CATELLA Solène
GR3

27/04/2017

PROJET INFORMATIQUE INDIVIDUEL

Livrable

Salvod'art

ENSC - 2016/2017

Table des matières

I. Rappel du sujet

Salvod'art est un système expert spécialisé dans le domaine des arts, et plus précisément celui de la peinture. Équivalent d'un Akinator des œuvres picturales, ce petit génie représenté sous les traits de Salvador Dali se fait fort de deviner le tableau auquel un utilisateur songerait. Quinze à vingt questions suffisent en moyenne pour que le système soit en mesure d'apporter une réponse concluante de part sa capacité d'inférence. En cas d'échec, hypothèse qui demeure hautement probable compte-tenu de l'immensité du champ des possibles, l'utilisateur est invité à enrichir la base de connaissance du système, seul moyen pour ce dernier d'apprendre par lui-même, de façon dynamique.

II. Spécifications générales

1. Description du système

Le programme informatique est en mesure, à partir d'une base de faits et de règles, d'identifier une œuvre picturale précise.

Lorsque l'identification réussit, autrement dit lorsque le système est en mesure, de par les réponses données par l'utilisateur, de retrouver une œuvre déjà enregistrée en mémoire, celle-ci est affichée à l'écran : une fiche technique est alors accessible depuis ce même menu via un clic sur image.

En revanche, si le système ne parvient pas à identifier à temps (c'est-à-dire après 25 tours) l'œuvre en question, l'utilisateur peut alors saisir le nom de l'œuvre à laquelle il pensait, son auteur, et éventuellement préciser une question qui aurait pu avoir du sens dans cette identification. Dans ces conditions, le système est alors capable d'apprendre par lui même, sur la base des informations lui étant fournies par ses différents utilisateurs.

2. Description des données

Le système prend en entrée les données relatives aux questions. Le choix de la question pertinente à poser à un moment précis du jeu est déterminant puisqu'il oriente la conduite de l'interrogatoire de façon plus ou moins judicieuse. Au fur et à mesure de la recherche, les questions se font de plus en plus précises, sans pour autant être trop spécifiques, l'idéal étant de couvrir le plus de cas possibles tout en parvenant à solutionner l'énigme en un nombre de coups minimum.

Les données fournies en sortie sont celles relatives aux œuvres, stockées dans la table *Œuvre*. La plupart des tableaux ont été saisis manuellement, directement dans la base ; les autres ont été entrés par l'utilisateur lui-même, via le menu mis en place à cet effet.
La base de données contient à ce jour 125 œuvres de 45 peintres différents, aussi diverses que possible.

3. Exigences techniques

a. Contraintes

L'application a été développée en C#, un langage qui se prête bien au développement d'applications (technologie WinForm) et qui permet de manipuler des bases de données via l'outil de requêtage LINQ.

b. Exigences fonctionnelles et non fonctionnelles

Désignation : *Capacités de résolution et d'apprentissage du système*

Description : Le système sera capable d'inférence à partir des réponses qui lui seront données par l'utilisateur. Si le système échoue à la résolution de l'énigme, il sera néanmoins apte à faire évoluer sa base de connaissance via ses interactions avec l'utilisateur.

Désignation : *Format des questions*

Description : Les questions posées à l'utilisateur seront nécessairement fermées, de façon à contraindre les choix de réponses possibles. Elles devront être concises et non ambiguës pour que l'utilisateur s'engage au maximum dans ses choix de réponses ("Oui" ou "Non") et soit le moins de fois possibles dans l'obligation d'avouer son ignorance en choisissant la solution "Je ne sais pas".

Désignation : *Format des réponses*

Description : Trois choix de réponses possibles seront communs à l'ensemble des questions : "Oui", "Non", et "Je ne sais pas".

Désignation : *Durée de l'énigme*

Description : La résolution de l'énigme devra pouvoir se faire en un nombre de coups minimum, le maximum étant de 25. Plus la durée de l'énigme sera importante, moins le système pourra être considéré comme crédible.

