

# CAS 703 Term Project

## The Graduation Requirements Course Selection DSL

Lyuyang Wang<sup>1</sup>, Yaoxu Wang<sup>2</sup>

<sup>1,2</sup> Computing and Software, McMaster University

### 1. Introduction

The Graduation Requirements Course Selection DSL automates personalized course planning for high school students. Students can specify their grade level, academic interests, and desired degree program for college study to generate a list of recommended courses that meet their school's graduation requirements and university entry requirement. The DSL includes a simple syntax with custom functions and rules to calculate course credits and requirements and provide constraints like minimum credit hours and required courses. The DSL also generates visualizations to track progress and provide tools for exploring academic interests and discovering new courses.

### 2. Metamodel for the Course Selection Domain-Specific Language

#### 2.1. Metamodel Overview

The metamodel for the Course Selection domain-specific language consists of the following classes, as the class diagram of the metamodel please refer to Figure 1.

- **Course**: Represents a course offering in the system, with attributes such as `courseID`, `name`, `description`, and `creditHours`. It has relationships with other `Course` instances (`prerequisites` and `parentCourse`), `Teacher`, and `Department`.
- **CourseCategory**: An enumeration representing different course categories, such as Math, Science, Humanities, Social Studies, Arts, Technology, Language, and Physical Education.
- **Student**: Represents a student in the system, with attributes such as `name`, `gradeLevel`, and `academicInterests`. It has relationships with `DegreeProgram` (`desiredDegreeProgram`) and `Course` (`selectedCourses`).
- **DegreeProgram**: Represents a degree program, with attributes such as `name` and `description`. It has a containment relationship with `GraduationRequirement` instances (`graduationRequirements`).
- **GraduationRequirement**: Represents the graduation requirements, with attributes such as `name`, `minimumCreditHours`, and `requiredCourseCategories`. It has a relationship with `Course` (`requiredCourses`).
- **Teacher**: Represents a teacher, with attributes such as `name`. It has a relationship with `Course` (`courses`).
- **Department**: Represents a department, with attributes such as `name`. It has containment relationships with `Course` (`courses`) and `Teacher` (`teachers`).
- **CourseSelection**: Acts as a container for the entire model, holding instances of `Student`, `DegreeProgram`, `Course`, `Department`, `Teacher`, and `GraduationRequirement`.

**2.2. Assumptions**

- A course can have multiple prerequisites and can be a prerequisite for multiple other courses.
- Graduation requirements include a minimum number of credit hours and can have multiple required courses and course categories.
- A teacher can teach multiple courses, and a course can have only one teacher.
- A department can have multiple courses and teachers.

**2.3. Alternative Design Decisions**

- Using a containment relationship between Course and Teacher: Instead of a simple reference relationship between Course and Teacher, a containment relationship could be used. However, this would imply that a course cannot be shared among teachers, and a teacher would not exist independently of the courses they teach. The current design allows for more flexibility in assigning teachers to courses.
- Including a "CourseLevel" attribute in the Course class: This attribute could represent different course levels (e.g., beginner, intermediate, advanced). However, this information could also be included in the "description" attribute, and adding another attribute might unnecessarily increase the complexity of the metamodel.

