1. **Getting started**
   1. Open the exercise solution and temporary unload the ‘Tests’ project.
   2. Run the application and understands what’s happening.
   3. Load the test project.
   4. Install NUnit, the NUnit test adapter and Moq packages for the test project, using NuGet Package manager.
   5. Run the unit tests and see what happens. Explore the existing tests and their purpose.
2. **Write unit tests**
   1. Write a test to confirm that a car gets created without tires.
   2. Write a test that validates that the Repaint function changes the car color.
   3. Write a test to confirm that driving 2 miles increases the distance in kilometers with the appropriate amount.
3. **Reduce duplicate code**
   1. Investigate the occurrence of code duplication in the unit tests.
   2. Implement the SetUp/TearDown pattern to reduce code duplication.
   3. Think of both positive and negative implications of applying this pattern. (Will be discussed afterwards).
4. **Build a bad unit test**
   1. Implement a unit test which validates that the car speed is 0 km/h after starting a new car.
   2. Identify the problem(s) with this unit test
5. **Apply dependency injection to the Engine**
   1. Create an interface ‘IEngine’ and define the required methods in it.
   2. Make the Diesel Engine implement the IEngine interface though it’s base class.
   3. Make the Car use the IEngine interface and apply the dependency injection pattern using the constructors.
   4. Run the application and the unit tests to validate that functionality didn’t change or break.
6. **Refactoring internal code to use dependency injection**
   1. Find the mechanism inside the Car class which interferes with the isolation criteria for unit tests.
   2. Refactor this functionality to become an external dependency of the Car and apply the dependency injection pattern to it. Call it ‘FileDistanceStorage’.
   3. Run the application and the unit tests to validate that functionality didn’t change or break.
7. **Mocking part two**
   1. Make the car use an engine mock object, instead of a Diesel engine
   2. Run the unit test and observe the results. Identify the cause.
   3. Make the broken unit test work again by adding Engine mock behavior.
   4. Observe what mocking the engine did for the unit test duration time.
   5. Verify that the car throws a CarBrokenException if the engine throws a EngineFailureException while starting.
8. **Mocking part one**
   1. Mock the FileDistanceStorage object and make the car use it for the tests
   2. Write a test that the DistanceKm property of a car is always initialized with zero
   3. Write a test which validates that the car stores the new distance value after driving 2 kilometers