OOP Basics Exam Retake - Storage Master

After the perilous journey of the OOP Basics exam's battles and magical potions, it's time to drop back down to the real world for a second – to the exciting world of "retail warehouse management"...

Overview

In this exam, you need to build a warehouse management project, which has support for products, storage for storing products, and vehicles for transporting products from one storage to another. The project will consist of entity classes and a controller class, which manages the interaction between the storage, vehicles and products.

Setup

To set up your project, create a new Visual Studio project with the name "StorageMaster". The project must have a **StartUp** class with the namespace "**StorageMaster**". You are free to use any namespaces you want, as long as you have a class, called StartUp in the StorageMaster namespace. Not following this rule will lead to your code not compiling in the Judge system.

Task 1: Structure (150 points)

There are 3 types of entities in the application: **Products**, **Storage** and **Vehicles**:

Products

The **Product** is a **base class** for any **products** and it **should not be able to be instantiated**.

Data

- Price double
 - If a negative price is entered, throw an InvalidOperationException with the message "Price cannot be negative!"
- Weight double

Constructor

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

double price, double weight

Child Classes

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

- **Gpu** always has **0.7** weight
- **HardDrive** always has **1** weight
- Ram always has 0.1 weight
- **SolidStateDrive** always has **0.2** weight

Each type of product only receives its price upon initialization.

Vehicles

The Vehicle is a base class for any vehicles and it should not be able to be instantiated.

Data

Capacity - int





















- Trunk IReadOnlyCollection of Products
- IsFull bool
 - Returns true if the sum of the products' weights is equal to or larger than the vehicle capacity (calculated property)
- IsEmpty bool
 - Returns true if the vehicle doesn't have any products in the trunk (calculated property)

Constructor

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

int capacity

Behavior

void LoadProduct(Product product)

If the vehicle is already full, throw an InvalidOperationException with the message "Vehicle is full!".

If this check passes, the product is added to the vehicle's trunk.

Product Unload()

If the vehicle's trunk is empty, throw an InvalidOperationException with the message "No products left in vehicle!".

If this check passes, the last product in the trunk is removed from the vehicle's trunk and returned to the caller.

Child Classes

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

- Van always has 2 capacity
- Truck always has 5 capacity
- **Semi** always has **10** capacity

Concrete vehicles **don't receive** anything upon initialization.

Storage

The Storage is a base class for any storage and it should not be able to be instantiated.

The storage is a building, which holds products. It also has a garage of vehicles with a fixed length. The length is determined by the garage slots of the storage.

Data

- Name string
- Capacity int the maximum weight of products the storage can handle
- GarageSlots int the number of garage slots the storage's garage has
- IsFull bool
 - Returns true if the sum of the products' weights is equal to or larger than the storage capacity (calculated property)
- Garage IReadOnlyCollection of vehicles
 - Read-only representation of the garage array.
- Products IReadOnlyCollection of products
 - Read-only representation of the products in storage.



© Software University Foundation. This work is licensed under the CC-BY-NC-SA license.



















Constructor

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

string name, int capacity, int garageSlots, IEnumerable<Vehicle> vehicles

Behavior

Vehicle GetVehicle(int garageSlot)

If the provided garage slot number is equal to or larger than the garage slots, throw an InvalidOperationException with the message "Invalid garage slot!".

If the garage slot is empty, throw an InvalidOperationException with the message "No vehicle in this garage slot!"

The method returns the retrieved vehicle.

int SendVehicleTo(int garageSlot, Storage deliveryLocation)

Gets the vehicle from the specified garage slot (and delegates the validation of the garage slot to the GetVehicle method).

Then, the method checks if there are any free garage slots. A free garage slot is denoted by a null value.

If there is no free garage slot, throw an InvalidOperationException with the message "No room in garage!".

Then, the garage slot in the source storage is **freed** and the vehicle is added to the **first free garage slot**.

The method returns the garage slot the vehicle was assigned when it was transferred.

int UnloadVehicle(int garageSlot)

If the storage is full, throw an InvalidOperationException with the message "Storage is full!".

Gets the vehicle from the specified garage slot (and delegates the validation of the garage slot to the GetVehicle method).

Then, until the vehicle empties, or the storage fills up, the vehicle's products are unpacked and are added to the storage's products.

The method returns the number of unloaded products.

Child Classes

There are several concrete types of storages and each of them has a default set of vehicles:

- AutomatedWarehouse always has 1 capacity and 2 garage slots
 - Default vehicles: 1 Truck
- **DistributionCenter** always has **2** capacity and **5** garage slots
 - Default vehicles: 3 Vans
- Warehouse always has 10 capacity and 10 garage slots
 - Default vehicles: 3 Semi trucks

Each type of storage receives a name upon initialization.

















