C# OOP Advanced Exam – Festival Manager

The biggest music festival kingpin in Bulgaria, Bay Kiro is looking for junior developers and has hired you to create a stunning festival management platform, called **Festival Manager**.

Overview

Your task is to build a software project, which manages a **music festival**. This festival will have **sets**, each of which has a list of **performers** (which have a list of **instruments**) and a list of **songs** to play during the set. The sets are **performed in order**. At the end of the program, a **performance report** is generated for each set.

Task I: Structure

Unfortunately, the previous maintainer of this code was doing a pretty terrible job. All he managed to do correctly was **create all the interfaces** and the **`SetController`**. **Do not modify the interfaces or their namespaces!**

The main structure of the program should include the following elements:

- **`Engine`** – **Processes input commands** and **sends them** to the relevant **controllers** to handle
- **`FestivalController`** – **Handles** all the **commands** listed in the **I/O** section
- **`SetController`** – Responsible for **performing the sets**
- **`Stage`** – **Repository**, which **holds** all the **sets**, **songs** and **performers**, and provides **methods** for **retrieving** and **storing** them.

Guidelines

- Make sure your Visual Studio is up-to-date. If it isn't, **disable Lightweight Solution Load**, otherwise you risk a bug where **the .NET Core skeleton doesn't build** (and has red squiggles everywhere).
- Upload **only the `FestivalManager` project** in every problem **except 03. Unit Tests**. For **03. Unit Tests**, upload **only the `FestivalManager.Tests` project** with **all using statements pointing to `FestivalManager` removed**.
- **Do not modify the interfaces or their namespaces!**
- You will have to refactor everything else as you see fit. 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** or **properties** in the concrete class than the interface has defined!
- Make sure you have **no public fields** anywhere.

Below, you will find a detailed description of all entities and their methods.

Sets

An abstract **set** has the following characteristics:

- **`string` ➔ `Name`**
- **`TimeSpan` ➔ `MaxDuration`**
- **`TimeSpan` ➔ `ActualDuration`** – **Sum** of the **duration of all the songs** in the set (**calculated** property)
- **`IReadOnlyCollection` ➔ `Performers`**
- **`IReadOnlyCollection` ➔ `Songs`**

Sets have several **methods**, but the most interesting one is **`bool CanPerform()`**, which ensures the set can be performed under these conditions:

- There is at least **1 performer** in the set
- There is at least **1 song** in the set
- All performers have at least **1 instrument**, which is **not broken**.

The other interesting method is the **`AddSong(ISong song)`** method. If you attempt to add a song, which would be longer than the set allows, throw an **`InvalidOperationException`** with the message **`"Song is over the set limit!"`**.

Create **three classes** for each of the three **types** of sets we have:

- **Short** – **15 minutes long** at max
- **Medium** – **40 minutes long** at max
- **Long** – **60 minutes long** at max

Instruments

An abstract **Instrument** has the following characteristics:

- **`Double`** ➔ **`Wear`**
  - If the wear ever goes **below 0**, set it to 0, and if it ever goes **above 100**, set it to 100
  - All instruments have **100 `Wear` on initialization**
- **`Int`** ➔ **`RepairAmount`**
- **`Bool`** ➔ **`IsBroken`** – true if the **instrument's wear** is **less than or equal to 0** (calculated property).

An **instrument** has **three** methods – **`void WearDown()`** and **`void Repair()`**, which **decrease** and **increase** the wear of the instrument by the **RepairAmount** respectively.

The following classes inherit **Instrument**:

- **`Drums`** – **20 `RepairAmount`**
- **`Guitar`** – **60 `RepairAmount`**
- **`Microphone`** – **80 `RepairAmount`**

Performer

A performer has these characteristics:

- **`string`** ➔ **`Name`**
- **`int`** ➔ **`Age`**
- **`IReadOnlyCollection`** ➔ **`Instruments`**

Song

A song has these characteristics:

- **`string`** ➔ **`Name`**
- **`TimeSpan`** ➔ **`Duration`**

Stage

The **stage** class is a **repository** where all the current **songs**, **performers** and **sets** are **stored**. It has:

- Methods for **Adding sets**, **performers** and **songs**

- Methods for retrieving **sets**, **performers** and **songs by name**
- Methods for checking if a **set**, **performer** or **song exists by name**.

Note: The **`Stage`** is **shared** between the **festival controller** and the **set controller**.

Task II: Business Logic

Your code should only catch exceptions on the **engine level**.

Commands

The software needs to be able to process several commands in the form of **methods**:

RegisterSet {name} {type}

Creates a **set** of the specified **type** with the specified **name** and **adds it to the stage's sets**. Upon a successful set registration, the command returns **`"Registered {type} set"`**.

SignUpPerformer {name} {age} {instrument1} {instrument2} {instrumentN}

Creates a **performer** with the specified **name** and **age**, which holds a list of **instruments** and **adds them to the stage**. Upon successful creation, the command returns **`"Registered performer {performerName}"`**.

Note: **Performers can have no instruments**. This is valid input.