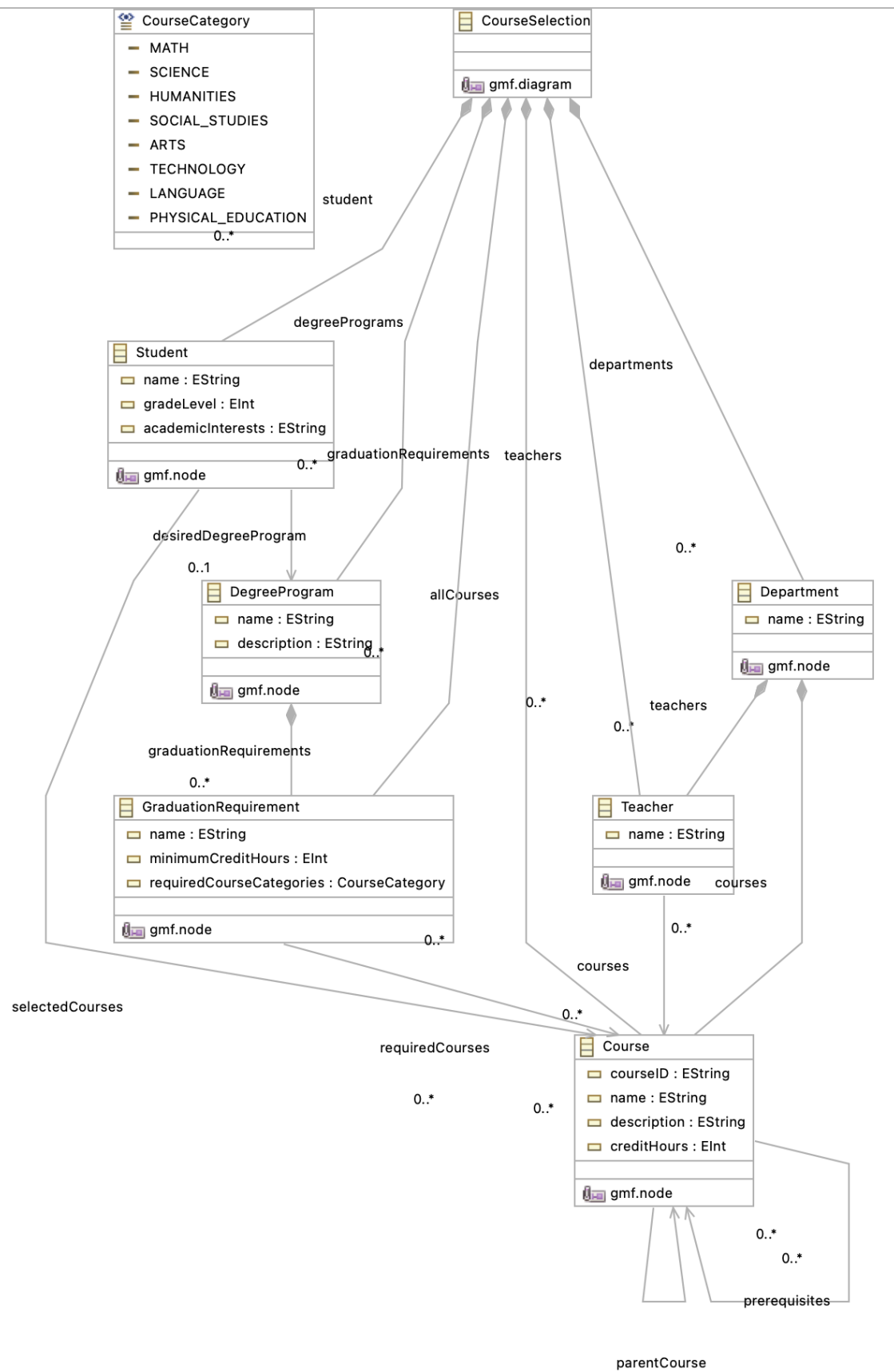


Figure 1. class diagram of the metamode

### 3. Concrete Syntax and Supporting Editor for the Course Selection Language

We have used Epsilon's Eugenia, a GMF-based tool, to define the concrete syntax and implement a supporting graphical editor for the Course Selection domain-specific language (DSL). This section of the report will provide a screenshot of the editor, discuss and justify the syntax design and editor implementation decisions, and reflect on the strengths and weaknesses of the selected concrete syntax compared to alternatives. The editor is shown in Figure 2.

#### 3.1. Concrete Syntax Design

Using the GMF annotations in the provided EMF model, we have designed a graphical concrete syntax for the Course Selection DSL. The design decisions and their justifications are as follows:

- **Nodes:** We have used the @gmf.node annotation to represent classes as graphical nodes. The label attribute is set to "name" for each class, indicating that the name attribute should be displayed as the label of the node.
- **Links:** We have used the @gmf.link annotation to represent relationships between classes as graphical connections. For example, the relationship between Course and Department is represented by an arrow with the label "belongTo".
- **Enumerations:** The CourseCategory enumeration is represented using the @gmf.label annotation, which specifies the text displayed for each value in the enumeration.
- **Compartments:** We have used the @gmf.compartment annotation to group related elements within a parent node. For instance, the Department class contains compartments for courses and teachers.

#### 3.2. Editor Implementation

In the Course Selection DSL, the Department class includes two Course instances and two Teacher instances. The DegreeProgram class contains one GraduationRequirement instance. There is a Student class that connects the DegreeProgram and Course classes. One of the courses has a prerequisite, which is the other course. The two Teacher instances each point to one of the two courses, and the GraduationRequirement instance points to one of the courses.

