Programmation :
Evaluator 2.0

William De Decker & Mathieu David

28 octobre 2016

ECAM
Brussels Engineering School

1 | Introduction

L'énoncé de ce premier travail nous demandait de reprendre le code écrit pour les labos 2 & 3 et de l'étendre de façon à présenter une interface graphique. Pour faciliter certaines fonctionnalités, il était aussi demandé de modifier les relations entre les différentes classes où nécessaire. Pour ce faire nous avons du modifier quelque peu la structure et les classes du code précédent. Nous détaillerons dans les chapitres suivants les modifications apportées au code ainsi que le fonctionnement de ce dernier.

2 | Mode d'emploi

Lors du lancement de l'application, nous nous retrouvons devant un menu proposant de créer un nouvel "établissement" ou d'importer des données. L'option import rechargera l'objet "établissement" qui aura été exporté lors d'une utilisation précédente de l'application. L'autre option permettra de créer un établissement en repartant de zéro (il n'y aura donc aucun élève, aucun prof et aucun cours déjà présent). Pour choisir une des options proposées dans le menu, l'utilisateur devra entrer le chiffre correspondant à l'action désirée suivit d'un retour à la ligne.

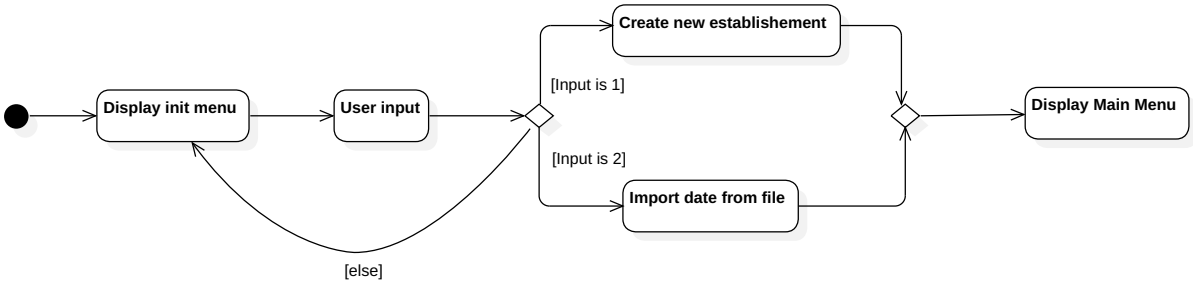
Une fois l'initialisation complète, le menu principal est affiché. La prochaine action est également choisie par l'utilisateur de la même façon. La plupart des actions du menu principal amènent à des sous-menus.

