C# OOP Demo Exam - 04 August 2019

Motocross World Championship MXGP

1. Overview

The FIM (Fédération Internationale de Motocyclisme) Motocross World Championship (MXGP) is the one of the biggest motocross champoinship ever. You love to ride motorcycles and you are the biggest fan on MXGP and for that reason, MXGP hired you to create platform for storing information about riders, motorcycles and races.

2. Setup

- Upload **only the `MXGP`** package in every problem **except Unit Tests**
- **Do not modify the classes, 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

3. Task 1: Structure (50 points)

You are given **8** interfaces, and you have to implement their functionality in the **correct classes**.

It is not required to implement your structure with **`Engine`**, **`CommandHandler`**, **`ConsoleReader`**, **`ConsoleWriter`** and enc. It's good practice but it's not required.

There are **3** types of entities and 3 repositories in the application: **Motorcycle, Rider, Race and Repository**:

Motorcycle

`Motorcycle` is a **base class** for any **type of motorcycle** and it **should not be able to be instantiated**.

Data

- **`Model`** – **`string`**
    - If the model **is null, whitespace or less than 4 symbols,** throw an **`ArgumentException`** with message **`"Model {model} cannot be less than 4 symbols."`**
    - All models are unique
- **`HorsePower`** – **`int`**
    - **Every type** of motorcycle has different range of valid horsepower. If the horsepower is not in the valid range, throw an **`ArgumentException`** with message **`"Invalid horse power: {horsepower}."`**
- **`CubicCentimeters`** – **`double`**
    - **Every type** of motorcycle has different cubic centimeters.

Behavior

`double CalculateRacePoints(int laps)`

The **`CalculateRacePoints`** calculates the race points in the concrete race with this formula:

`cubic centimeters / horsepower * laps`

Constructor

A **`Motorcycle`** should take the following values upon initialization:

`string model, int horsePower, double cubicCentimeters`

Child Classes

There are several concrete types of **Motorcycles**:

PowerMotorcycle

The **cubic centimeters** for this type of motorcycle are **`450`**. Minimum **horsepower** is **`70`** and maximum **horsepower** is **`100`**.

If you receive horsepower which is not in the given range throw **`ArgumentException`** with message **`"Invalid horse power: {horsepower}."`**.

SpeedMotorcycle

The **cubic centimeters** for this type of motorcycle are **`125`**. Minimum **horsepower** is **`50`** and maximum **horsepower** is **`69`**.

If you receive horsepower which is not in the given range throw **`ArgumentException`** with message **`"Invalid horse power: {horsepower}."`.**

Rider

Data

- **`Name`** – **`string`**
  - If the name **is null, empty** or less than **`5`** **symbols** throw an **`ArgumentException`** with message **`"Name {name} cannot be less than 5 symbols."`**
  - All names are unique
- **`Motorcycle`** – **`Motorcycle`**
- **`NumberOfWins`** – **`int`**
- **`CanParticipate`** – **`bool`**
  - Default behaviour is **`false`**
  - Rider can participate in race, **ONLY** if he has motorcycle (motorcycle is not **`null`**)

Behavior

`void AddMotorcycle(Motorcycle motorcycle)`

This method adds a motorcycle to the rider. If the motorcycle **`null`**, throw **`ArgumentNullException`** with message **`"Motorcycle cannot be null."`.**

If motorcycle is not null, save it and after that rider can participate to race.

`void WinRace()`

When rider win race, number of wins should be increased.

Constructor

A **`Rider`** should take the following values upon initialization:

`string name`

Race

Data

- **Name** – **string**
  - If the name **is null, empty** or less than **5 symbols** throw an **ArgumentException** with message **"Name {name} cannot be less than 5 symbols."**
  - All names are unique
- **Laps** – **int**
  - Throws **ArgumentException** with message **"Laps cannot be less than 1."**, if the laps are less than **1**.
- **Riders** – A **collection** of **riders**

Behavior

void AddRider(Rider rider)

This method adds a rider to the race if the rider is valid. If a rider is not valid, throw an exception with the appropriate message.

Exceptions are:

- If a rider is **null** throw an **ArgumentNullException** with message **"Rider cannot be null."**
- If a rider **cannot** participate in the race (the rider doesn't own a motorcycle) throw an **ArgumentException** with message **"Rider {rider name} could not participate in race."**
- If the rider already **exists** in the race throw an **ArgumentNullException** with message: **"Rider {rider name} is already added in {race name} race."**

Repository

The repository holds information about the entity.