III. Description de l'architecture de l'application

1. Architecture des données et relations inter-tables

Les données relatives aux œuvres, aux questions, aux connaissances ainsi qu'aux réponses, sont respectivement stockées dans les tables *Œuvre*, *Question*, *Connaissance* et *Reponse*.

Table « Œuvre » :

Les données concernant les œuvres ont dans un premier temps été saisies manuellement, et ce pour pouvoir tester l'algorithme mis en place. Chaque œuvre est nécessairement identifiée par un numéro (ID_0) et un nom (Nom). Les champs relatifs au peintre (Artiste), au genre (Genre), à la description de l'œuvre (Descriptif) et à l'image associée (Img) sont eux facultatifs. Deux paramètres supplémentaires entrent en jeu dans la logique même de l'algorithme :

- Un booléen (Flag) permettant de dissocier, sur la base des réponses données par l'utilisateur aux questions lui étant posées, les œuvres encore en lice de celles qui ne le sont supposément plus. Au début du jeu, le flag de toutes les œuvres est passée à *true* ;
- Un entier comptabilisant le nombre de fois qu'une œuvre a été faite devinée (Occurrence), et à partir duquel il sera possible, en cas d'égalité de proposition, de départager une ou plusieurs œuvres candidates (l'œuvre de plus forte occurrence étant évidemment préférée).

Table « Question » :

Une question est caractérisée par un identifiant unique (ID_Q), un libellé sous forme de mot clef (Libelle) et éventuellement un type (Type) pour pouvoir permettre au système d'identifier ce sur quoi portera la question et d'adopter une formulation de question conséquente (genre, personnage, peintre, accessoire, tons, attitude, lieu, atmosphère, etc.).

193 questions ont été enregistrées en mémoire ; toutes ont été saisies manuellement. Lorsque cela était possible et suffisamment pertinent, des regroupements ont été effectués (ex : lac/fleuve/mer, église/cathédrale, bateaux/barques/voiliers, etc.).

Table « Reponse » :

Cette table comporte seulement deux colonnes :

- Une première colonne relative à l'identifiant de la réponse ;
- Une deuxième colonne permettant d'associer à un identifiant donné une certaine réponse (Oui = 1, Non = 2 et Ne Sais Pas = 3).

Table « Connaissance » :

Cette table joue un rôle central puisqu'elle fait le lien entre toutes celles précédemment citées. Chaque connaissance est caractérisée par un identifiant unique (ID_C). Le reste des champs de la table Connaissance sont des clefs étrangères faisant référence aux clefs primaires des tables *Question*, *Reponse* et *Œuvre*. Ainsi, chaque connaissance enregistrée fait référence pour une œuvre donnée à une certaine question et à une certaine réponse.

Le schéma relationnel des données présenté ci-contre permet de mieux apprécier l'architecture de la base telle qu'elle a été conçue :