La disposition des menus est montrée dans les annexes section A.1

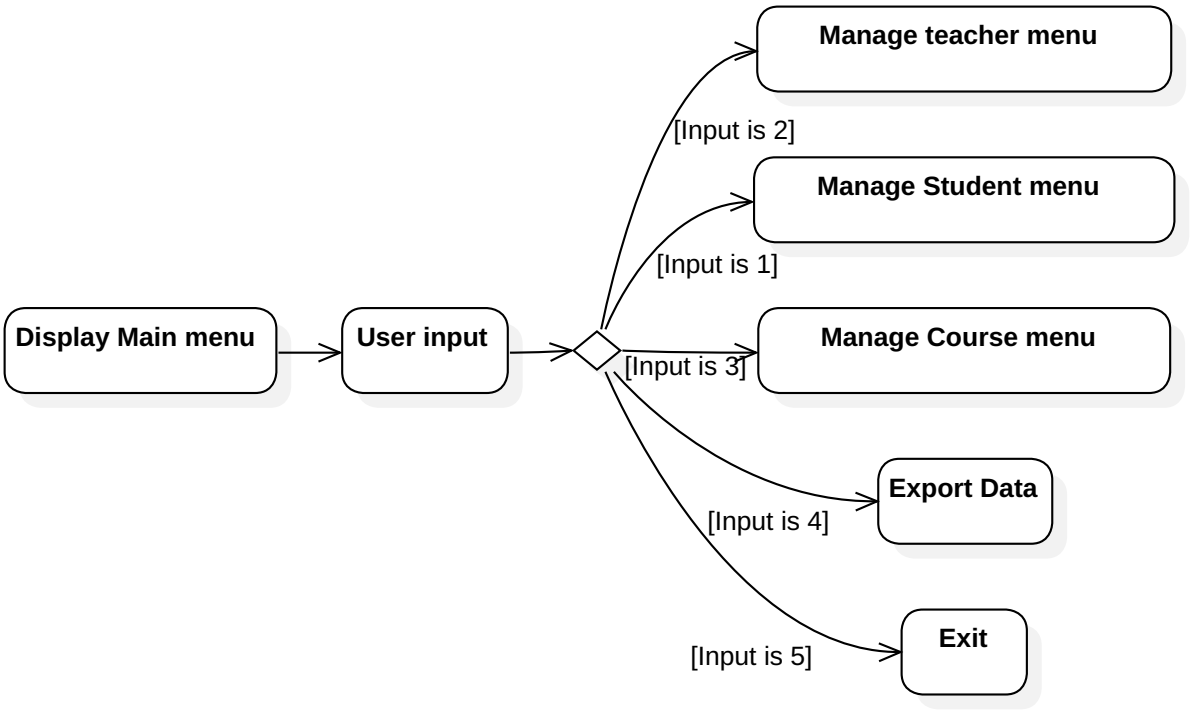
Exemple d'utilisation

Comme exemple d'utilisation de notre application nous vous proposons de créer un nouvel étudiant : *Sébastien d'Oreye*.

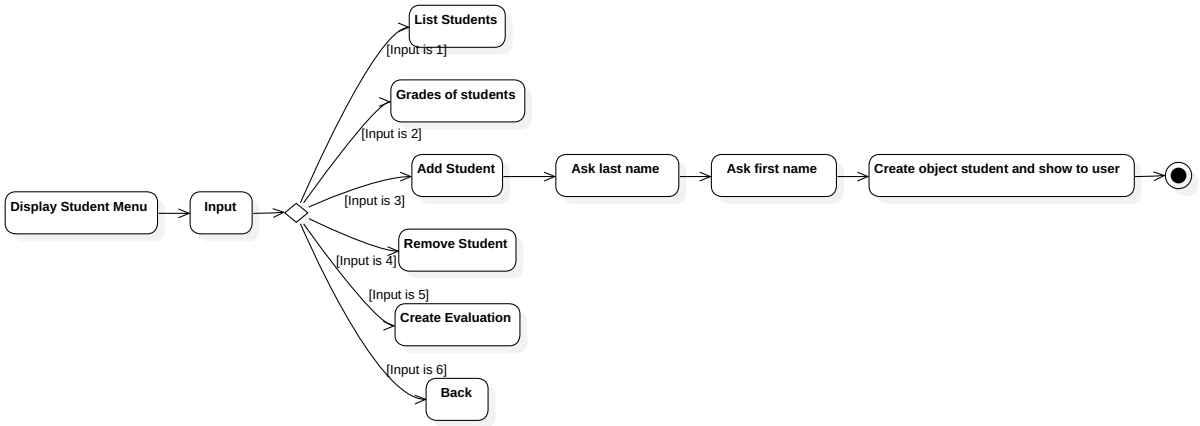
Première étape dans ce processus, charger l'application et choisir entre créer un nouvel établissement ou en importer un déjà existant. Supposons ici qu'il s'agit de la première manipulation de notre application et qu'il n'y ait donc aucun établissement à importer.



Passons cette première étape en revue. Comme le montre ce premier diagramme, lors de l'ouverture de l'application un premier choix s'offre à nous : créer un nouvelle établissement ou en importer un déjà existant. Comme précisé plus haut il n'en existe pas encore à importer, il nous faut donc en créer un nouveau et donc taper 1 dans la console de commande. Celle-ci nous invite alors à choisir un nom pour l'établissement. Laissez donc libre cour à votre imagination, tout est permis ici. Une fois ce nom entré, notre établissement est créé et nous pouvons passer au menu suivant.



Nous nous retrouvons à présent dans le menu principal. A partir d'ici, il n'est plus possible de revenir au menu d'initialisation sans relancer l'application. Le menu principal offre plusieurs choix pour gérer les étudiants, les professeurs et les cours de l'établissement. Comme dans notre cas nous aimerions créer un nouvel étudiant, c'est l'option *Manage Student* qui nous intéresse ici. Tapez donc 1 pour avoir accès au menu gérant les étudiants.