To implement the supporting editor for the Course Selection DSL, we have used Epsilon's Eugenia tool. The editor has been designed to reflect the relationships and instances described in the domain. The following are the key features of the implemented editor:

- **Departments:** The Department class is represented as a node containing two compartments, one for Course instances and another for Teacher instances. In the editor, users can create two Course instances and two Teacher instances within each Department node.
- **Degree Programs:** The DegreeProgram class is represented as a node containing a compartment for GraduationRequirement instances. Users can create one GraduationRequirement instance within each DegreeProgram node.
- **Students:** The Student class is represented as a node connecting to the DegreeProgram and Course classes through links. This allows users to visually associate a student with their desired degree program and selected courses.
- **Course Prerequisites:** Courses can have prerequisites, represented by a link between two Course nodes. In the editor, users can create a link from one course to another to indicate that the first course is a prerequisite for the second course.
- **Teachers and Courses:** The Teacher class is represented as a node connected to the Course class through links. Users can create links between Teacher and Course nodes, indicating that a teacher is responsible for teaching a specific course.

- **Graduation Requirements:** The GraduationRequirement class is represented as a node connected to the Course class through links. Users can create links between GraduationRequirement and Course nodes, specifying which courses are required to fulfill the graduation requirements.

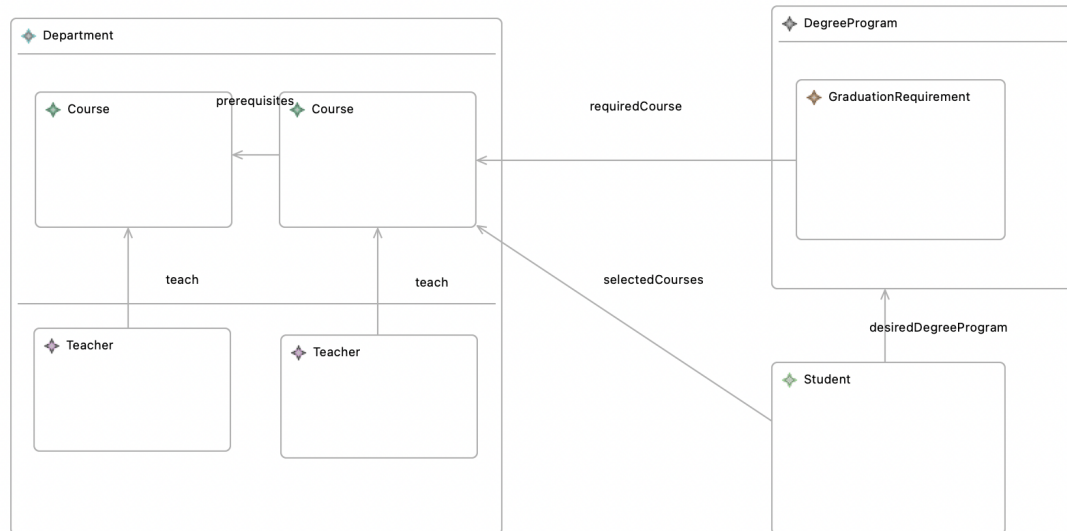


Figure 2. Eugenia Editor

### 3.3. Strengths and Weaknesses

#### 3.3.1. Strengths

- **Visual representation:** Eugenia enables the creation of a graphical editor that provides a visually intuitive representation of the domain. This makes it easy for users to understand the relationships between courses, teachers, departments, and students.
- **Customizability:** The GMF-based editor generated by Eugenia can be customized to suit the specific requirements of your DSL. This allows for tailored user experiences and support for advanced features such as validation, undo/redo, and copy/paste.
- **Ease of maintenance:** Eugenia simplifies the process of updating the syntax and regenerating the editor as the metamodel evolves. Changes to the EMF model can be easily reflected in the graphical editor by updating the GMF annotations and regenerating the editor using Eugenia.

#### 3.3.2. Weaknesses

- **Complexity:** Creating a GMF-based editor using Eugenia requires a steep learning curve, especially for users unfamiliar with Eclipse and EMF. The development process involves multiple steps and requires understanding of both the EMF and GMF frameworks.
- **Scalability:** The graphical syntax might become cluttered and difficult to navigate for large, complex models. As the number of elements and relationships in the domain increases, the readability of the graphical editor may suffer.

