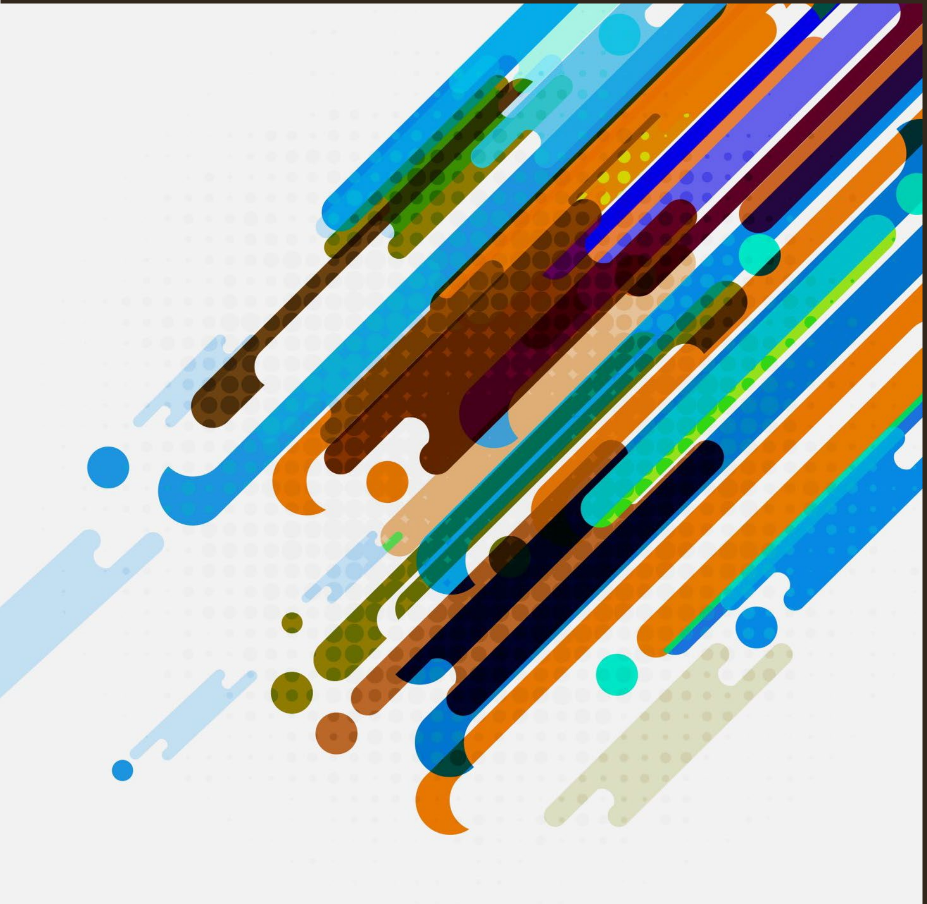
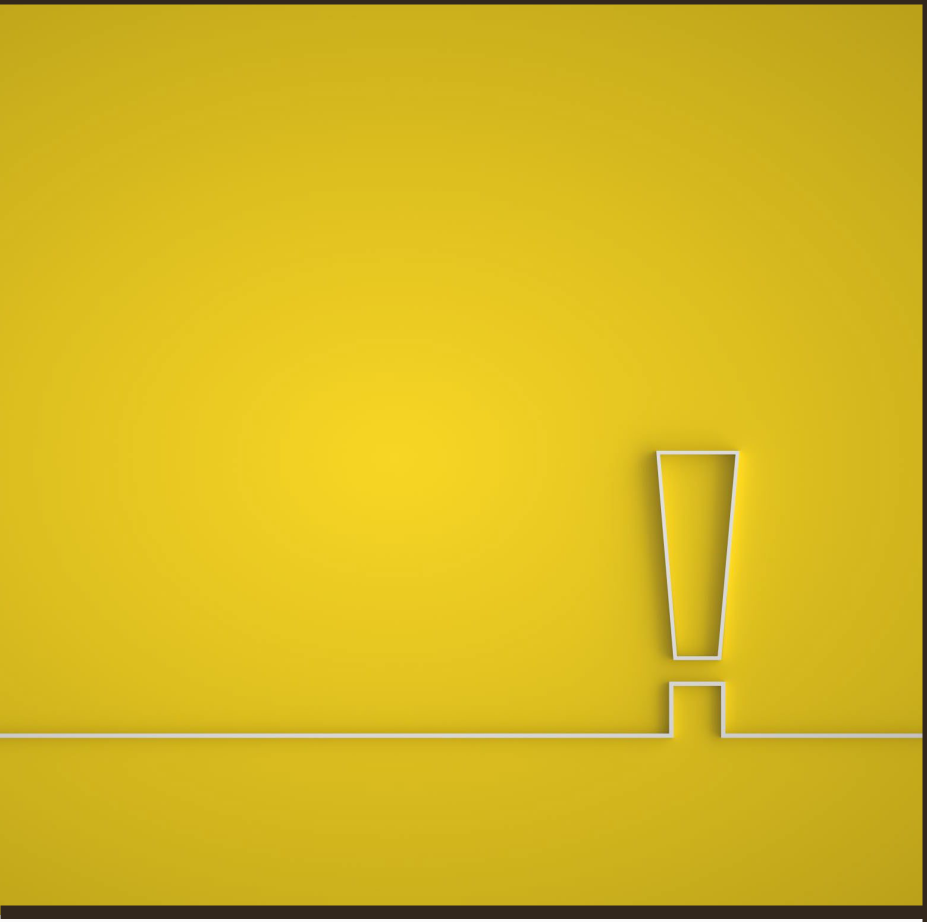