RegisterSong {name} {mm:ss}

Creates a **song** with the specified **name** and **duration** and **adds it to the stage's songs**. Upon successful creation, the command returns **`"Registered song {songName} ({duration:mm\\:ss})"`**.

AddSongToSet {songName} {setName}

Adds the **song** with the given **name** to the **set** with the given **name**.

If the set **doesn't exist** in the stage, throw an **`InvalidOperationException`** with the message **`"Invalid set provided"`**.

If the **song doesn't exist** in the stage, throw an **`InvalidOperationException`** with the message **`"Invalid song provided"`**.

If successful, the command returns **`"Added {songName} ({duration:mm\\:ss}) to {setName}"`**.

AddPerformerToSet {performerName} {setName}

Adds the specified **performer** with the specified **name** to the **set**.

If the **performer doesn't exist** in the stage, throw an **`InvalidOperationException`** with the message **`"Invalid performer provided"`**.

If the set **doesn't exist** in the stage, throw an **`InvalidOperationException`** with the message **`"Invalid set provided"`**.

If successful, the command returns **`"Added {performerName} to {setName}"`**.

RepairInstruments

Gets **all instruments** from **all performers** and **repairs only the ones,** which have a **wear level lower than 100**. When the repairs are finished, the command returns **`"Repaired {repairedInstrumentsCount} instruments`**".

LetsRock

This command is supposed to call **`PerformSets()`**, which is implemented correctly in the skeleton. The command retrieves all sets from the stage, ordered by their actual duration (descending), then by the count of their performers (descending). For each one, if it cannot be performed, **`"-- Did not perform"`** is printed as part of the command output.

If the set can be performed, each song is printed in the format "**`-- {songIndex}. {songName} ({songDuration:mm\\:ss})`**". **After each song**, each performer's instruments are **worn down** (e.g. if the set has **two** songs, they would be worn down **twice**). The command's output ends with **`"-- Set Successful"`** (only if the set was performed).

Tasks III:

Reflection

You need to refactor the given factories and implement new ones. Factories must **use reflection**, so it will be easy for us to follow the **Open/Closed Principle**. You are required to implement **two factories**:

- **`SetFactory`**
- **`InstrumentFactory`**

Feel free to make additional factories for the other entities, even though they will not be tested.

Your task is to implement these factories in such a way that it will be **easy to extend the number of concrete types of each entity**.

NOTE: Make sure you reference the Calling Assembly, instead of the Executing Assembly, since the code that's going to be calling your factories in the tests depends on this!

No static factories are allowed**!**

Unit Testing

Like you saw at the beginning, there is a class, which does not need refactoring - **`SetController`.** This is the class, against which you need to **write unit tests**. In your skeleton, you are provided with a **perfectly working `SetController`**, but it still needs to be **tested**, because in **Judge**, we have prepared some **bugs**, and you need to catch them in your unit tests.

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

Note: The **`SetController`** you need to test is in the **global namespace**, as are any entities, which it depends on, so **remove any using statements** pointing towards any entities and controllers before submitting your code. Do not use the **`FestivalController`** in your tests, as it's not a part of the classes you are provided with.

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

Input

- The input will come from the console in the form of commands, in the format specified above - each command on new line
- The input sequence ends when you receive the command "**END**"
- **Any** type of command, except "**END**" can be given at any time.

Output

The **output** of each command must be printed **on a new line**.

If an exception is thrown because of invalid state (invalid song or set name, etc.), they should be printed in the following format: "**`ERROR: {exceptionMessage}`**".

After the "**END**" command is received, you must **produce a report** in the following format:

The **first line** of the report has the festival's **total length** (**sum** of **all sets' actual duration**) in the following format:

`"Festival length: {totalFestivalLength}"`

After that, **for each of the sets**, print the following info:

First off, print the **set's name** and **actual duration** in the following format:

`"--{setName} ({actualSetDuration}):"`

Then, get all the performers, **ordered by age in descending order**, and for each one, print info in the following format, **ordered descending by wear**:

"**`---{performerName} ({instrument1Name [{wear}%], instrument2Name [{wear}%], etc.})`**".

If an instrument is **broken**, put "**`{instrumentName} [broken]`**" **instead** of its wear.

Then, print information about each song in the set.

If there are **no songs in the set**, print **`"--No songs played"`**. Otherwise, print **`"--Songs played:"`** and for each song, print **`"----{songName} ({songDuration:mm\\:ss})"`**

Constraints

- **All durations will be in the format "`mm:ss`"** and will **always** have **leading zeroes**.
- **All input durations will be at least 00:00 and at most 59:59**.
- A **festival**'s **total duration** can be **60 minutes long or more**.
- All input lines will be **valid** commands with **valid** arguments.
- There will be **at most 30** commands
- **All** rules **specified above** will be **strictly followed**, there will be **NO** unexpected input or conditions

Examples

You are also provided with a **test archive**, which contains the tests below as a **.zip** file.

