**C# OOP Retake Exam**

**University Competition**

* **Overview**

The end of the last year in high school is approaching and the students will have to choose wisely their way to the future. Everyone will try to enter the best universities. You will be in the department, developing the governmental centralized ranking system. This system must have support for **Subject**, **Student** and **University**. The project will consist of **model classes** and a **controller class**, which manages the **interaction** between the **subjects, students** and **universities**.

* **Setup**
* Upload **only the UniversityCompetition** project in every task **except** **Unit Tests.**
* **Do not modify the interfaces or their packages.**
* Use **strong cohesion** and **loose coupling.**
* **Use inheritance and the provided interfaces wherever possible.**
* This includes **constructors**, **method parameters,** and **return types.**
* **Do not** violate your **interface** **implementations** by adding **more public methods** in the concrete class than the interface has defined.
* Make sure you have **no public fields** anywhere.
* **Exception messages** and **output messages** can be found in the **"Utilities"** folder.
* For solving this problem use **Visual Studio 2019, Visual Studio 2022** and **netcoreapp 3.1**
* **Task 1: Structure (50 points)**

**For this task’s evaluation logic in the methods isn’t included.**

You are given **4** interfaces, and you must implement their functionality in the **correct classes**.

There are 3types of entities and 3 repositories in the application: **Subject, Student, University and Repository (SubjectRepository, StudentRepository and UniversityRepository) for each of them**:

**Subject**

**Subject** is a **base class** for any **type of Subject,** and it **should not be able to be instantiated**.

**Data**

* **Id** – **int**
* The **Id** property will take its value upon adding the entity to the **SubjectRepository**. Every new **Subject** will take the next consecutive number in the repository, starting from 1. The property will be set in the **AddSubject method** from the **Controller class**.
* **Name** - **string**
* If the name **is null or whitespace,** throw an **ArgumentException** with the message "Name cannot be null or whitespace!"
* **Rate – double**
* The significance of the **Subject**.

**Constructor**

A **Subject** should take the following values upon initialization:

**int subjectId, string subjectName, double subjectRate**

**Child Classes**

There are three concrete types of **Subject**:

**TechnicalSubject**

**TechnicalSubject has a** constant value for **subjectRate = 1.3**

The constructor of the **TechnicalSubject** should take the following parameters upon initialization:

**int subjectId, string subjectName**

**EconomicalSubject**

**EconomicalSubject has a** constant value for **subjectRate = 1.0**

The constructor of the **EconomicalSubject** should take the following parameters upon initialization:

**int subjectId, string subjectName**

**HumanitySubject**

**HumanitySubject has a** constant value for **subjectRate = 1.15**

The constructor of the **HumanitySubject** should take the following parameters upon initialization:

**int subjectId, string subjectName**

**Student**

**Data**

* **Id** – **int**
* The **Id** property will take its value upon adding the entity to the **StudentRepository**. Every new **Student** will take the next consecutive number in the repository, starting from 1. The property will be set in the **AddStudent method** from the **Controller class**.
* **FirstName** - **string**
* If the name **is null or whitespace,** throw an **ArgumentException** with the message "Name cannot be null or whitespace!"
* **LastName** - **string**
* If the name **is null or whitespace,** throw an **ArgumentException** with the message "Name cannot be null or whitespace!"
* **CoveredExams** – **IreadOnlyCollection<int> -** A collection of integer values, representing the subject ids of all covered exams by the student.
* **University** – **IUniversity –** The University where the student managed to join, after covering all the required exams.

**Behavior**

**void CoverExam(ISubject subject)**

Takes the subject’s id and adds it to the collection of **CoveredExams**

**void JoinUniversity(IUniversity university)**

This method sets value of the property **University.**

**Constructor**

A **Student** should take the following values upon initialization:

**int studentId, string firstName, string lastName**

**University**

**Data**

* **Id** – **int**
* The **Id** property will take its value upon adding the entity to the **UniversityRepository**. Every new **University** will take the next consecutive number in the repository, starting from 1. The property will be set in the **AddUniversity method** from the **Controller class**.
* **Name** - **string**
* If the name **is null or whitespace,** throw an **ArgumentException** with the message "Name cannot be null or whitespace!"
* **Category – string** available categories are: **Technical, Economical, Humanity**
* If the value does not match the allowed categories **(case-sensitive),** throw an **ArgumentException** with the message "University category **{value}** is not allowed in the application!"
* **Capacity – int** – the maximum possible number of students admitted to the university.
* If the value is less than zero throw **ArgumentException** with the message: "University capacity cannot be a negative value!"
* **RequiredSubjects – IReadOnlyCollection<int>** - A collection of integer values, representing the subject ids of all the subjects on which the student has to have the exams covered.

**Constructor**

A **University** should take the following values upon initialization:

**int universityId, string universityName, string category, int capacity,**

**ICollection<int> requiredSubjects**

**SubjectRepository**

The repository holds information about the subjects.

**Data**

* **Models** – **IReadOnlyCollection<ISubject>**

**Behavior**

**void AddModel(ISubject subject)**

Adds a **Subject** in the collection.

