## C# OOP Exam – 10 April 2021

## Overview

Aquariums are nice and interesting species can live in there. You have to create an **AquaShop** project, which keeps track of the fish in the aquariums. The Aquariums have **Fish** with different environment requirements. Your task is to add, feed and take care of the fish.

## Setup

* Upload **only the** AquaShopproject in every problem **except** **Unit Tests**
* **Do not modify the interfaces or their namespaces**
* 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

## Task 1: Structure (50 points)

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

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

There are **3** types of entities in the application: **Aquarium, Fish, Decoration**. There should also be **DecorationRepository**.

### Decoration

Decoration is a **base class** of any **type of decoration** and it **should not be able to be instantiated**.

#### Data

* **Comfort** - **int**
* **Price** - **decimal**
  + The price of the decoration

#### Constructor

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

int comfort, decimal price

public abstract class Decoration : IDecoration

{

private int comfort;

private decimal price;

public Decoration(int comfort, decimal price)

{

this.Comfort = comfort;

this.Price = price;

}

public int Comfort { get; }

public decimal Price { get; }

}

#### Child Classes

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

##### Ornament

Has **1 comfort** and its **price** is **5**.

Constructorshould take no values upon initialization.

##### Plant

Has **5 comfort** and its **price** is **10**.

Constructorshould take no values upon initialization.

### Fish

Fish is a **base class** of any **type of fish** and it **should not be able to be instantiated**.

#### Data

* **Name** - **string**
  + If the name **is null or whitespace,** throw an **ArgumentException** with message: "Fish name cannot be null or empty."
  + All names are unique
* **Species** - **string**
  + If the species **is null or whitespace,** throw an **ArgumentException** with message: "Fish species cannot be null or empty."
* **Size** - **int**
  + The size of the **Fish**
* **Price** - **decimal**
  + The price of the **Fish**
  + If the price is below or equal **0,** throw an **ArgumentException** with message:

"Fish price cannot be below or equal to 0."

#### Behavior

##### abstract void Eat()

The **Eat()** method increases the **Fish**’s size.

#### Constructor

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

string name, string species, decimal price

public abstract class Fish : IFish

{

private string name;

private string species;

private decimal price;

public Fish(string name, string species, decimal price)

{

this.Name = name;

this.Species = species;

this.Price = price;

}

public string Name

{

get

{

return this.name;

}

private set

{

if (string.IsNullOrWhiteSpace(value))

{

throw new ArgumentException(ExceptionMessages.InvalidFishName);

}

this.name = value;

}

}

public string Species

{

get

{

return this.species;

}

private set

{

if (string.IsNullOrWhiteSpace(value))

{

throw new ArgumentException(ExceptionMessages.InvalidFishSpecies);

}

this.species = value;

}

}

public int Size { get; protected set; }

public decimal Price

{

get

{

return this.price;

}

private set

{

if (value <= 0)

{

throw new ArgumentException(ExceptionMessages.InvalidFishPrice);

}

this.price = value;

}

}

public abstract void Eat();

}

#### Child Classes

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

##### FreshwaterFish

Has **3 initial size**.

**Can only live in FreshwaterAquarium!**

Constructorshould take the following values upon initialization:

string name, string species, decimal price

###### **Behavior**

**void Eat()**

* The method **increases** the fish’s size by **3**.

##### SaltwaterFish

Has **5 initial size**.

**Can only live in SaltwaterAquarium!**

Constructorshould take the following values upon initialization:

string name, string species, decimal price

###### **Behavior**

**void Eat()**

* The method **increases** the fish’s size by **2**.

### Aquarium

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

#### Data

* **Name** - **string**
  + If the name **is null or whitespace,** throw an **ArgumentException** with message: "Aquarium name cannot be null or empty."
  + All names are unique
* **Capacity** - **int**
  + The **number** of **Fish** аn **Aquarium** **can have**
* **Decorations** - **ICollection<IDecoration>**
* **Fish** -**ICollection<IFish>**
* **Comfort** - calculated property, which returns **int**
  + How is it calculated: The **sum** of **each decoration’s comfort** in the **Aquarium**

#### Behavior

##### void AddFish(IFish fish)

**Adds** a **Fish** in the **Aquarium** if there is **capacity** for it, otherwise throw an **InvalidOperationException** with message "Not enough capacity.";

##### bool RemoveFish(IFish fish)

Removes a **Fish** from the **Aquarium**. Returns **true** if the **Fish** is removed successfully, otherwise - **false**.