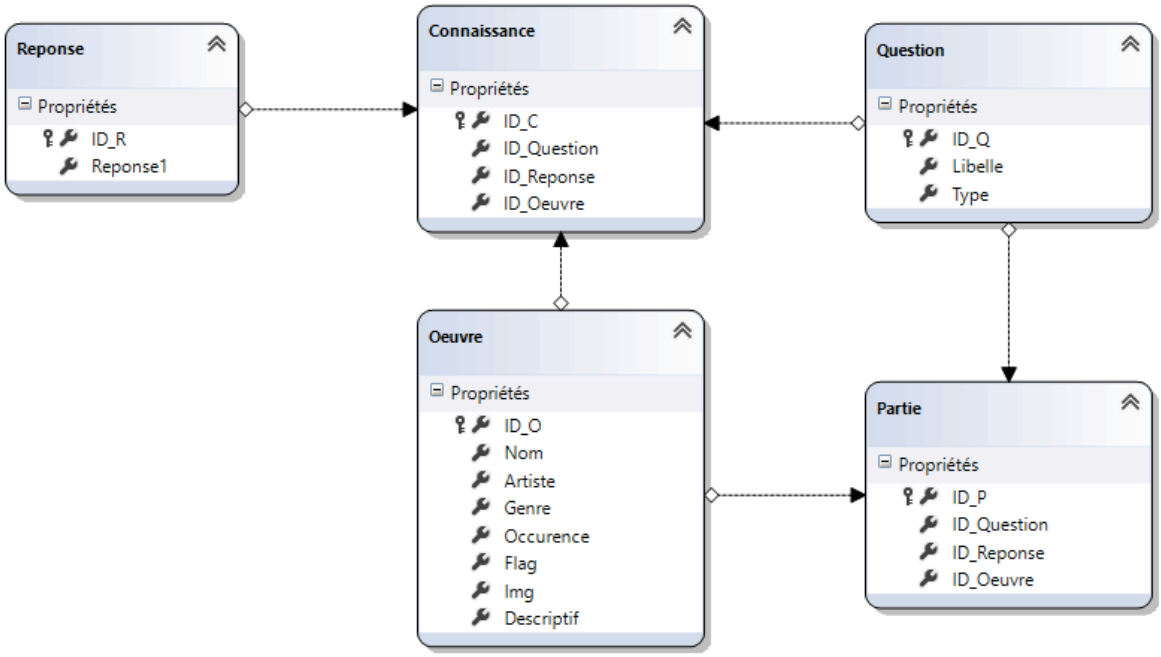

Schéma relationnel de la base de données

NB : La table « Partie » mise en relation avec les tables *Œuvre* et *Question* n'a pas été mise à contribution dans le programme, faute de temps. Comme son nom l'indique, celle-ci était supposée garder en mémoire l'historique de chaque partie (questions posées, réponses données, œuvre ou non identifiée). Cela aurait en effet permis d'avoir un référentiel pour chaque œuvre ayant été faite devinée, et ainsi comparer d'un jeu à l'autre les réponses données par les différents utilisateurs à une même question et pour un même tableau de peinture. Il aurait alors sans doute été possible de vérifier la bonne correspondance entre le répondu et l'attendu pour une œuvre donnée, et ainsi identifier d'une part les questions les plus conflictuelles -c'est-à-dire les questions les plus équivoques- et de vérifier d'autre part la malhonnêteté d'un joueur qui s'amuserait à conduire en erreur le système.

2. Formulaires, classes et fonctions

Le programme tel qu'il a été conçu ne comporte qu'une seule véritable classe dans laquelle sont manipulées les œuvres et les questions : il s'agit de la classe *GestionQuestionOeuvre*. Toutes les fonctions mobilisées lors du déroulement algorithmique s'y trouvent.
On distingue :

1. Des fonctions d'initialisation

    - *InstancierTableauOeuvres() :* cette fonction se charge de l'instanciation d'une matrice [nbOeuvres*3], dans laquelle sont stockés, pour chaque œuvre, un coefficient de pondération fixe et un indice de correspondance variable entre le répondu et l'attendu. Le premier coefficient est calculé sur la base des questions et des réponses déjà renseignées dans la base pour chacune des œuvres. Ainsi, si 4 questions ont été renseignées pour un même tableau, le coefficient de pondération de celui-ci vaudra 100/4 = 25%. Autrement dit, à chaque bonne réponse donnée par l'utilisateur pour ces dites questions, la probabilité qu'il s'agisse de la bonne œuvre à identifier grimpera d'1/4 de plus. Lorsqu'au bout d'un certain nombre de tours cet indice de correspondance égalise ou dépasse les 75%, l'œuvre est faite candidate.
      NB : Ces paramètres sont stockés dans le tableau 2d oeuvresEnCriticite.

    - *RecenserGenresPeintresPossibles()* : cette fonction permet de recenser tous les peintres et tous les genres distincts enregistrés dans la base pour ensuite les stocker respectivement dans les listes peintres et genres.