**ISubject FindById(int id)**

Returns a **Subject** with the given id, if it exists in the repository, otherwise returns null.

**ISubject FindByName(string name)**

Returns a **Subject** with the given name, if it exists in the repository, otherwise returns null.

**StudentRepository**

The repository holds information about the students.

**Data**

* **Models** – **IReadOnlyCollection<IStudent>**

**Behavior**

**void AddModel(IStudent student)**

Adds a **Student** in the collection.

**IStudent FindById(int id)**

Returns a **Student** with the given id, if it exists in the repository, otherwise returns null.

**IStudent FindByName(string name)**

Returns a **Student** with the given name ***(Split the given string by single space and check both the first and the last name of the student)***, if it exists in the repository, otherwise returns null.

**UniversityRepository**

The repository holds information about the universities.

**Data**

* **Models** – **IReadOnlyCollection<IUniversity>**

**Behavior**

**void AddModel(IUniversity university)**

Adds a **University** in the collection.

**IUniversity FindById(int id)**

Returns a **University** with the given id, if it exists in the repository, otherwise returns null.

**IUniversity FindByName(string name)**

Returns a **University** with the given name, if it exists in the repository, otherwise returns null.

* **Task 2: Business Logic (150 points)**

**The Controller Class**

The business logic of the program should be concentrated around several **commands**. You that you musts, which you have to implement in the correct classes.

The first interface is **IController**. You must create a **Controller** class, which implements the interface and implements all of its methods. The constructor of **Controller** does not take any arguments. The given methods should have the logic described for each in the Commands section. When you create the **Controller** class, go into the **Engine** class constructor and uncomment the "this.controller = new Controller();" line.

**Data**

You need to keep track of some things, this is why you need some private fields in your controller class:

* **subjects – SubjectRepository**
* **students – StudentRepository**
* **universities - UniversityRepository**

**Commands**

There are several commands, which control the business logic of the application. They are stated below.

**AddSubject Command**

**Parameters**

* **subjectName – string**
* **subjectType - string**

**Functionality**

The method should create and add a new entity of **ISubject** to the **SubjectRepository.**

* If the given subjectType is not supported in the application, return the following message: **"**Subject type **{subjectType}** is not available in the application!**"**
* If there is already added a **Subject** with the given name, return the following message: **"{subjectName}** is already added in the repository.**"**
* If none of the above cases is reached, create a new **Subject** from the appropriate type and add it to the **SubjectRepository**. Return the following message: **"{subjectType}** **{subjectName}** is created and added to the **{relevantRepositoryTypeName}**!**"**

**AddUniversity Command**

**Parameters**

* **universityName – string**
* **category** – **string**
* **capacity** – **int**
* **requiredSubjects – List<string>**

**Functionality**

The method should create and add a new entity of **IUniversity** to the **UniversityRepository.**

* If there is already added a **University** with the given name, return the following message: **"{universityName}** is already added in the repository.**"**
* If the above case is not reached, convert the given collection of requiredSubjects into **collection of integers**, containing every required **subject’s id**. The subjects will be already added into the **SubjectRepository**. Create a new **University** and add it to the **UniversityRepository**. Return the following message: **"{universityName}** university is created and added to the **{relevantRepositoryTypeName}**!**"**

**AddStudent Command**

**Parameters**

* **firstName – string**
* **lastName** – **string**

**Functionality**

The method should create and add a new entity of **IStudent** to the **StudentRepository.**

* If there is already added a **Student** with the given firstName and lastName, return the following message: **"{firstName}** **{lastName}** is already added in the repository.**"**
* If the above case is not reached, create a new **Student** and add it to the **StudentRepository**. Return the following message: **"**Student **{firstName}** **{lastName}** is added to the **{relevantRepositoryTypeName}**!**"**

**TakeExam Command**

**Parameters**

* **studentId – int**
* **subjectId - int**

**Functionality**

The method should add the given subjectId to the collection **CoveredExams** of the **Student** with the given studentId.

* If a Student with the given **studentId** doesn’t exist in the **StudentRepository**, return the following message: **"**Invalid student ID!**"**
* If a Subject with the given **subjectId** doesn’t exist in the **SubjectRepository**, return the following message: **"**Invalid subject ID!**"**
* If the **Student** with the given **studentId** has already covered the exam (check in the **CoveredExam** collection of the **Student**) on the **Subject** with the given **subjectId**, return the following message: **"{studentFirstName}** **{studentLastName}** has already covered exam of **{subjectName}**.**"**

If none of the above cases is reached, add the given subjectId to the collection **CoveredExams** of the **Student** with the given studentId. Return the following message: **"{studentFirstName}** **{studentLastName}** covered **{subjectName}** exam!**"**

**ApplyToUniversity Command**

**Parameters**

* **studentName – string**
* **universityName – string**

**Functionality**

The method should set the **University** property of the **Student** with the given studentName ***(Split the given string by whitespace and check both the first and the last name of the student),*** to the value of the **University** with the given universityName.