##### void AddDecoration(IDecoration decoration)

Adds a **Decoration** in the **Aquarium**.

##### void Feed()

The **Feed()** method **feeds** **all fish**, calls their **Eat()** method.

##### string GetInfo()

**Returns** a **string** with **information** about the **Aquarium** in the format below. If the **Aquarium doesn't have fish**, print **"none"** instead.

"{aquariumName} ({aquariumType}):  
Fish: {fishName1}, {fishName2}, {fishName3} (…) / none  
Decorations: {decorationsCount}  
Comfort: {aquariumComfort}"

#### Constructor

An **Aquarium** should take the following values upon initialization:

string name, int capacity

public abstract class Aquarium : IAquarium

{

private string name;

protected Aquarium(string name, int capacity)

{

this.Name = name;

this.Capacity = capacity;

this.Decorations = new List<IDecoration>();

this.Fish = new List<IFish>();

}

public string Name

{

get

{

return this.name;

}

private set

{

if (string.IsNullOrWhiteSpace(value))

{

throw new ArgumentException(ExceptionMessages.InvalidAquariumName);

}

this.name = value;

}

}

public int Capacity { get; }

public int Comfort => this.Decorations.Sum(x => x.Comfort);

public ICollection<IDecoration> Decorations { get; }

public ICollection<IFish> Fish { get; }

public void AddDecoration(IDecoration decoration) => this.Decorations.Add(decoration);

public void AddFish(IFish fish)

{

if (this.Fish.Count >= this.Capacity)

{

throw new InvalidOperationException(ExceptionMessages.NotEnoughCapacity);

}

this.Fish.Add(fish);

}

public void Feed()

{

foreach (IFish fish in this.Fish)

{

fish.Eat();

}

}

public string GetInfo()

{

StringBuilder sb = new StringBuilder();

sb.AppendLine($"{this.Name} ({this.GetType().Name}):");

if (this.Fish.Any())

{

sb.AppendLine($"Fish: {String.Join(", ", this.Fish.Select(f => f.Name))}");

}

else

{

sb.AppendLine($"Fish: none");

}

//sb.AppendLine($"Fish: {(this.Fish.Any() ? String.Join(", " , this.Fish.Select(f => f.Name)) : "none")}");

sb.AppendLine($"Decorations: {this.Decorations.Count}");

sb.AppendLine($"Comfort: {this.Comfort}");

return sb.ToString();

}

public bool RemoveFish(IFish fish)

{

this.Fish.Remove(fish);

if (this.Fish.Contains(fish))

{

return false;

}

return true;

}

#### Child Classes

There are 2 concrete types of **Aquarium**:

##### FreshwaterAquarium

Has **50 capacity**.

Constructorshould take the following values upon initialization:

string name

##### SaltwaterAquarium

Has **25 capacity**

Constructorshould take the following values upon initialization:

string name

### DecorationRepository

The **decoration repository** is a **repository** for the **decorations** that are in the **AquaShop**.

#### Data

* Models - **a** **collection of decorations (unmodifiable)**

#### Behavior

##### void Add(IDecoration decoration)

* **Adds** a **decoration** in the **collection**.

**bool Remove(IDecoration decoration)**

* **Removes** a **decoration** from the **collection**. **Returns true** if the deletion was **sucessful**, **otherwise** - **false**.

**IDecoration FindByType(string type)**

* **Returns** the **first** **decoration** of the **given type**, if there is. **Otherwise**, returns **null**.

public class DecorationRepository : IRepository<IDecoration>

{

private List<IDecoration> decorations;

public DecorationRepository()

{

decorations = new List<IDecoration>();

}

public IReadOnlyCollection<IDecoration> Models => this.decorations.AsReadOnly();

public void Add(IDecoration model) => this.decorations.Add(model);

public IDecoration FindByType(string type) => this.decorations.FirstOrDefault(d => d.GetType().Name == type);

public bool Remove(IDecoration model) => this.decorations.Remove(model);

}

## 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** Controller **class SHOULD NOT handle exceptions! The tests are designed to expect exceptions, not messages!**

The first interface is **I**Controller. You must create a Controllerclass, which implements the interface and implements all of its methods. The constructor of Controllerdoes not take any arguments. The given methods should have the logic described for each in the Commands section.