Nous rentrons enfin dans le menu de gestion des étudiants. D'ici vous pouvez créer ou supprimer des élèves, obtenir la liste des élèves déjà existants, obtenir le bulletin d'un élève ou lui ajouter des nouvelles évaluations. Détaillons la procédure de création d'un élève. Choisissez l'option 3, après lequel il vous sera demandé de choisir un nom : *d'Oreye*, et un prénom : *Sébastien*. Le système vous confirmera si l'élève à bien été ajouté, ou non. Intérieurement, un nouvelle objet de type `Student` est créé et sauvegardé dans l'établissement.

3 | Description de l'architecture

Passons en revue l'architecture du programme. C'est à dire, les relations entre les différentes classes qui ont été créés et les motivations de nos choix. Un diagramme de classes est disponible en annexe B

Menu

Commençons par la classe `Menu` car elle ne dépend d'aucune autre classe. Quand on s'est lancé dans la création du premier menu, nous nous sommes vite rendu compte que le code serait fort répétitif et difficile à étendre si nous codions les menus manuellement. Nous avons donc inscrit le code au sein d'une classe menu qui gère l'affichage et le choix, valide ou invalide, de l'utilisateur. La classe `Menu` stocke une liste d'options et d'actions correspondantes dans une `List<Tuple<string, Func<bool>>>`. De façon a simplifier d'avantage la création d'un menu, Mathieu David s'est inspiré d'un 'pattern' fort utilisé en RUST qui permet de chaîner les méthodes, comme dans l'exemple sorti de notre code montré à la Figure 3.1.

Pour cela, il suffit simplement que les méthodes renvoient l'objet en question avec `this`. Grâce à cette petite astuce, la création des différents menu est devenue un jeu d'enfant.

```
main_menu
    .set_title("What would you like to do?")
    .add_option("Manage students", student_menu.Run, false)
    .add_option("Manage teachers", teacher_menu.Run, false)
    .add_option("Manage courses", course_menu.Run, false)
    .add_option("Export data", EvaluatorApp.ExportEstablishment)
    .add_option("Exit", EvaluatorApp.Exit, false);
```

FIGURE 3.1 – Création d'un menu avec les différentes options

EvaluatorApp

Toute la logique d'exécution est écrite au sein de cette classe. En temps normal on aurait plutôt utiliser des fonctions libres pour cette partie du projet. Mais comme C# ne permet pas d'en avoir nous avons utilisé une classe statique contenant toutes ces fonctions. Ces méthodes s'occupent surtout de la création des menus et de l'interaction avec l'utilisateur. Tout ce qui touche aux données ne fait qu'appeler des méthodes sur un objet `Establishment`.

Establishment

Est la classe contenant toutes les données. Notamment, une collection d'étudiants, de professeurs et de cours. Ceux-ci sont stockés dans les dictionnaires :

```
private Dictionary<Student, Student> students;
private Dictionary<Teacher,Teacher> teachers;
private Dictionary<string, Course> courses;
```

A première vue, il peut paraître étrange d'avoir des dictionnaires qui ont une clé du même type que leur valeur. Si nous avons déjà l'objet, à quoi bon le chercher dans un dictionnaire ? !

L'explication est simple, la classe `Person`, dont `Student` et `Teacher` héritent, réimplémente les méthodes `public bool Equals(Object obj)` et `public int GetHashCode()` de façon à ce qu'il n'y ait que le prénom et le nom qui soit pris en compte dans la condition d'unicité. Les objet stockés dans les dictionnaires pourraient donc contenir plus d'informations que les objets de même types qui permettent de les retrouver. Ce qui justifie l'utilisation du même type en tant que clé et valeur. Grâce à ceci, il est donc possible de retrouver un élève, caractérisé par un nom, un prénom et une collection d'évaluations, en ne donnant que le nom et prénom de celui-ci. L'utilisation de dictionnaires au lieu de listes évite de devoir parcourir la collection à chaque fois qu'on veut vérifier si un objet est présent ou qu'on veut récupérer les valeurs.

Autres classes

Les autres classes utilisées sont relativement semblables aux classes obtenues après le labo 2 & 3. La classe `Teacher` contient maintenant aussi une liste des cours qu'un professeur donne. La classe `Student` ne possède plus de liste d'évaluations mais un dictionnaire : `private Dictionary<Course, List<Grade>> grades;` qui permet de garder les évaluations séparées pour chaque cours, facilitant la création du bulletin et autres statistiques.

4 | Conclusions

Pour terminer ce rapport, nous vous proposons un feed-back de ce projet.