### 3.4. Alternatives

- **Textual DSL:** Instead of using a graphical concrete syntax, you could define a textual concrete syntax for your language. This can be achieved using tools like Xtext or ANTLR.

Textual syntaxes can be more compact and easier to work with for large models. However, they may not be as visually intuitive as graphical syntaxes, especially for users who are not familiar with the domain.

- **Tree-based editor:** Another alternative is to use a tree-based editor, which provides a hierarchical representation of the model elements. This can be achieved using the EMF tree editor or a custom implementation. Tree-based editors can provide a more scalable representation of the domain but may lack the visual intuitiveness of a graphical editor.

#### 4. Epsilon Validation Language

various validation constraints are implemented for different elements in our domain model, such as students, courses, graduation requirements, departments, degree programs, and teachers. These constraints help ensure that the model adheres to certain rules that cannot be expressed directly in the metamodel. Here is a brief explanation of each constraint:

- **ValidName:** This constraint is defined for Student, Department, DegreeProgram, and Teacher. It checks if the name attribute is defined and not empty. A fix is provided to set the name to 'Unknown' if it's empty.
- **ValidGradeLevel:** This constraint is defined for Student. It checks if the gradeLevel attribute is between 1 and 12.
- **TooManyPrerequisites:** This constraint is defined for Course. It checks if the number of prerequisites for a course is more than 5. A fix is provided to remove extra prerequisites if there are too many.
- **ValidCreditHours:** In the Student context, this constraint ensures that the total credit hours of the selected courses for a student are equal to or greater than the minimum credit hours required for each GraduationRequirement. In the Course context, it checks if the course credit hours are within the range of 1 to 6. If any of these conditions aren't met, an appropriate message is displayed.
- **PrerequisitesCompleted:** In the Student context, this constraint checks if all prerequisites for the selected courses are included in the student's completedCourses list. If not, a message is displayed.
- **PrerequisitesNotIncludeSelf:** This constraint is defined for Course. It checks if a course is not a prerequisite of itself.
- **ValidRequirements:** This constraint is defined for GraduationRequirement. It checks if a GraduationRequirement has at least one required course or one required course category.
- **AtLeastOneTeacher:** This constraint is defined for Department. It checks if a department has at least one teacher.
- **AtLeastOneGraduationRequirement:** This constraint is defined for DegreeProgram. It checks if a DegreeProgram has at least one GraduationRequirement.
- **BelongsToDepartment:** This constraint is defined for Teacher. It checks if a teacher belongs to a department.

Next, we are going to integrate the implemented constraints with the editor discussed in section 4 to provide in-editor errors/warnings. The overview of metamodel as shown on Figure 3.

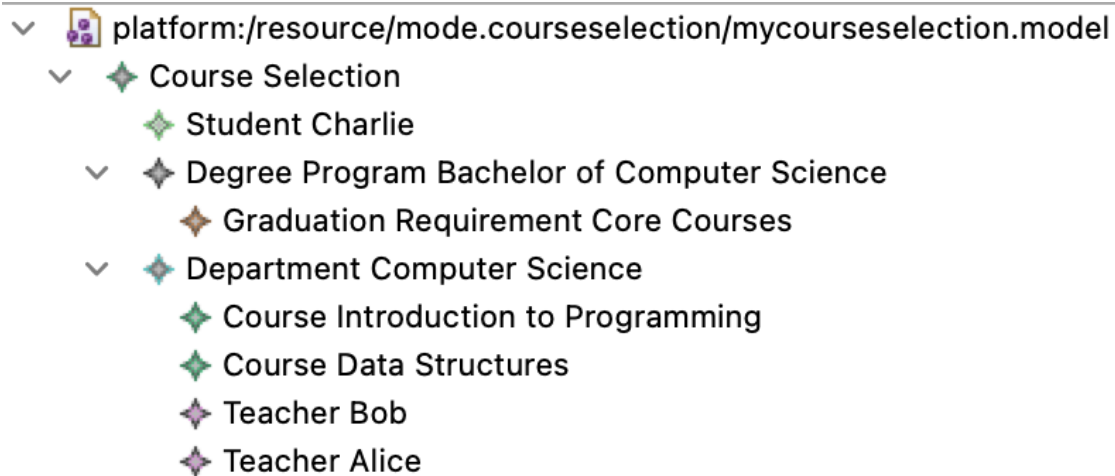


Figure 3. Model Overview

- **Course.PrerequisitesNotIncludeSelf:** To violate the PrerequisitesNotIncludeSelf constraint, we designate the "Data Structure" course as its own prerequisite in our .model file. Please see Figure 4.