### Data

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

* **decorations** - **DecorationRepository**
* **aquariums** - **collection of IAquarium**

### Commands

There are several **commands**, which control the **business** **logic** of the **application**. They are **stated** **below**. The **Aquarium** **name** passed to the methods will **always** be **valid**!

#### AddAquarium Command

##### Parameters

* aquariumType - string
* aquariumName - string

##### Functionality

**Adds** an Aquarium. **Valid** types are: "**FreshwaterAquarium**" and "**SaltwaterAquarium**".

If the **Aquarium** **type** is **invalid**, you have to **throw an InvalidOperationException** with **the following message:**

* "Invalid aquarium type."

If the **Aquarium** is **added successfully**, the method should **return** the following **string**:

* "Successfully added {aquariumType}."

#### AddDecoration Command

##### Parameters

* **type** - **string**

##### Functionality

**Creates** a **decoration** of the **given type** and **adds** it to the **DecorationRepository**. **Valid** types are: "**Ornament**" and "**Plant**". If the decoration **type** is **invalid**, throw an **InvalidOperationException** with message:

* "Invalid decoration type."

The **method** should **return** the following **string** if the **operation** is **successful**:

* "Successfully added {decorationType}."

#### InsertDecoration Command

##### Parameters

* aquariumName - string
* decorationType - string

##### Functionality

**Adds** the desired Decoration to the Aquarium with the **given name**. You have to remove the Decoration from the DecorationRepository if the insert is **successful**.

If there is **no such decoration**, you have to **throw an InvalidOperationException** with **the following message**:

* "There isn't a decoration of type {decorationType}."

If **no errors** are **thrown**, **return** a string with the following message "Successfully added {decorationType} to {aquariumName}.".

#### AddFish Command

##### Parameters

* aquariumName - string
* fishType - string
* fishName - string
* fishSpecies - string
* price - decimal

##### Functionality

**Adds** the desired Fish to the Aquarium with the **given name**. **Valid** Fish types are: "**FreshwaterFish**", "**SaltwaterFish**".

If the **Fish** **type** is **invalid**, you have to **throw an** **InvalidOperationException** with **the following message** "Invalid fish type.".

If **no errors** are **thrown**, **return** one of the following messages:

* "Water not suitable." - if the **Fish** **cannot live** in the **Aquarium**
* "Successfully added {fishType} to {aquariumName}." - if the **Fish** is **added successfully** in the **Aquarium**

#### FeedFish Command

##### Parameters

* **aquariumName** - **string**

##### Functionality

Feeds all **Fish** in the **Aquarium** with the given name.

**Returns** a **string** with information about **how many fish** were **fed**, in the following **format**:

* "Fish fed: {fedCount}"

#### CalculateValue Command

##### Parameters

* **aquariumName** - **string**

##### Functionality

Calculates the value of the **Aquarium** with the given name. It is calculated by the sum of all **Fish**’s and **Decorations**’ prices in the **Aquarium**.

**Return** a **string** in the following **format**:

* "The value of Aquarium {aquariumName} is {value}."
  + The **value** should be **formatted** to the **2nd decimal place**!

#### Report Command

##### Functionality

Returns information about each aquarium. You can use the overridden **GetInfo Aquarium** method.

"{aquariumName} ({aquariumType}):  
Fish: {fishName1}, {fishName2}, {fishName3} (…) / none  
Decorations: {decorationsCount}  
Comfort: {aquariumComfort}

{aquariumName} ({aquariumType}):  
Fish: {fishName1}, {fishName2}, {fishName3} (…) / none  
Decorations: {decorationsCount}  
Comfort: {aquariumComfort}

(…)"

**Note: Use \r\n or Environment.NewLine for a new line. There is not an empty row between different aquariums.**

#### Exit Command

##### Functionality

Ends the program.

### Input / Output

You are provided with one interface, which will help you 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.