Nous devons avouer que dans un premier plan ce projet nous semblait plus être une corvée qu'une opportunité, mais ce premier angle de vue a vite changé. Il est ensuite devenu pour l'un d'entre nous une opportunité de se lâcher et de montrer de quoi il était capable en programmation (tout en transmettant son savoir déjà acquis) et pour l'autre une opportunité de comprendre et d'appliquer la matière vue en cours, parfois fort abstraite et difficile à comprendre. Mais il a surtout permit de souder un binôme qui ne se connaissait pas vraiment en entrant en 3BA.

D'un point du vue purement professionnel il a permis à William De Decker de bien comprendre l'importance de l'orienté objet, surtout les avantages que celui-ci offre, et de l'importance de diviser son code en fonction et en classes pour éviter de répéter des bouts de codes quasi identiques encore et encore. Ces notions, bien que déjà vues en 2BA ont ici pu être mieux assimilées et mises en pratique.

Une fois de plus, Git (+ GitHub) s'est montré indispensable pour travailler aisément en équipe. C'est un outil puissant et incontournable au développement informatique de nos jours.

A l'aboutissement de ce projet, nous sommes fière de pouvoir dire que nous avons maîtrisé les bases du C# et de la programmation orienté objet. Nous avons également entraîné notre capacité à assimiler de nouvelles compétences par nous même, ce qui est un atout majeur pour le futur.

A | Menu

A.1 Disposition des menus

1) Manage students
    1) List students
    2) Grade of students
    3) Add student
    4) Remove student
    5) Create evaluation
    6) Back
2) Manage Teachers
    1) List teachers
    2) Add teacher
    3) Remove teacher
    4) Back
3) Manage Courses
    1) List courses
    2) Add course
    3) Remove course
    4) Students signed up for a course [**Non-implémenté**]
    5) Statistics for a course [**Non-implémenté**]
    6) Back
4) Export data
5) Exit

A.2 Description des actions

Dans les explications suivantes, [T] sera utilisé comme paramètre générique pour éviter les descriptions redondantes.

List [**T**] affiche une liste de tous les T qui ont été ajoutés.

Add [**T**] demande à fournir quelques informations nécessaires à la création de T et crée l'objet T si il n'existe pas déjà.

Remove [**T**] demande à fournir quelques informations nécessaires pour retrouver T et supprime T si il existe.

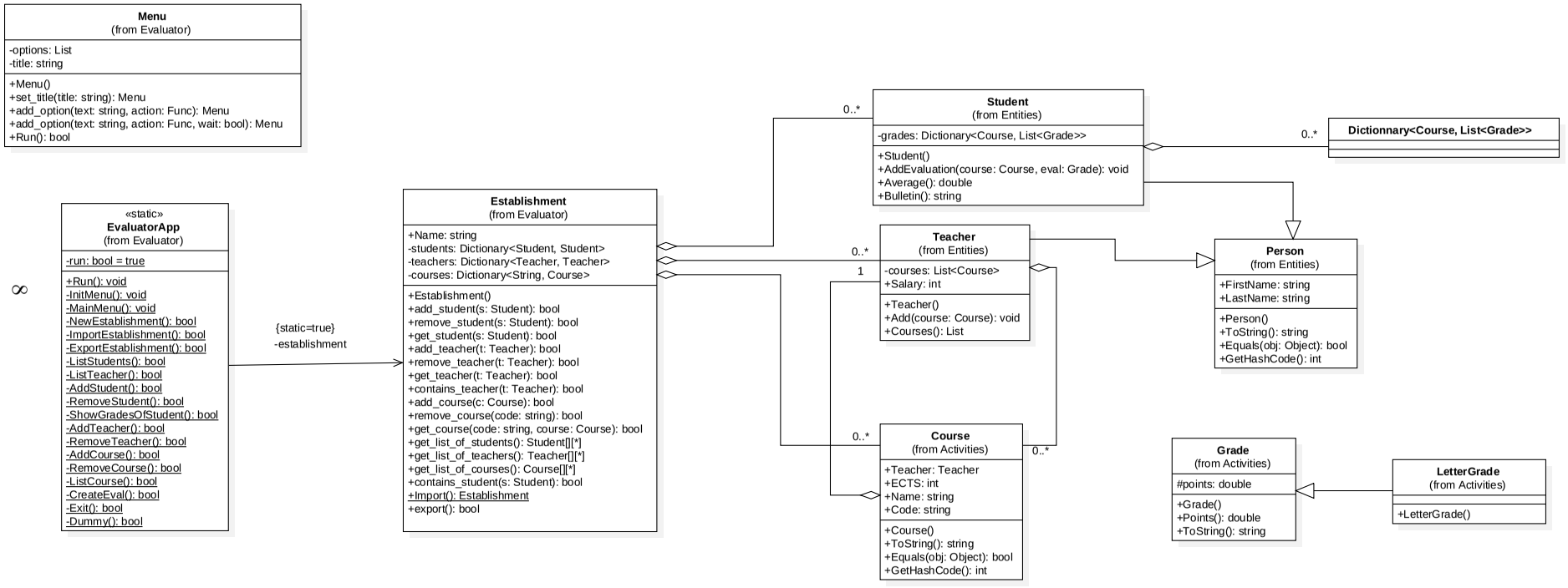
Grade of student demande un nom et prénom et affiche le bulletin de cet étudiant si il existe.

