ISIT 324 Homework 6 – Mocking a Used Car Lot v2.0

**30 Points Possible**

# Assignment Setup

For this assignment you’ve been given a .cs file named **CarsWithPizzazz** on the assignment page. Create a VS Class Library (.NET Framework) project and copy the code in CarsWithPizzazz.cs into the generated Class1.cs file. You may change the name of the Class1.cs file if you’d like.

Add a Unit Test Project (.NET Framework). All your tests will go into the UnitTest1.cs file (you’re free to rename it, though.)

When you’re done you will submit the .cs file that contains your unit tests.

# The Scenario

Otto’s Autos is a used car company in Rancho Cucamonga, CA. You’re working on a project to help automate inventory at Otto’s used car lot.

You have been given a .cs file named **CarsWithPizzazz** to act as the software under test**.** Your job is to test some methods that are intended to go against a database layer, but the database is not yet available. Thus, you must prepare a mock to mimic the database access behaviors, and you must write unit tests to validate that each method that relies on data from the database works.

## Notes on application structure

* In **CarsWithPizzazz.cs** you’ll find an interface named **IAutoDBAccess** that includes two methods: **LoadLot,** which is intended to load a collection of cars (instances of the **Auto** class) and **SaveLot,** which is intended to update the database based on changes made to the collection.[[1]](#footnote-2)
* There’s a class that implements that interface called **CarCollection** and you’ll note that the implementations of **LoadLot** and **SaveLot** therein are sorely wanting. The fact that they haven’t actually been written yet and need to be simulated is the premise of this exercise.
* **Auto** is a class that specifies the details of a particular car. The collection that **LoadLot** returns and that **SaveLot** consumes is a List of Autos.

## The AutoControl Class and Its Methods

Data access and manipulation is the province of the **AutoControl** class. It includes a number of methods. Some methods only read from the collection, but others update it:

* **Read** methods
  + **FindCar** – returns a single car based on VIN (Vehicle Identification Number)
  + **FindCarsByMake** – returns a list of cars of a specified Make
* **Update** methods
  + **AddCar** adds a car to the collection
  + **RemoveCar** removes a car from the collection

Here’s the thing: each of the **Read** methods begins with a call to **LoadLot()** to get the records from the database and load them into the collection. So do each of the **Update** methods, but they also conclude with a call to **SaveLot()** to cause any updates to be reflected in the DB. **SaveLot()** returns true when there are no problems encountered writing to the DB, but if it fails for any reason, it returns false.

Some of the methods throw application-specific exceptions. You’ll find their definitions towards the end of CarsWithPizzazz.cs. The validation rules appear as comments in the code. You can also infer them from the tests below.

# Testing CarsWithPizzazz

## The test data:

Use these values to simulate the cars in the database:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **VIN** | **Year** | **Make** | **Model** | **Location on Lot** |
| 01xxxxxxxxxxxxxxx | 2008 | Cadillac | CTS-V | A5 |
| 02xxxxxxxxxxxxxxx | 1964 | Dodge | Dart | F3 |
| 03xxxxxxxxxxxxxxx | 1963 | Cadillac | Fleetwood | A23 |
| 04xxxxxxxxxxxxxxx | 1995 | Hummer | H1 (Gas) | C7 |
| 05xxxxxxxxxxxxxxx | 1958 | Triumph | TR3 | A1 |
| 06xxxxxxxxxxxxxxx | 1968 | Triumph | TR5 | A2 |

## The tests:

Here are the tests you need to build:

* FindCar
  + Return instance of Auto when requested car is found.
  + Throw VINNotFoundException when the requested car is not found.
* FindCarsByMake
  + Return the correct number of instances (2) when looking for “Cadillac” as a car make.
  + Return zero when looking for “Audi.”
* AddCar
  + Return properly update collection when the add succeeds. To ensure that the collection is properly updated requires two checks:
    - The updated collection must have the correct count of items, and
    - The last item in the collection must have the VIN associated with the auto to add.
  + Throw DuplicateVINException if there’s already a car on the lot with the new auto’s VIN.
  + Throw DuplicateLocationException if there’s already car at the same spot on the lot as the new car.
  + Throw InvalidVINException when the VIN is not exactly 17 characters long (no joke…that’s the actual required length of a Vehicle Identification Number.)
* RemoveCar
  + Return collection with requested car removed car if the car was initially on the lot.
  + Return VINNotFoundException when the car to be removed is not on the lot.

**Important hint**

While most of the solution can be developed by following the pattern in the example program posted on the Test Doubles module page, there is at least one element that does not. To setup the fake version of **SaveLot()** you’ll need to figure out how to mock an incoming parameter. Check the references on the Test Doubles module page for help.

Might be a little tricky, but I have faith in you!

# For full credit:

* **Do not submit a zipped VS solution for this assignment.** Submit your answer as a single .cs file containing all your Unit Tests.
* **Do not, under any circumstances, change the code you’ve been given!** Not one line. Your Unit Test must run against the code in **CarsWithPizzazz.cs** as is**.** If you feel the desire to change the code, resist! If you run into an insurmountable problem and really, really, really think you need to change that code, reach out via the discussion (or, if not thus inclined, reach out to me via email.) Friends don’t let friends change the software under test.
* You must build the unit tests specified above for each of the methods. *Note: all of the methods work properly (at least to the best of my knowledge. No trickery here.)*
* Use Moq (from NuGet) to enable mocking.
* For the mimicked test data content, use the values in the preceding test data table.
* Use one test class per method, named properly (where “properly” means according to the standard I’ve been promoting.)
* Name each method properly (where “properly” means according to the standard I’ve been promoting.)
* Arrange your test methods using the “arrange, act, assert” pattern (including comments).
* Use the SUT alias for the “CarsWithPizzazz” namespace. Do not include a “using CarsWithPizzazz” directive.

1. The word “lot” is used because this application is for a used car lot, not because “lot” has any deep computer science meaning**.)**  [↑](#footnote-ref-2)