* If a **Student** with the given studentName doesn’t exist in the **StudentRepository**, return the following message: **"{studentFirstName} {studentLastName}** is not registered in the application!**"**
* If a **University** with the given universityName doesn’t exist in the **UniversityRepository**, return the following message: **"{universityName}** is not registered in the application!**"**
* If the **Student** with the given studentName **has not covered all the required exams** for the **University** with the given name, return the following message: **"{studentName}** has not covered all the required exams for **{universityName}** university!**"**
* If the **Student** with the given studentName has already joined the **University** with the given universityName, return the following message: **"{studentFirstName}** **{studentLastName}** has already joined **{UniversityName}**.**"**

If none of the above cases is reached, set the **University** property of the **Student** with the given studentName, to the value of the **University** with the given universityName. Return the following message: **"{studentFirstName}** **{studentLastName}** joined **{universityName}** university!**"**

**UniversityReport Command**

**Parameters**

* **universityId – int**

**Functionality**

* Find the **University** with the given **universityId**.
* Returns the following string report:

**"**\*\*\* **{universityName}** \*\*\*

Profile: **{universityCategory}**

Students admitted: **{studentsCount}**

University vacancy: **{capacityLeft}"**

**Note: studentsCount => the count of all students admitted in the given university**

**Note: capacityLeft => the university capacity – the count of all admitted students in the university**

**Note: Do not use** "\r\n" **for a new line.**

**Input / Output**

You are provided with one interface, which will help with the correct execution process of your program. The interface is **Engine** and the class implementing this interface should read the input and when the program finishes, this class should print the output.

**Input**

Below, you can see the **format** in which **each command** will be given in the input:

* **AddSubject {subjectName} {subjectType}**
* **AddUniversity {universityName} {category} {capacity} {requiredSubjects}**
* **AddStudent {firstName} {lastName}**
* **TakeExam {studentId} {subjectId}**
* **ApplyToUniversity {studentName} {universityName}**
* **UniversityReport {universityId}**
* **CompetitionReport**
* **END**

**Output**

Print the output from each command when issued. If an exception is thrown during any of the commands' execution, print the exception message.

**Examples**

|  |
| --- |
| **Input** |
| **AddSubject Mathematics TechnicalSubject**  **AddSubject Physics TechnicalSubject**  **AddSubject Geography EconomicalSubject**  **AddSubject Biology HumanitySubject**  **AddSubject Chemistry TechnicalSubject**  **AddSubject Literature HumanitySubject**  **AddSubject History EconomicalSubject**  **AddSubject ComputerProgramming TechnicalSubject**  **AddUniversity GreatEinstein Technical 50 Physics,Mathematics,ComputerProgramming**  **AddUniversity IbnSena Humanity 30 Biology,Physics,Chemistry**  **AddStudent Alice Pitt**  **AddStudent Boris Grey**  **AddStudent Rob Butterscotch**  **TakeExam 1 4**  **TakeExam 1 2**  **TakeExam 1 5**  **TakeExam 3 2**  **TakeExam 3 1**  **ApplyToUniversity Alice Pitt GreatEinstein**  **ApplyToUniversity Alice Pitt IbnSena**  **ApplyToUniversity Rob Butterscotch GreatEinstein**  **TakeExam 3 8**  **ApplyToUniversity Rob Butterscotch GreatEinstein**  **UniversityReport 1**  **Exit** |
| **Output** |
| **TechnicalSubject Mathematics is created and added to the SubjectRepository!**  **TechnicalSubject Physics is created and added to the SubjectRepository!**  **EconomicalSubject Geography is created and added to the SubjectRepository!**  **HumanitySubject Biology is created and added to the SubjectRepository!**  **TechnicalSubject Chemistry is created and added to the SubjectRepository!**  **HumanitySubject Literature is created and added to the SubjectRepository!**  **EconomicalSubject History is created and added to the SubjectRepository!**  **TechnicalSubject ComputerProgramming is created and added to the SubjectRepository!**  **GreatEinstein university is created and added to the UniversityRepository!**  **IbnSena university is created and added to the UniversityRepository!**  **Student Alice Pitt is added to the StudentRepository!**  **Student Boris Grеy is added to the StudentRepository!**  **Student Rob Butterscotch is added to the StudentRepository!**  **Alice Pitt covered Biology exam!**  **Alice Pitt covered Physics exam!**  **Alice Pitt covered Chemistry exam!**  **Rob Butterscotch covered Physics exam!**  **Rob Butterscotch covered Mathematics exam!**  **Alice Pitt has not covered all the required exams for GreatEinstein university!**  **Alice Pitt joined IbnSena university!**  **Rob Butterscotch has not covered all the required exams for GreatEinstein university!**  **Rob Butterscotch covered ComputerProgramming exam!**  **Rob Butterscotch joined GreatEinstein university!**  **\*\*\* GreatEinstein \*\*\***  **Profile: Technical**  **Students admitted: 1**  **University vacancy: 49** |

* **Task 3: Unit Tests (100 points)**

You will receive a skeleton with three classes inside – **UniversityLibrary** and **TextBook**. **UniversityLibrary** class will have some methods, fields, and constructors. Cover the whole class with the unit test to make sure that the class is working as intended. In Judge, you upload **.zip** **(**with **UniversityLibrary.Tests** inside**)** from the **skeleton**.