Create evaluation demande un code du cours, nom et prénom de l'étudiant et une cote. Une cote peut être entrée soit au format américain ("A+", "B-", "F", ...) soit en tant que nombre à virgule. Dans le deuxième cas il vous sera demandé de préciser la cote maximale obtenable. Ceci permet d'encoder des cotes sur n'importe quelle valeur. Cette dernière sera automatiquement ramenée sur 100 afin de faciliter la lecture et la création d'un bulletin ou le calcul d'une moyenne.

Students signed up for a course n'a pas été implémenté par manque de temps de notre part. Nous avons en effet privilégié la qualité de code ainsi que la robuste de celui-ci et n'avons ainsi pas eu le temps de terminer toutes les fonctionnalités que nous désirions implémenter...

Statistics for a course n'a pas été implémenté pour les mêmes raisons.

B | Diagramme de classes


Menu
(from Evaluator)
-options: List
-title: string
+Menu()
+set_title(title: string): Menu
+add_option(text: string, action: Func): Menu
+add_option(text: string, action: Func, wait: bool): Menu
+Run(): bool
«static»
EvaluatorApp
(from Evaluator)
-run: bool = true
+Run(): void
-InitMenu(): void
-MainMenu(): void
-NewEstablishment(): bool
-ImportEstablishment(): bool
-ExportEstablishment(): bool
-ListStudents(): bool
-ListTeacher(): bool
-AddStudent(): bool
-RemoveStudent(): bool
-ShowGradesOfStudent(): bool
-AddTeacher(): bool
-RemoveTeacher(): bool
-AddCourse(): bool
-RemoveCourse(): bool
-ListCourse(): bool
-CreateEval(): bool
-Exit(): bool
-Dummy(): bool
∞
{static=true}
-establishment
Establishment
(from Evaluator)
+Name: string
-students: Dictionary<Student, Student>
-teachers: Dictionary<Teacher, Teacher>
-courses: Dictionary<String, Course>
+Establishment()
+add_student(s: Student): bool
+remove_student(s: Student): bool
+get_student(s: Student): bool
+add_teacher(t: Teacher): bool
+remove_teacher(t: Teacher): bool
+get_teacher(t: Teacher): bool
+contains_teacher(t: Teacher): bool
+add_course(c: Course): bool
+remove_course(code: string): bool
+get_course(code: string, course: Course): bool
+get_list_of_students(): Student[][*]
+get_list_of_teachers(): Teacher[][*]
+get_list_of_courses(): Course[][*]
+contains_student(s: Student): bool
+Import(): Establishment
+export(): bool
0..*
Student
(from Entities)
-grades: Dictionnary<Course, List<Grade>>
+Student()
+AddEvaluation(course: Course, eval: Grade): void
+Average(): double
+Bulletin(): string
0..*
Dictionnary<Course, List<Grade>>
0..*
1
Teacher
(from Entities)
-courses: List<Course>
+Salary: int
+Teacher()
+Add(course: Course): void
+Courses(): List
Person
(from Entities)
+FirstName: string
+LastName: string
+Person()
+ToString(): string
+Equals(obj: Object): bool
+GetHashCode(): int
0..*
0..*
Course
(from Activities)
+Teacher: Teacher
+ECTS: int
+Name: string
+Code: string
+Course()
+ToString(): string
+Equals(obj: Object): bool
+GetHashCode(): int
Grade
(from Activities)
#points: double
+Grade()
+Points(): double
+ToString(): string
LetterGrade
(from Activities)
+LetterGrade()