2. Des fonctions relatives au choix et à la formulation des questions

    - *PoserQuestionV1(out int numQ)*, *PoserQuestionV2(out int numQ, int numEtape)* dans lesquelles se situe toute l'algorithmie relative au tirage de la bonne question à poser.
        - La fonction *PoserQuestionV1()* est la première version de l'algorithme de tirage de questions mis au point. Ce premier modèle algorithmique comptabilisait pour chaque question n'ayant pas encore été posée le nombre de oui d'une part (compteOuiQues) et de non d'autre part (compteNonQues) ayant été renseignés pour chaque œuvre encore en jeu. La question qui était sélectionnée était alors celle enregistrant le plus grand nombre de oui ou de non, c'est-à-dire la question permettant d'éliminer le plus d'œuvres possibles pour une réponse donnée.

Etant donnée que ce problème est stochastique (on ne connaît pas à l'avance la réponse que l'utilisateur donnera) cet algorithme n'était pas optimal, bien que fonctionnel. C'est la raison pour laquelle une deuxième version a été mise au point.

- La fonction *PoserQuestionV2(out int numQ, int numEtape)* n'élit pas la question la plus renseignée toutes œuvres confondues, mais celle qui, à chaque nouveau tour, permet d'éliminer le plus de candidats possibles et ce quelle que soit la réponse donnée par l'utilisateur.
  Cette fonction prend en paramètre un numéro de question (celui de la question qui sera posée) et un entier relatif au numéro d'étape en cours. Ce numéro d'étape permet de faire une proposition de dernier recours, dans le cas de figure où le système n'aurait toujours pas identifié, un tour avant la fin du jeu, une œuvre potentielle. Est alors choisie l'œuvre dont l'indice de correspondance est le plus important. Le reste du temps, autrement dit sur toutes les étapes précédentes, le choix de la bonne question se fait comme tel :
    - Sont respectivement stockés pour chaque question n'ayant pas encore été posée le nombre de oui+neSaisPas d'une part et le nombre de non+neSaisPas d'autre part dans deux dictionnaires distincts.
    - Une première étape consiste à ordonner les questions selon le nombre de tableaux éliminés au total, c'est-à-dire en considérant la somme des oui, des non et des neSaisPas. Ces questions ordonnées sont provisoirement stockées dans une liste de Question.
    - Une fois ce premier tri effectué, on détermine les questions pertinentes de celles qui ne le sont pas. Une question dite « pertinente » est une question qui, pour un taux de couverture suffisamment intéressant, permet, quelle que que soit la réponse donnée par l'utilisateur, d'éliminer autant de tableaux d'un côté que de l'autre. Le taux de recouvrement minimal est recalculé à chaque étape : il est fonction du nombre d'œuvres n'ayant pas encore été éliminées et du nombre de tours qu'il reste avant que le jeu ne prenne fin. Une « bonne » question est d'abord une question dont la somme des oui, des non et des neSaisPas

est supérieure à ce taux de recouvrement. Mais cela ne suffit pas, dans la mesure où si on ne se contentait que de ce seul critère, on ne tiendrait pas compte des déséquilibres qu'il pourrait y avoir entre le nombre de oui et le nombre de non pour chacune des questions. La question idéale est bien évidemment la question qui enregistre autant de oui que de non (différence nulle). Ce cas de figure étant relativement rare, toutes les questions enregistrant le plus faible écart possible sont considérées comme étant pertinentes (l'écart oui-non ne peut ici pas être supérieur au taux de recouvrement minimal).

- Vient ensuite la deuxième étape au cours de laquelle un second et dernier ordonnancement a lieu. Il s'agit de classer les unes par rapport aux autres les questions de même taux de couverture. Ce deuxième tri se fait alors sur l'écart oui-non précédemment calculé : pour un même taux de couverture, une question équilibrée sera évidemment plus intéressante qu'une question plus ou moins déséquilibrée.
- La « bonne » question à poser est alors celle qui apparaît en première position dans la liste des questions triées. Si aucune des questions ne satisfait la double condition taux de recouvrement minimal/équilibre des réponses, c'est la question la plus renseignée qui est alors élue (cf. premier algorithme). Lorsque toutes les questions sont renseignées à égale mesure, un tirage au sort a lieu pour déterminer laquelle de celles-ci sera posée à l'utilisateur. Il aurait évidemment été préférable de raisonner de façon récursive, en considérant chacune des questions qui aurait été posée après que l'utilisateur ait donnée sa réponse à la question courante (cf. V).