You are given the **Engine** class with written logic in it. In order the code to be **compiled**, some parts are **commented**, **don’t forget to comment them out**. The **try-catch block** is also **commented** in order for the program to **throw exceptions and for you to see them**, **comment it out** when you are **ready** with this too.

#### Input

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

* **AddAquarium** **{aquariumType} {aquariumName}**
* **AddDecoration** **{decorationType}**
* **InsertDecoration** **{aquariumName} {decorationType}**
* **AddFish {aquariumName} {fishType} {fishName} {fishSpecies} {price}**
* **FeedFish {aquariumName}**
* **CalculateValue {aquariumName}**
* **Report**
* **Exit**

#### 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** |
| **AddAquarium SaltwaterAquarium Underworld**  **AddAquarium FreshwaterAquarium Swamp**  **AddFish Underworld FreshwaterFish Nemo Clownfish 13.40**  **AddFish Underworld SaltwaterFish Nemo Clownfish 13.40**  **AddAquarium FreshwaterAquarium Riverworld**  **AddFish Riverworld FreshwaterFish Emerald Catfish 7.32**  **AddFish Underworld SweetwaterFish Diamond Catfish 3.50**  **AddDecoration Plant**  **InsertDecoration Riverworld Plant**  **InsertDecoration Underworld Plant**  **AddDecoration Plant**  **InsertDecoration Underworld Plant**  **FeedFish Riverworld**  **AddFish Riverworld FreshwaterFish Species 20**  **AddFish Riverworld FreshwaterFish Name 20**  **AddFish Riverworld FreshwaterFish Name Species -10**  **Report**  **Exit** |
| **Output** |
| **Successfully added SaltwaterAquarium.**  **Successfully added FreshwaterAquarium.**  **Water not suitable.**  **Successfully added SaltwaterFish to Underworld.**  **Successfully added FreshwaterAquarium.**  **Successfully added FreshwaterFish to Riverworld.**  **Invalid fish type.**  **Successfully added Plant.**  **Successfully added Plant to Riverworld.**  **There isn't a decoration of type Plant.**  **Successfully added Plant.**  **Successfully added Plant to Underworld.**  **Fish fed: 1**  **Fish name cannot be null or empty.**  **Fish species cannot be null or empty.**  **Fish price cannot be below or equal to 0.**  **Underworld (SaltwaterAquarium):**  **Fish: Nemo**  **Decorations: 1**  **Comfort: 5**  **Swamp (FreshwaterAquarium):**  **Fish: none**  **Decorations: 0**  **Comfort: 0**  **Riverworld (FreshwaterAquarium):**  **Fish: Emerald**  **Decorations: 1**  **Comfort: 5** |

|  |
| --- |
| **Input** |
| **AddAquarium SaltwaterAquarium DangerZone**  **AddDecoration Plant**  **AddDecoration Plant**  **AddDecoration Ornament**  **InsertDecoration DangerZone Plant**  **InsertDecoration DangerZone Plant**  **InsertDecoration DangerZone Ornament**  **AddFish DangerZone SaltwaterFish Curibou Angelfish 22.33**  **AddFish DangerZone SaltwaterFish Devil Anglerfish 48.84**  **FeedFish DangerZone**  **AddFish DangerZone EuryhalineFish Greeny Chromide 9.99**  **CalculateValue DangerZone**  **FeedFish DangerZone**  **Report**  **Exit** |
| **Output** |
| **Successfully added SaltwaterAquarium.**  **Successfully added Plant.**  **Successfully added Plant.**  **Successfully added Ornament.**  **Successfully added Plant to DangerZone.**  **Successfully added Plant to DangerZone.**  **Successfully added Ornament to DangerZone.**  **Successfully added SaltwaterFish to DangerZone.**  **Successfully added SaltwaterFish to DangerZone.**  **Fish fed: 2**  **Invalid fish type.**  **The value of Aquarium DangerZone is 96.17.**  **Fish fed: 2**  **DangerZone (SaltwaterAquarium):**  **Fish: Curibou, Devil**  **Decorations: 3**  **Comfort: 11** |