Data

- **Models** – A **collection of T (entity)**

Behavior

void Add(T model)

Adds an entity in the collection.

bool Remove(T model)

Removes an entity from the collection.

T GetByName(string name)

Returns an entity with that name.

Collection<T> GetAll()

Returns all entities (unmodifiable)

Child Classes

Create **MotorcycleRepository**, **RiderRepository** and **RaceRepository** repositories.

4. Task 2: Business Logic (150 points)

The Controller Class

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

Note: The `ChampionshipController` class SHOULD NOT handle exceptions! The tests are designed to expect exceptions, not messages!

Note: The `ChampionshipController` class SHOULD HAVE empty constructor!

The first interface is **`IChampionshipController`**. You must implement a **`ChampionshipController`** class, which implements the interface and implements all of its methods. The given methods should have the following logic:

Commands

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

CreateRider Command

Parameters

- **`riderName – string`**

Functionality

Creates a rider with the given name and adds it to the appropriate repository.

The method should **return** the following message:

`"Rider {name} is created."`

If a rider with the given name already exists in the rider repository, throw **`an ArgumentException`** with message **`"Rider {name} is already created."`**

CreateMotorcycle Command

Parameters

- **`type - string`**
- **`model - string`**
- **`horsePower - int`**

Functionality

Create a **motorcycle** with the provided **model** and **horsepower** and add it to the repository. There are two types of motorcycles: **"SpeedMotorcycle"** and **"PowerMotorcycle".**

If the **motorcycle** already exists in the appropriate repository throw an **`ArgumentException`** with following message:

`"Motorcycle {model} is already created."`

If the motorcycle is successfully created, the method should **return** the following message:

`"{"SpeedMotorcycle"/ "PowerMotorcycle"} {model} is created."`

AddMotorcycleToRider Command

Parameters

- **`riderName – String`**

- `motorcycleModel - String`

Functionality

Gives the motorcycle with given name to the rider with given name (if exists).

If the rider **does not exist** in rider repository, throw **`InvalidOperationException`** with message

- `"Rider {name} could not be found."`

If the motorcycle **does not exist** in motorcycle repository, throw **`InvalidOperationException`** with message

- `"Motorcycle {name} could not be found."`

If everything is successful you should add the motorcycle to the rider and return the following message:

- `"Rider {rider name} received motorcycle {motorcycle name}."`

AddRiderToRace Command

Parameters

- `raceName – string`
- `riderName - string`

Functionality

Adds a rider to the race.

If the race **does not exist** in the race repository, throw an **`InvalidOperationException`** with message:

- `"Race {name} could not be found."`

If the rider **does not exist** in the rider repository, throw an **`InvalidOperationException`** with message:

- `"Rider {name} could not be found."`

If everything is successful, you should add the rider to the race and return the following message:

- `"Rider {rider name} added in {race name} race."`

CreateRace Command

Parameters

- `name – string`
- `laps – int`

Functionality

Creates a race with the given name and laps and adds it to the race repository.

If the race with the given name already **exists**, throw an **`InvalidOperationException`** with message:

- `"Race {name} is already created."`

If everything is successful you should return the following message:

- `"Race {name} is created."`

StartRace Command

Parameters

- `raceName – string`

Functionality

This method is the biggest deal. If everything is valid, you should **arrange** all riders and then return the three fastest riders. To do this you should sort all riders in **descending** order by the result of **`CalculateRacePoints`** method in the motorcycle object. At the end, if everything is valid **remove** this race from race repository.

If the race **does not exist** in race repository, throw an **`InvalidOperationException`** with message:

- **`"Race {name} could not be found."`**

If the participants in the race are less than **3**, throw an **`InvalidOperationException`** with message:

- **`"Race {race name} cannot start with less than 3 participants."`**

If everything is successful, you should return the following message:

- **`"Rider {first rider name} wins {race name} race."`**
  `"Rider {second rider name} is second in {race name} race."`
  `"Rider {third rider name} is third in {race name} race."`

End Command

Exit the program.