| Tasks EPackage Registry Properties Console Validation |                             |
|---|-----------------------------|
| Property  | Value                       |
| Category  | MATH                        |
| Course ID   | CS201                       |
| Credit Hours  | 7                           |
| Department  | Department Computer Science |
| Description   |                             |
| Name  | Data Structures             |
| Parent Course   | Course Data Structures      |
| Prerequisites   | Course Data Structures      |
| Teacher   | Teacher Bob                 |

Figure 4. Set prerequisites of Course Class

- **Course.ValidCreditHours** To violate the ValidCreditHours constraint, we assign a value of 7 credit hours to the "Data Structure" course, which should instead range from 1 to 6. Please refer to Figure 5.

| Tasks EPackage Registry Properties Console Validation |                             |
|---|-----------------------------|
| Property  | Value                       |
| Category  | MATH                        |
| Course ID   | CS201                       |
| Credit Hours  | 7                           |
| Department  | Department Computer Science |
| Description   |                             |
| Name  | Data Structures             |
| Parent Course   | Course Data Structures      |
| Prerequisites   | Course Data Structures      |
| Teacher   | Teacher Bob                 |

Figure 5. Set Credit Hours of Course Class

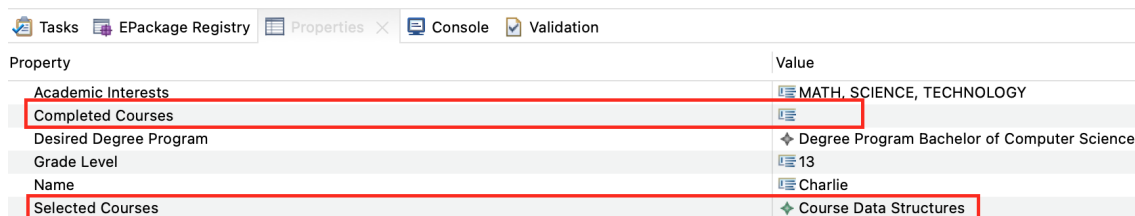
- **Student.ValidGradeLevel** To violate the ValidGradeLevel constraint, we assign a Grade Level of 13 to the student "Charlie", which should instead fall within the range of 1 to 12. Please refer to Figure 6.



| Property               | Value                                       |
|------------------------|---|
| Academic Interests     | MATH, SCIENCE, TECHNOLOGY                   |
| Completed Courses      |   |
| Desired Degree Program | Degree Program Bachelor of Computer Science |
| Grade Level            | 13  |
| Name                   | Charlie                                     |
| Selected Courses       | Course Data Structures                      |

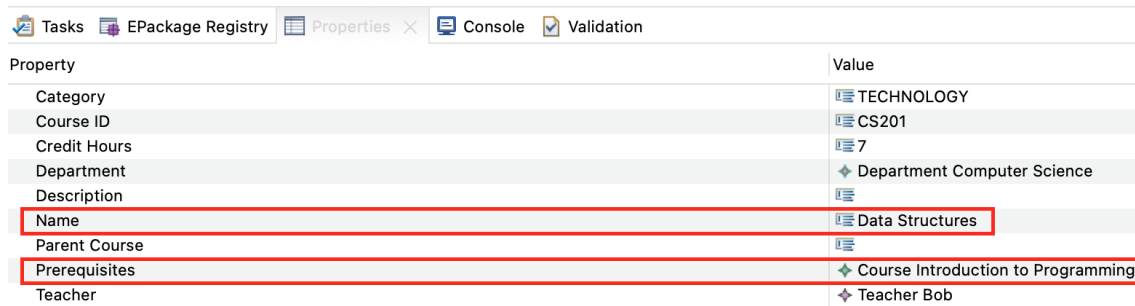
Figure 6. Set Grade Level of Student Class

- **Student.PrerequisitesCompleted** To violate the PrerequisitesCompleted constraint, we set the prerequisite of the "Data Structure" course to the "Introduction to Programming" course, while the list of completed courses for student "Charlie" does not include the "Introduction to Programming" course. Please refer to Figure 7 and Figure 8.