- FormulerQuestion : c'est dans cette fonction que la question à poser à l'utilisateur, sur la base de son identifiant et de son type, est formulée. Une instruction « Switch case » permet de répertorier tous les différents cas de figure, et d'humaniser davantage la façon dont le système s'adresse à l'utilisateur.

3. Des fonctions relatives au choix et à la formulation des questions

   - *MemoriserQuestionReponse(int numQ, int numR) :* comme son nom l'indique, cette fonction a pour objectif d'enregistrer en mémoire les réponses données par l'utilisateur à chacune des questions lui étant posée dans un dictionnaire prévu à cet effet. La question *VerifierQuestionDoublon()* y est aussi appelée.

   - *VerifierQuestionDoublon()* : une fois que la réponse de l'utilisateur donnée à une question est récupérée, on vérifie si d'autres questions et d'autres œuvres peuvent indirectement être éliminées. Les questions relatives au genre de la peinture (portrait, paysage, scène de vie, scène d'intérieur, tableau historique, etc.) et aux peintres sont concernées par cette approche. Dans le cas de figure où l'utilisateur confirmerait l'identité du peintre, toutes les autres questions relatives aux autres peintres et toutes les œuvres n'étant pas de ce peintre se verraient alors écartées. Un tel traitement a lieu dans la fonction EliminerQuestionsOeuvresDoublons(*List<int> peintresOuGenres, int num*).

4. Des fonctions relatives à la réactualisation des questions à poser et à la réaffectation de la criticité des œuvres candidates.

   - *EliminerQuestionsOeuvres(int numQ, int numRep) :* cette fonction vise à éliminer les œuvres ne pouvant plus être considérées comme candidates à l'issue de la réponse donnée par l'utilisateur (leur flag passe à *false*). La question ayant été posée est elle aussi éliminée, de façon à ne pas pouvoir être posée une seconde fois.
   - *EliminerQuestionsOeuvresDoublons(List<int> peintresOuGenres, int num)* : comme précisé plus haut, cette fonction a pour vocation d'éliminer les questions doublons et les œuvres « hors genre » ou « hors peintre » à l'issue de ce qu'a répondu le joueur.

   - *AffecterCriticiteOeuvre(int reponse, int question, int[,] œuvres)* : c'est dans cette fonction que la réaffectation de la criticité/pertinence de chaque œuvre est effectuée. Un requêtage sur la table Connaissance permet d'identifier toutes les œuvres concernées par le combiné question/réponse. Celles-ci voient alors leur indice de pertinence augmenter de leur coefficient de pondération respectif (cf. *InstancierTableauOeuvres()*). Les œuvres dépassant un indice de criticité supérieur ou égal à 60% sont stockées dans une liste d'œuvres probables. Initialement, cette liste avait été pensée pour l'application d'heuristiques (convergence vers une œuvre probable au bout d'un certain nombre de tours par exemple).

5. Des fonctions de fin de partie

    - *DeciderOeuvreCandidate()* : cette fonction permet d'élire comme œuvre candidate celle enregistrant la plus forte occurrence via la fonction *ChoisirOeuvreParOccurence(int ocurrence).* En cas de stricte égalité, l'œuvre se voit être tirée au hasard.

    - *ChoisirOeuvreParOccurence(int ocurrence)* : cette fonction permet, comme son nom l'indique, de choisir l'œuvre probable de plus forte occurrence. Ainsi, si à la fin du jeu deux œuvres enregistrent exactement le même indice de criticité, celle ayant été faite devinée le plus grand nombre de fois sera évidemment privilégiée.