Input / Output

You are provided with one interface, which will help with the correct execution process of your program. The interface is **`IEngine`** 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:

- **`CreateRider {name}`**
- **`CreateMotorcycle {motorcycle type} {model} {horsepower}`**
- **`AddMotorcycleToRider {rider name} {motorcycle name}`**
- **`AddRiderToRace {race name} {rider name}`**
- **`CreateRace {name} {laps}`**
- **`StartRace {race name}`**
- **`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

`CreateRider Michael`
`CreateRider Peter`
`CreateMotorcycle Speed Honda 60`
`CreateMotorcycle Power Suziki 80`

```
CreateMotorcycle Power Yamaha 70
CreateRace Loket 2
AddMotorcycleToRider Michael Honda
AddMotorcycleToRider Peter Suziki
AddMotorcycleToRider Michael Yamaha
StartRace Loket
AddRiderToRace Loket Michael
AddRiderToRace Loket Peter
StartRace Loket
CreateRider Brian
AddRiderToRace Loket Brian
CreateMotorcycle Speed KTM-SX 55
AddMotorcycleToRider Brian KTM-SX
AddRiderToRace Loket Brian
StartRace Loket
End
```

Output

```
Rider Michael is created.
Rider Peter is created.
SpeedMotorcycle Honda is created.
PowerMotorcycle Suziki is created.
PowerMotorcycle Yamaha is created.
Race Loket is created.
Rider Michael received motorcycle Honda.
Rider Peter received motorcycle Suziki.
Rider Michael received motorcycle Yamaha.
Race Loket cannot start with less than 3 participants.
Rider Michael added in Loket race.
Rider Peter added in Loket race.
Race Loket cannot start with less than 3 participants.
Rider Brian is created.
Rider Brian could not participate in race.
SpeedMotorcycle KTM-SX is created.
Rider Brian received motorcycle KTM-SX.
Rider Brian added in Loket race.
Rider Michael wins Loket race.
Rider Peter is second in Loket race.
Rider Brian is third in Loket race.
```

Input
CreateRider Kevin CreateRider Kevin CreateRider Jose CreateMotorcycle Speed KTM-SX-F 100 CreateMotorcycle Power KTM 100 CreateMotorcycle Power KTM-SX-F 100 CreateMotorcycle Power KTM-SX-F 100 StartRace Imola CreateRace Imola 4 AddRiderToRace Lommel Kevin AddRiderToRace Imola Jose AddRiderToRace Imola Kevin AddMotorcycleToRider Kevin KTM-SX-F AddRiderToRace Imola Kevin CreateMotorcycle Speed Honda 60 CreateMotorcycle Power Suziki 80 CreateMotorcycle Power Yamaha 70 CreateRace Loket 2 CreateRider Michael CreateRider Peter AddMotorcycleToRider Michael Honda AddMotorcycleToRider Peter Suziki AddRiderToRace Imola Michael AddRiderToRace Imola Peter StartRace Imola End
Output
Rider Kevin is created. Rider Kevin is already created. Name Jose cannot be less than 5 symbols. Invalid horse power: 100. Model KTM cannot be less than 4 symbols. PowerMotorcycle KTM-SX-F is created. Motorcycle KTM-SX-F is already created. Race Imola could not be found.

```
Race Imola is created.
Race Lommel could not be found.
Rider Jose could not be found.
Rider Kevin could not participate in race.
Rider Kevin received motorcycle KTM-SX-F.
Rider Kevin added in Imola race.
SpeedMotorcycle Honda is created.
PowerMotorcycle Suziki is created.
PowerMotorcycle Yamaha is created.
Race Loket is created.
Rider Michael is created.
Rider Peter is created.
Rider Michael received motorcycle Honda.
Rider Peter received motorcycle Suziki.
Rider Michael added in Imola race.
Rider Peter added in Imola race.
Rider Peter wins Imola race.
Rider Kevin is second in Imola race.
Rider Michael is third in Imola race.
```

5. Task 3: Unit Tests (100 points)

You will receive a skeleton with **`RaceEntry`**, **`UnitMotorcycle`** and **`UnitRider`** classes inside. The class will have some methods, properties, fields and one constructor, which are working properly. You are **NOT ALLOWED** to change any class. Cover the whole class (**`RaceEntry`**) with unit tests to make sure that the class is working as intended.

You are provided with a **unit test project** in the **project skeleton**. **DO NOT modify its NuGet packages**.

Note: The **TheRace** you need to test is in the **global namespace**, so **remove any using statements,** pointing towards the namespace **TheRace**.

Do **NOT** use **Mocking** in your unit tests!