## public class Controller : IController

## {

## private List<IAquarium> aquariums;

## private DecorationRepository decorations;

## public Controller()

## {

## this.aquariums = new List<IAquarium>();

## this.decorations = new DecorationRepository();

## }

## public string AddAquarium(string aquariumType, string aquariumName)

## {

## IAquarium aquarium = default;

## if (aquariumType == "FreshwaterAquarium")

## {

## aquarium = new FreshwaterAquarium(aquariumName);

## }

## else if (aquariumType == "SaltwaterAquarium")

## {

## aquarium = new SaltwaterAquarium(aquariumName);

## }

## else

## {

## throw new InvalidOperationException(ExceptionMessages.InvalidAquariumType);

## }

## aquariums.Add(aquarium);

## return string.Format(OutputMessages.SuccessfullyAdded, aquariumType);

## }

## public string AddDecoration(string decorationType)

## {

## IDecoration decoration = default;

## if (decorationType == "Ornament")

## {

## decoration = new Ornament();

## }

## else if (decorationType == "Plant")

## {

## decoration = new Plant();

## }

## else

## {

## throw new InvalidOperationException(ExceptionMessages.InvalidDecorationType);

## }

## decorations.Add(decoration);

## return string.Format(OutputMessages.SuccessfullyAdded, decorationType);

## }

## public string AddFish(string aquariumName, string fishType, string fishName, string fishSpecies, decimal price)

## {

## IFish fish;

## IAquarium currentAquarium = this.aquariums.FirstOrDefault(a => a.Name == aquariumName);

## if (fishType == nameof(FreshwaterFish))

## {

## fish = new FreshwaterFish(fishName, fishSpecies, price);

## if (currentAquarium.GetType().Name != nameof(FreshwaterAquarium))

## {

## return OutputMessages.UnsuitableWater;

## }

## currentAquarium.AddFish(fish);

## }

## else if (fishType == nameof(SaltwaterFish))

## {

## fish = new SaltwaterFish(fishName, fishSpecies, price);

## if (currentAquarium.GetType().Name != nameof(SaltwaterAquarium))

## {

## return OutputMessages.UnsuitableWater;

## }

## currentAquarium.AddFish(fish);

## }

## else

## {

## throw new InvalidOperationException(ExceptionMessages.InvalidFishType);

## }

## return string.Format(OutputMessages.EntityAddedToAquarium, fishType, aquariumName);

## }

## public string CalculateValue(string aquariumName)

## {

## IAquarium aquarium = this.aquariums.FirstOrDefault(a => a.Name == aquariumName);

## decimal fishPrice = aquarium.Fish.Sum(f => f.Price);

## decimal decorationPrice = aquarium.Decorations.Sum(d => d.Price);

## decimal totalPrice = decorationPrice + fishPrice;

## return string.Format(OutputMessages.AquariumValue, aquariumName, totalPrice);

## }

## public string FeedFish(string aquariumName)

## {

## IAquarium currentAquarium = this.aquariums.FirstOrDefault(a => a.Name == aquariumName);

## currentAquarium.Feed();

## return string.Format(OutputMessages.FishFed, currentAquarium.Fish.Count);

## }

## public string InsertDecoration(string aquariumName, string decorationType)

## {

## IDecoration currentDecoration = decorations.FindByType(decorationType);

## if (currentDecoration == null)

## {

## throw new InvalidOperationException(string.Format(ExceptionMessages.InexistentDecoration, decorationType));

## }

## IAquarium currentAquarium = aquariums.FirstOrDefault(a => a.Name == aquariumName);

## currentAquarium.AddDecoration(currentDecoration);

## this.decorations.Remove(currentDecoration);

## return string.Format(OutputMessages.EntityAddedToAquarium, decorationType, aquariumName);

## }

## public string Report()

## {

## StringBuilder sb = new StringBuilder();

## foreach (IAquarium aquarium in aquariums)

## {

## sb.Append(aquarium.GetInfo() + Environment.NewLine);

## }

## return sb.ToString().TrimEnd();

## }

## Task 3: Unit Tests (100 points)

You will receive a skeleton with **Fish** and **Aquarium** classes inside. The class will have some methods, fields and one constructor, which are working properly. You are **NOT ALLOWED** to change any class. Cover the whole class 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** use **Mocking** in your unit tests!

namespace Aquariums.Tests

{

using NUnit.Framework;

using System;

[TestFixture]

public class AquariumsTests

{

[Test]

public void Ctor()