NB : ces deux fonctions sont opérationnelles, mais n'ont pas été implémentées à temps dans la fonction PoserQuestionV2. Il aurait pourtant seulement suffit, dans le cas où aucune proposition n'aurait été faite un tour avant la fin du jeu, d'appeler la fonction *DeciderOeuvreCandidate().*

    - *ArreterPartie(int[,] tab, int numEtape, out int numTab)* : cette fonction est appelée à chaque nouveau tour. Elle permet de rechercher, à l'issue de la réponse donnée par l'utilisateur, si une proposition de réponse peut être faite. Ainsi, pour toutes les œuvres n'ayant pas encore été éliminées, on vérifie si l'indice de criticité de chacune d'entre elles dépasse 75% : si c'est le cas, l'œuvre fait l'objet d'une proposition. Sinon, s'il ne reste plus aucune œuvre, est choisie celle qui semble le plus se rapprocher des réponses ayant été données par le joueur.

    - *EliminerTableau(int oeuvre)* : comme son nom l'indique, cette fonction est chargée d'éliminer un tableau en particulier. Elle est appelée lorsque le joueur réfute la proposition de réponse faite par le système.

Les formulaires sont quant à eux au nombre de quatre :

- Un formulaire qui fait office de page d'accueil : *MenuPrincipal* ;
- Un formulaire dans lequel se fait l'affichage des questions posées à l'utilisateur : *MenuQuestion* ;
- Un formulaire final qui peut revêtir deux formes, selon que l'identification a échoué ou non : *MenuResolutionEnigme*. Dans un cas comme dans l'autre, c'est à partir de ce menu que l'enregistrement des questions et réponses pour une œuvre donnée est réalisé *;*
- Et enfin le formulaire *InformationsTableau*, qui permet d'afficher les données relatives à un tableau si celui-ci a correctement été identifié.

3. Structure algorithmique

En chaque début de nouvelle partie, le tableau des œuvres est instancié, les peintres et genre recensés. Un appel à la fonction PoserQuestionV2 est effectué tant que la partie n'est pas terminé (nombre de tours < 25). En fonction de la réponse qui a été donnée par l'utilisateur, un certain nombre d'œuvres est éliminé. Celles restant en compétition voient de leur côté leur indice de criticité augmenter ou demeurer inchangé. Une fois cette réaffectation opérée, la fonction ArreterPartie() permet de vérifier si une œuvre est d'ores et déjà candidate. Si c'est le cas, l'interface est redessinée via l'appel de la fonction « DessinerInterfaceProposition() ». Plusieurs cas de figure sont distingués selon le nombre de tentatives de réponses ayant déjà été réalisées. Si la proposition n'est pas la bonne mais qu'il s'agit seulement de la première tentative, l'œuvre est éliminé et le jeu reprend son cours. S'il est question en revanche de la deuxième tentative, et que celle-ci se solde encore par un échec, le jeu prend fin, quelle que soit l'étape à laquelle ce dernier se trouve.

II. Résultats et tests

Il est bien évidemment difficile d'évaluer objectivement la pertinence et l'efficacité de cet algorithme sur la base de seulement 125 tableaux. Cependant, les résultats obtenus à l'issue de l'implémentation du second algorithme sont plutôt satisfaisants et conformes à ce à quoi on pouvait s'attendre. Si l'identification n'aboutit pas toujours dans certains cas (nombre de « je ne sais pas » trop important, œuvres plus ou moins bien renseignées, etc.), il demeure que la logique de recherche finalement mise en place s'avère plus performante qu'elle ne l'était auparavant (cf. algorithmie de PoserQuestionV1()).
Aussi, les principales exigences ont été relevées :

- Le programme est bien capable d'inférence à partir des réponses lui étant données par l'utilisateur ;
- Le système est en mesure d'apprendre de ses interactions avec les différents utilisateurs
- La restructuration du savoir via le réordonnancement des questions posées à l'utilisateur est effectif
- Et enfin, la limitation de la résolution de l'énigme à un nombre fini de questions a été respectée.

