, et je me serais penché soit sur un autre TER soit sur des entretiens dans des entreprises.

Là je savais que la gestion des événements et l'affichage pouvaient être développés correctement.

Je disposai des bases pour rendre le jeu très interactif, et tout ce qui allait concerner l'interface allait peut-être me demander une lecture de la *JavaDoc* ou de forums, mais le bagage suffisant pour ne pas nager m'avait été donné depuis trois mois.

Les nombreux TP nous ont habitué à intégrer des nouvelles bibliothèques, de les utiliser sans même connaître leur particularité sur le bout des doigts. La mécanique d'utilisation de JAVA me semble encore plus importante que la connaissance des différentes bibliothèques.

La principale difficulté au niveau du codage résidait dans la gestion des formules, leur saisie, la vérification de la consistance, leur affichage etc.

Mais je dois l'avouer et remercier les auteurs, le logiciel SAToulouse dont j'ai pu assez rapidement me familiariser tant le code source est parfaitement organisé, clair et commenté m'a rendu la tâche plus facile et nous a fait gagner un temps précieux.

J'ai appris également à résumer mon avancée hebdomadaire, à la présenter à mon partenaire et tuteur.

Je me suis habitué à prendre en compte leurs remarques, convenir avec eux d'une évolution.

En bref ce travail m'a donné une idée de tous les aspects que la gestion d'un projet recouvre dans une entreprise. Le codage prend une place importante mais pas hégémonique.

La part de coût, faisant parti des trois notions clés de la gestion d'un projet était ici à omettre, travaillant au bénévolat et n'utilisant que des logiciels libres pour le réaliser.

## VI- Glossaire

**API:** interface fournie par un programme informatique. Elle permet l'interaction des programmes les uns avec les autres, de manière analogue à une interface homme-machine, qui rend possible l'interaction entre un homme et une machine.

Du point de vue technique une API est un ensemble de fonctions, procédures ou classes mises à disposition par une bibliothèque logicielle, un système d'exploitation ou un service. La connaissance des API est indispensable à l'interopérabilité entre les composants logiciels.

**final:** indique qu'un élément ne peut être changé dans la suite du programme. Il peut s'appliquer aux méthodes et attributs d'une classe ainsi que la classe elle-même. Selon le contexte, on utilise final dans un souci de conception ou d'optimisation. C'est pourquoi il est parfois employé de façon incorrecte.

**GUI:** Graphical User Interface

**initComponents:** méthode liée au plugin Matisse de Netbeans, comprenant l'interprétation en code de ce que l'utilisateur a créé graphiquement.

**Javadoc:** outil développé par Sun Microsystems permettant de créer une documentation d'API en format HTML depuis les commentaires présents dans un code source en Java.

**JDialog:** permet de créer des boîtes de dialogue, qui permettent des interaction avec l'utilisateur de l'application. Les dialogues sont des conteneur de premier niveau (comme *JFrame*) : ils ne sont pas contenus dans d'autres cadres, mais dépendent d'un autre cadre.

**JFileChooser:** permet de sélectionner un ou plusieurs fichiers à partir du système de gestion de fichiers.

**JFrame:** permet de créer et de gérer des objets de type cadre ou fenêtre. En tant que conteneur, elle peut contenir d'autres composants d'interface tels que des boutons, des zones de saisie etc.

**JLabel:** sert à afficher du texte sur une interface ou de contenir une icône.

**JList:** La classe JList permet d'afficher une liste d'objets, et offre à l'utilisateur la possibilité de sélectionner un ou plusieurs objets de la liste.

Un JList peut être créé à partir d'un tableau d'objets, ou à partir d'un Vector, mais si on veut une liste qui peut être mise à jour par programme, il faut utiliser un ListModel.

**LaTeX:** méthode privilégiée d'écriture de documents scientifiques employant TeX. Il est particulièrement utilisé dans les domaines techniques et scientifiques pour la production de documents de taille moyenne ou importante (thèse ou livre, par exemple). Néanmoins, il peut être aussi employé pour générer des documents de types variés (par exemple, des lettres, ou des transparents).

**ListCellRenderer:** interface Swing contenant la méthode `getListCellRendererComponent()` retournant le composant à afficher dans la *JList*.

**LookAndFeel:** ensemble des spécificités et caractéristiques d'une interface qui lui donnent une identité et qui peut être perçu de différentes manières selon les utilisateurs. L'apparence de ces interfaces est principalement caractérisée par des paramètres de base comme les polices, les formes, les couleurs et la disposition des éléments. La perception et le ressenti sont quant à eux

plus influençables par l'interaction qui est caractérisée par d'autres paramètres comme les boutons et les menus.

Matisse: WYSIWYG (What you see is what you get) pour créer des interfaces en Java Swing ou AWT.

src: dossier d'un projet Eclipse ou Netbeans contenant l'ensemble des paquets du projet contenant eux même le contenu de chaque classe sous l'extension des .java, et un fichier supplémentaire .form pour toute classe ayant demandé l'utilisation de Matisse.

static: indique que la variable qui suit n'appartient pas à une instance particulière de la classe. Les variables ou méthodes statiques appartiennent à la classe elle-même. On peut ainsi les utiliser sans avoir une instance créée.



## VII- Annexe

✓ Notice du jeu