| Property               | Value                                       |
|------------------------|---|
| Academic Interests     | MATH, SCIENCE, TECHNOLOGY                   |
| Completed Courses      |   |
| Desired Degree Program | Degree Program Bachelor of Computer Science |
| Grade Level            | 13  |
| Name                   | Charlie                                     |
| Selected Courses       | Course Data Structures                      |

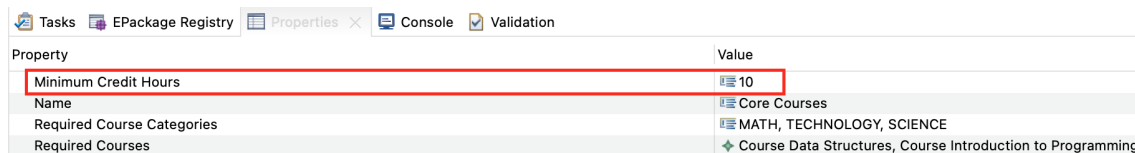
Figure 7. Set Completed Courses of Student Class to null



| Property      | Value                              |
|---------------|------------------------------------|
| Category      | TECHNOLOGY                         |
| Course ID     | CS201                              |
| Credit Hours  | 7                                  |
| Department    | Department Computer Science        |
| Description   |                                    |
| Name          | Data Structures                    |
| Parent Course |                                    |
| Prerequisites | Course Introduction to Programming |
| Teacher       | Teacher Bob                        |

Figure 8. Set Prerequisites of "Data Structure" Course to "Introduction of Programming" Course

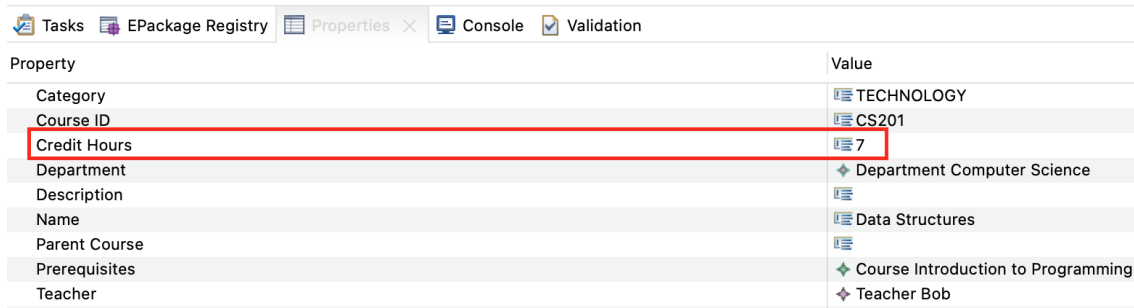
- **Student.ValidCreditHours** To violate the ValidCreditHours constraint, we assign a value of 7 credit hours to the "Data Structure" course and set the Minimum Credit Hours for "Graduation Requirement" to 10. Consequently, the student does not fulfill the requirements necessary for graduation. Please refer to Figures 9 and 10.



| Property                   | Value  |
|----------------------------|--|
| Minimum Credit Hours       | 10   |
| Name                       | Core Courses   |
| Required Course Categories | MATH, TECHNOLOGY, SCIENCE                                  |
| Required Courses           | Course Data Structures, Course Introduction to Programming |

Figure 9. Set Graduation Requirement Class Minimum Credit Hours to 10

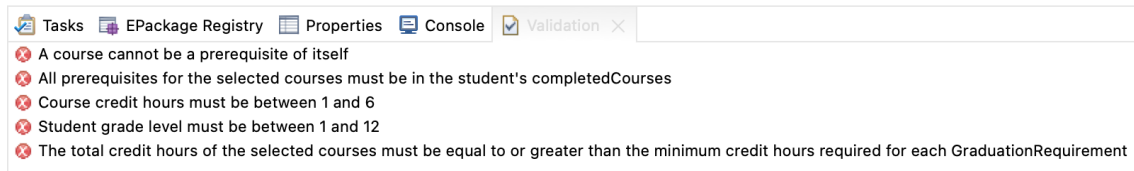




| Property      | Value                              |
|---------------|------------------------------------|
| Category      | TECHNOLOGY                         |
| Course ID     | CS201                              |
| Credit Hours  | 7                                  |
| Department    | Department Computer Science        |
| Description   |                                    |
| Name          | Data Structures                    |
| Parent Course |                                    |
| Prerequisites | Course Introduction to Programming |
| Teacher       | Teacher Bob                        |