Quelques tests ont été réalisés en fin de développement, après implémentation du second algorithme. Ces derniers ont permis de relever certains bugs (arrêt brutal du jeu sous conditions particulières entre autres), mais aussi de mieux apprécier l'intelligibilité ainsi que la subjectivité inhérentes aux différentes questions posées. Certaines questions ont unanimement

fait l'objet d'ambigüités, parmi lesquelles celles liées au mouvement de peinture (la plupart des utilisateurs n'avait aucune connaissance à ce sujet) ou encore aux tons et couleurs perçus.

III. Bilan et perspectives

1. Points positifs et négatifs

Points positifs :

- Sélection des questions via leur niveau de discrimination et leur taux de couverture ;
- Capacité du système à faire évoluer sa base de connaissance ;
- Bonne capacité du système à reconnaître un tableau, et marge de tolérance à quelques erreurs près ;
- Utilisation de l'outil de requêtage LINQ, permettant d'interroger la base et transformer les données avec efficacité ;
- Bonne structuration de la base de données ;
- Bonne humanisation du système (questions formulées par « type ») ;

Points négatifs :

- Structuration du code au sein d'une même classe qu'on pourrait qualifier de « fourre-tout », regroupant à la fois la gestion des œuvres et des questions, tendant à rendre le code peu lisible ;
- Saisie des données par l'utilisateur insuffisamment verrouillée (pas de moyen possible de vérifier la justesse de ce qui est saisi dans la BDD, état pouvant conduire à des biais importants)
- Absence de recul sur le remplissage de la base de données, ayant conduit à sous-évaluer l'efficacité de l'algorithme initialement mis en place d'une part et à formuler des questions trop équivoques d'autre part (hypothèses faites sur les connaissances des joueurs en art évaluées trop tardivement).

2. Evolutions possibles

Plusieurs évolutions du programme sont bien sûr à envisager, celui-ci étant loin d'être abouti.
D'abord pour ce qui est de la structuration du code : en effet, ce dernier peut sembler à plusieurs égards peu orienté-objet. Il aurait fallu, au fur et à mesure du développement, repenser l'architecture du programme, et proposer une structuration de classes plus intéressante. Cela aurait pu passer par l'exploitation plus approfondie des classes d'entité LINQ

to SQL qui sont mappées à la base de donnée. À ce jour, seule la classe Question a été pensée comme telle. Il aurait fallu en faire de même pour la classe Œuvre.

Autre point : le tirage des questions aurait pu être affiné, la sélection des questions les plus pertinentes est encore loin d'être optimale. Un deuxième niveau de difficulté visant à évaluer, par récursivité, la meilleure question à poser, aurait été idéal. Néanmoins, le manque de recul sur le remplissage de la base de données et sa taille relativement réduite n'ont pas aidé à apprécier les développements réalisés. Des sessions de test plus nombreuses auraient sans doute permis de relever des défauts de conception et de corriger certains biais liés à ma propre connaissance sur le sujet.

Aussi, l'implémentation de nouvelles heuristiques se serait avérée très intéressante. Ce type d'algorithme laisse le champ ouvert à beaucoup de possibles et de nombreuses possibilités. La capacité du système à appréhender l'honnêteté du joueur et à contrer sa mauvaise foi est un des points sur lequel j'aurais aimé travailler.

De façon plus anecdotique, le chargement des images dans la base de données via l'interface-utilisateur aurait été un plus.

IV. Conclusion

Bien que non complétement abouti, ce projet m'a beaucoup appris, tant au niveau technique (requêtage LINQ) que d'un point de vue plus algorithmique. Le tirage de la question la plus pertinente à poser à l'utilisateur s'est avéré ardue, mais réellement passionnante. Un des points forts de ce projet est de ne m'avoir laissé aucun temps de répit : sitôt résolu un problème en apparaissaient d'autres. La complexité algorithmique d'un tel système, lorsqu'il est suffisamment bien pensé, ne doit pas être sous-estimée. Les possibilités d'améliorations sont en effet infinies, ce qui est simultanément source d'exaltations et de frustrations.