Task 2: Business Logic (200 points)

The Controller Class

The business logic of the program should be concentrated around several commands. Implement a class called StorageMaster, which will hold the main functionality.

The Storage Master keeps track of the storage registry and the products pool (the products in the main storage). It also keeps track of the current vehicle (explained below).

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

The main functionality is represented by these **public methods**:

```
StorageMaster.cs
public string AddProduct(string type, double price)
  throw new NotImplementedException();
public string RegisterStorage(string type, string name)
 throw new NotImplementedException();
public string SelectVehicle(string storageName, int garageSlot)
  throw new NotImplementedException();
public string LoadVehicle(IEnumerable<string> productNames)
 throw new NotImplementedException();
public string SendVehicleTo(string sourceName, int sourceGarageSlot, string destinationName)
  throw new NotImplementedException();
public string UnloadVehicle(string storageName, int garageSlot)
  throw new NotImplementedException();
public string GetStorageStatus(string storageName)
  throw new NotImplementedException();
public string GetSummary()
  throw new NotImplementedException();
```

NOTE: The StorageMaster class should not handle any exceptions. That should be the responsibility of the class, which reads the commands and passes them to the StorageMaster.

















Commands

There are several commands that control the business logic of the application and you are supposed to build. They are stated below.

AddProduct Command

Parameters

- type string
- price double

Functionality

Creates a product and adds it to the product pool.

If the product's type is invalid, throw an InvalidOperationException with the message "Invalid product type!"

Returns "Added {type} to pool".

RegisterStorage Command

Parameters

- type string
- name string

Functionality

Creates a storage and adds it to the storage registry.

If the storage's type is invalid, throw an InvalidOperationException with the message "Invalid storage type!"

Returns "Registered {storageName}".

SelectVehicle Command

Parameters

- storageName string
- garageSlot int

Functionality

Sets the current vehicle to the vehicle in that storage's garage slot. The current vehicle is the vehicle, which the LoadVehicle method will interact with.

Returns "Selected {vehicleType}".

LoadVehicle Command

Parameters

productNames – IEnumerable<string>

Functionality

Loads the current vehicle with as many of the provided product types as possible without filling up the vehicle.

The method goes through each of the product names and performs the following operations:

If there are no items in the product pool with that name, throw an InvalidOperationException with the message "{name} is out of stock!".

If there are, the last product with that name in the pool is removed from the pool and loaded in the vehicle.















Returns "Loaded {loadedProductsCount}/{productCount} products into {vehicleType}".

Note: The productCount is just the number of products the command received as a parameter.

SendVehicleTo Command

Parameters

- sourceName string
- garageSlot int
- destinationName string

Functionality

If either the source storage or the destination storages don't exist, throw an InvalidOperationException with the message "Invalid source storage!" or "Invalid destination storage!"

Then, the method gets the vehicle from the storage at the provided garage slot and sends it to the destination storage.

Returns "Sent {vehicleType} to {destinationName} (slot {destinationGarageSlot})".

UnloadVehicle Command

Parameters

- storageName string
- garageSlot int

Functionality

The method **gets the vehicle** in the storage's **garage slot**. Then, the vehicle is **unloaded** at the storage.

The method returns "Unloaded {unloadedProductsCount}/{productsInVehicle} products at {storageName}".

GetStorageStatus Command

Parameters

storageName – string

Functionality

The method gets the storage with that name from the storage registry and performs some aggregation on it:

The storage's products are counted, grouped by name, sorted by the product count (descending), then by product name (ascending).

Then, every vehicle's **name** in the garage is retrieved. If there is no vehicle in that garage, put "**empty**" in its garage slot.

The command produces two lines:

The first line is the stock format: "Stock ($\{0\}/\{1\}$): [$\{2\}$]". The first parameter is the sum of the products' weight, the second parameter is the storage's capacity. The third parameter is the stock info, described above, separated by commas.

The second line is the garage format: "Garage: [{0}]". The only parameter is the vehicle names (and empty garage slots), separated by a pipe character "|".

The method returns these two lines, separated by a **new line**.

For examples, check the sample input in the **I/O** section.



















GetSummary Command

Functionality

The method gets all the storages in the storage registry, ordered by the sum of their products' price (descending). For each one, a string is produced in the following format:

```
{storageName}:
Storage worth: ${totalMoney:F2}
```

The method returns all the **formatted storage strings**, separated by **new lines**.