Figure 10. Set student select course's credit to 7

The result of editor errors/warnings is shown below:



| Task | EPackage Registry | Properties | Console | Validation   |
|------|-------------------|------------|---------|--|
| ✖    |                   |            |         | A course cannot be a prerequisite of itself  |
| ✖    |                   |            |         | All prerequisites for the selected courses must be in the student's completedCourses   |
| ✖    |                   |            |         | Course credit hours must be between 1 and 6  |
| ✖    |                   |            |         | Student grade level must be between 1 and 12   |
| ✖    |                   |            |         | The total credit hours of the selected courses must be equal to or greater than the minimum credit hours required for each GraduationRequirement |

Figure 11. Editor's errors

## 5. Model Operation for DSL

The model management operation involves the transformation of an EMF model, representing course selection information, into an HTML file for visualization. This process requires the use of Epsilon's EGL and EGX languages to define templates and rules to generate the desired output. The main steps of the operation are as follows:

- Create an EGL template (courseSelection2HTML.egl) that provides an HTML structure for displaying student information from the EMF model. This template includes a table layout and styles to ensure proper formatting.
- Define an EGX file (generateHTML.egx) containing a rule that specifies the transformation of the source EMF model using the EGL template. The rule also sets the target output file (in this case, courseSelection.html).
- Execute the EGX transformation using Epsilon's configuration.

Challenges and design choices:

- The EGL template is designed specifically to display student information. To include other model elements such as DegreeProgram, Course, Department, Teacher, and GraduationRequirement, additional tables and formatting may be needed. The template can be extended as required, but keeping the HTML structure and layout clean and manageable may prove challenging as more data is added.
- In the EGL template, we used a simple table layout with basic CSS styles to make the output more readable. However, more advanced styling or layout techniques could be employed to improve the visualization.
- Executing the transformation requires the correct configuration of file paths and dependencies. Ensuring that the Epsilon JAR, EGX file, and EMF model are all correctly referenced is essential to the successful execution of the model management operation.
- The current approach relies on a single EGL template and EGX rule, which may not be the most efficient or scalable solution for complex models. In such cases, using multiple templates or rules could be considered to better manage and modularize the transformation process.

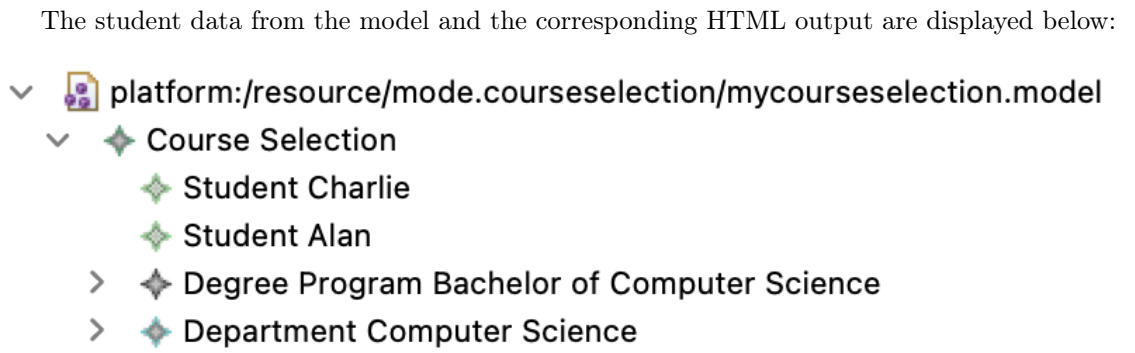


Figure 12. The model of student data

Course Selection

Students

| Name    | Grade Level | Academic Interests          | Desired Degree Program       | Selected Courses | Completed Courses           |
|---------|-------------|-----------------------------|------------------------------|------------------|-----------------------------|
| Charlie | 13          | MATH , SCIENCE , TECHNOLOGY | Bachelor of Computer Science | Data Structures  |                             |
| Alan    | 12          | ART , MATH                  | Bachelor of Computer Science | Data Structures  | Introduction to Programming |

Figure 13. HTML result