{

Aquarium aquarium = new Aquarium("a", 1);

Assert.That(aquarium.Name == "a");

Assert.That(aquarium.Capacity == 1);

}

[Test]

public void NameIsInCorrect()

{

Assert.Throws<ArgumentNullException>(() => new Aquarium(null, 1));

Assert.Throws<ArgumentNullException>(() => new Aquarium(String.Empty, 1));

}

[Test]

public void NameIsCorrect()

{

Aquarium aquarium = new Aquarium("a", 1);

Assert.That("a", Is.EqualTo(aquarium.Name));

}

[Test]

public void CapacityIsInCorrect()

{

Assert.Throws<ArgumentException>(() => new Aquarium("a", -1));

}

[Test]

public void CapacityIsCorrect()

{

Aquarium aquarium = new Aquarium("a", 1);

Assert.That(1, Is.EqualTo(aquarium.Capacity));

}

[Test]

public void FishCountIsCorrect()

{

Aquarium aquarium = new Aquarium("a", 5);

Fish fish1 = new Fish("Poly");

Fish fish2 = new Fish("Petra");

aquarium.Add(fish1);

aquarium.Add(fish2);

Assert.AreEqual(2, aquarium.Count);

}

[Test]

public void AddFishIsInCorrect()

{

Aquarium aquarium = new Aquarium("a", 1);

Fish fish1 = new Fish("Poly");

Fish fish2 = new Fish("Petra");

aquarium.Add(fish1);

Assert.Throws<InvalidOperationException>(()=> aquarium.Add(fish2));

}

[Test]

public void AddFishIsCorrect()

{

Aquarium aquarium = new Aquarium("a", 1);

Fish fish1 = new Fish("Poly");

aquarium.Add(fish1);

Assert.That(fish1.Available == true);

}

[Test]

public void AddFishIsNotCorrect()

{

Aquarium aquarium = new Aquarium("a", 0);

Assert.Throws<InvalidOperationException>(()=> aquarium.Add(new Fish("Peppy")));

}

[Test]

public void RemoveFishIsCorrect()

{

Aquarium aquarium = new Aquarium("a", 5);

Fish fish1 = new Fish("Poly");

Fish fish2 = new Fish("Petra");

aquarium.Add(fish1);

aquarium.Add(fish2);

aquarium.RemoveFish("Poly");

Assert.AreEqual(1, aquarium.Count);

}

[Test]

public void RemoveFishIsNotCorrect()

{

Aquarium aquarium = new Aquarium("a", 5);

Assert.Throws<InvalidOperationException>(()=> aquarium.RemoveFish(null));

}

[Test]

public void RemoveFishIsInCorrect()

{

Aquarium aquarium = new Aquarium("a", 5);

Fish fish1 = new Fish("Poly");

aquarium.Add(fish1);

Assert.Throws<InvalidOperationException>(()=> aquarium.RemoveFish("Koko"));

}

[Test]

public void SellFishIsInCorrect()

{

Aquarium aquarium = new Aquarium("a", 5);

Fish fish1 = new Fish("Poly");

aquarium.Add(fish1);

Assert.Throws<InvalidOperationException>(()=> aquarium.SellFish("Koko"));

}

[Test]

public void SellFishIsNotCorrect()

{

Aquarium aquarium = new Aquarium("a", 5);

Assert.Throws<InvalidOperationException>(()=> aquarium.SellFish(null));

}

[Test]

public void SellFishIsCorrect()

{

Aquarium aquarium = new Aquarium("a", 5);

aquarium.Add(new Fish("Poly"));

Fish fish = aquarium.SellFish("Poly");

Assert.AreEqual(fish.Name, "Poly");

Assert.AreEqual(fish.Available, false);

}

[Test]

public void ReportIsCorrect()

{

Aquarium aquarium = new Aquarium("a", 5);

Fish fish1 = new Fish("Poly");

Fish fish2 = new Fish("Petra");

aquarium.Add(fish1);

aquarium.Add(fish2);

Assert.That(aquarium.Count, Is.EqualTo(2));

Assert.That(fish1.Available == true);

Assert.That(fish2.Available == true);

Assert.That(aquarium.Report(), Is.EqualTo("Fish available at a: Poly, Petra"));

}

}

}