Task 3: Input / Output (100 points)

Input

• You will receive commands until you receive "END" as a command.

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

- AddProduct {type} {price}
- RegisterStorage {type} {name}
- SelectVehicle {storageName} {garageSlot}
- LoadVehicle {productName1} {productName2} {productNameN}
- SendVehicleTo {sourceName} {sourceGarageSlot} {destinationName}
- UnloadVehicle {storageName} {garageSlot}
- GetStorageStatus {storageName}

Output

Print the output from each command when issued. When the end command is received, print the output from the **GetSummary** command.

If an InvalidOperationException is thrown during any of the commands' execution, print:

• "Error:" plus the message of the exception

Constraints

The commands will always be in the provided format.

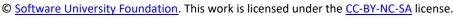
Examples

Input

RegisterStorage DistributionCenter SofiaDistribution
RegisterStorage Warehouse AmazonWarehouse
AddProduct Gpu 1200
AddProduct SolidStateDrive 205
AddProduct HardDrive 70
AddProduct HardDrive 120
SelectVehicle SofiaDistribution 0
LoadVehicle HardDrive Gpu
SendVehicleTo SofiaDistribution 0 AmazonWarehouse



UnloadVehicle AmazonWarehouse 3





















END

Output

Registered SofiaDistribution Registered AmazonWarehouse

Added Gpu to pool

Added SolidStateDrive to pool

Added HardDrive to pool Added HardDrive to pool

Selected Van

Loaded 2/2 products into Van

Sent Van to AmazonWarehouse (slot 3) Unloaded 2/2 products at AmazonWarehouse

AmazonWarehouse:

Storage worth: \$1320.00 SofiaDistribution: Storage worth: \$0.00

Input

AddProduct HardDrive -20

RegisterStorage InvalidStorage LoshHackerStorage

RegisterStorage Warehouse GoodHackerStorage

SelectVehicle GoodHackerStorage 0

LoadVehicle HardDrive

SendVehicleTo LoshHackerStorage 0 GoodHackerStorage SendVehicleTo GoodHackerStorage 0 LoshHackerStorage

Output

Error: Price cannot be negative! Error: Invalid storage type! Registered GoodHackerStorage

Selected Semi

Error: HardDrive is out of stock! Error: Invalid source storage! Error: Invalid destination storage!

GoodHackerStorage: Storage worth: \$0.00

Input

RegisterStorage DistributionCenter AmazonDistribution

RegisterStorage Warehouse AmazonWarehouse

AddProduct HardDrive 80 AddProduct HardDrive 70 AddProduct HardDrive 120 AddProduct Gpu 800

SelectVehicle AmazonDistribution 0

LoadVehicle SolidStateDrive

LoadVehicle HardDrive Gpu HardDrive

SendVehicleTo AmazonDistribution 0 AmazonWarehouse

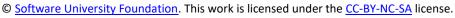
GetStorageStatus AmazonWarehouse UnloadVehicle AmazonWarehouse 3 GetStorageStatus AmazonWarehouse

END

Output

Registered AmazonDistribution





















Registered AmazonWarehouse

Added HardDrive to pool

Added HardDrive to pool

Added HardDrive to pool

Added Gpu to pool

Selected Van

Error: SolidStateDrive is out of stock!

Loaded 3/3 products into Van

Sent Van to AmazonWarehouse (slot 3)

Stock (0/10): []

Garage: [Semi|Semi|Semi|Van|empty|empty|empty|empty|empty|empty]

Unloaded 3/3 products at AmazonWarehouse Stock (2.7/10): [HardDrive (2), Gpu (1)]

Garage: [Semi|Semi|Semi|Van|empty|empty|empty|empty|empty|empty]

AmazonWarehouse:

Storage worth: \$990.00 AmazonDistribution: Storage worth: \$0.00

Task 4: Bonus (50 points)

Factories

You know that the keyword **new** is a bottleneck and we are trying to use it as little as possible. We even try to separate it in classes. These classes are called Factories and the naming convention for them is {TypeOfObject}Factory.

You need to implement three different factories, one for Products (ProductFactory), one for Storage (StorageFactory), and one for Vehicles (VehicleFactory). This is a design pattern and you can read more about it. Factory Pattern. The factories must contain a method

("CreateProduct/CreateStorage/CreateVehicle"), which instantiates objects of that type.

If you try to create a product/storage/vehicle with an invalid type, throw an InvalidOperationException with a message "Invalid product/storage/vehicle type!".

No static factories are allowed!