Input	Output
RegisterSet Set1 Short RegisterSet Set2 Medium SignUpPerformer Ivan 20 Guitar SignUpPerformer Gosho 24 Drums SignUpPerformer Pesho 19 Guitar Microphone RegisterSong Song1 01:02 AddSongToSet Song1 Set1 AddPerformerToSet Gosho Set1 AddPerformerToSet Pesho Set1 LetsRock END	Registered Short set Registered Medium set Registered performer Ivan Registered performer Gosho Registered performer Pesho Registered song Song1 (01:02) Added Song1 (01:02) to Set1 Added Gosho to Set1 Added Pesho to Set1 1. Set1: -- 1. Song1 (01:02) -- Set Successful

	2. Set2: -- Did not perform Results: Festival length: 01:02 --Set1 (01:02): ---Gosho (Drums [80%]) ---Pesho (Guitar [40%], Microphone [20%]) --Songs played: ----Song1 (01:02) --Set2 (00:00): --No songs played
RegisterSet Set1 Short RegisterSet Set2 Medium RegisterSet Set3 Long SignUpPerformer Pesho 23 Guitar SignUpPerformer Ivan 24 Drums SignUpPerformer Gosho 25 Microphone AddPerformerToSet Gosho Set1 AddPerformerToSet Pesho Set1 AddPerformerToSet Ivan Set2 RegisterSong Song1 05:00 RegisterSong Song2 05:00 RegisterSong Song3 05:00 RegisterSong Song4 00:01 AddSongToSet Invalid Set1 AddSongToSet Song2 Invalid AddSongToSet Song1 Set1 AddSongToSet Song2 Set1 AddSongToSet Song3 Set1 AddSongToSet Song4 Set1 LetsRock RepairInstruments END	Registered Short set Registered Medium set Registered Long set Registered performer Pesho Registered performer Ivan Registered performer Gosho Added Gosho to Set1 Added Pesho to Set1 Added Ivan to Set2 Registered song Song1 (05:00) Registered song Song2 (05:00) Registered song Song3 (05:00) Registered song Song4 (00:01) ERROR: Invalid song provided ERROR: Invalid set provided Added Song1 (05:00) to Set1 Added Song2 (05:00) to Set1 Added Song3 (05:00) to Set1 ERROR: Song is over the set limit! 1. Set1: -- 1. Song1 (05:00) -- 2. Song2 (05:00) -- 3. Song3 (05:00) -- Set Successful 2. Set2: -- Did not perform 3. Set3: -- Did not perform Repaired 2 instruments Results: Festival length: 15:00 --Set1 (15:00): ---Gosho (Microphone [80%]) ---Pesho (Guitar [60%]) --Songs played: ----Song1 (05:00) ----Song2 (05:00) ----Song3 (05:00) --Set2 (00:00): ---Ivan (Drums [100%]) --No songs played --Set3 (00:00): --No songs played
RegisterSet TestSet Short SignUpPerformer Gosho 21 Guitar	Registered Short set Registered performer Gosho

Microphone SignUpPerformer Pesho 23 Drums SignUpPerformer Ivan 20 Microphone AddPerformerToSet Ivan TestSet AddPerformerToSet Gosho InvalidSet AddPerformerToSet InvalidPerformer InvalidSet RegisterSong SongName 03:00 RegisterSong SongName2 02:00 RegisterSong SongName3 10:00 RegisterSong SongName4 00:01 AddSongToSet SongName TestSet AddSongToSet SongName2 TestSet AddSongToSet SongName3 TestSet AddSongToSet SongName4 TestSet AddSongToSet InvalidSongName TestSet AddSongToSet SongName InvalidSet LetsRock RepairInstruments RegisterSet TestSet2 Short LetsRock SignUpPerformer Ivancho 20 AddPerformerToSet Ivancho TestSet2 LetsRock END	Registered performer Pesho Registered performer Ivan Added Ivan to TestSet ERROR: Invalid set provided ERROR: Invalid performer provided Registered song SongName (03:00) Registered song SongName2 (02:00) Registered song SongName3 (10:00) Registered song SongName4 (00:01) Added SongName (03:00) to TestSet Added SongName2 (02:00) to TestSet Added SongName3 (10:00) to TestSet ERROR: Song is over the set limit! ERROR: Invalid song provided ERROR: Invalid set provided 1. TestSet: -- 1. SongName (03:00) -- 2. SongName2 (02:00) -- 3. SongName3 (10:00) -- Set Successful Repaired 1 instruments Registered Short set 1. TestSet: -- 1. SongName (03:00) -- 2. SongName2 (02:00) -- 3. SongName3 (10:00) -- Set Successful 2. TestSet2: -- Did not perform Registered performer Ivancho Added Ivancho to TestSet2 1. TestSet: -- Did not perform 2. TestSet2: -- Did not perform Results: Festival length: 15:00 --TestSet (15:00): ---Ivan (Microphone [broken]) --Songs played: ----SongName (03:00) ----SongName2 (02:00) ----SongName3 (10:00) --TestSet2 (00:00): ---Ivancho () --No songs played