SESSION 5: ONLINE TRANSACTION PROCESSING (OLTP)

Powering Real-Time Business Operations and Data Entry!



Session's Objective

- Review OLTP

- Setup LINQPad for Entity Framework Core

- Customer Add (AddEditCustomer)

Define the rules.

Create “Unit Tests” based on the rules using Arrange, Act, and Assert patterns.

Create the method for returning a customer (GetCustomer).

Refactor using Arrange, Act, and Assert patterns.

- Customer Edit (AddEditCustomer)

Define the rules.

Create “Unit Tests” based on the rules using Arrange, Act, and Assert patterns.

Refactor using Arrange, Act, and Assert patterns.

Overview

OLTP is a way of making sure that a series of operations on a database or an application are done correctly and completely, or not at all. It also ensures that the data remains consistent and reliable, even if there are problems like system failures or concurrent access. Transaction management involves creating, coordinating, and executing transactions and recovering from any errors or interruptions.

Characteristics

- **High Concurrency**: OLTP systems are designed to support simultaneous multiple users without system delays.
- **Fast Query Performance**: These systems prioritize quick, real-time query responses.
- **Short, Simple Transactions**: Transactions in OLTP systems often involve reading and updating a few records.

Database Design

Normalized Structure: OLTP databases tend to be highly normalized, which means data redundancy is minimized. This helps in ensuring data integrity.

Primary and Foreign Keys: These are used extensively to maintain relationships between tables and to ensure data consistency.

Transaction Management

•**ACID Properties**: Key characteristics ensuring reliability and consistency in Database Transactions:

•**Atomicity**: All or none of a transaction's operations are performed.

- E.g., Money transfers update both accounts or neither.

•**Consistency**: The database remains correct and maintains integrity constraints.

- E.g., Adding records also updates indexes and keys.

•**Isolation**: Concurrent transactions don't interfere or cause inconsistency.

- E.g., Two transactions updating a record wait in turn.

•**Durability**: Once done, transaction changes are saved and survive system failures.

- For example, updated balances stay saved even if power is lost.

Transaction Management (Continue)

- **Concurrency Control**: Mechanisms like locks and timestamps to ensure multiple transactions can happen simultaneously without conflicts.

- **Commit & Rollback**: Mechanisms to finalize transactions or revert them if issues arise.

Applications and Use Cases

- **E-commerce Systems**: Online shopping carts, stock trades, or any system that requires real-time user interaction.
- **CRM and ERP Systems**: Real-time systems that businesses use for daily operations.
- **Retail Sales**: Point-of-sale systems in retail environments.

OLTP Operations Insert (Addition of new data)

Data Addition: Entering new records into the database.

Primary Key: A unique identifier, often automatically generated, to differentiate new records.

Data Validation: Ensuring that the new data adheres to the database's constraints and rules.

Referential Integrity: Making sure relationships between tables remain consistent when adding new data.

Trigger Events: Actions that can automatically happen post-insert, like sending a confirmation email after account creation.

Concurrency: Handling multiple users trying to insert data simultaneously without conflicts.

OLTP Operations

Update (Modification of existing data)

Data Alteration: Modifying the content of one or more existing records.

Record Locking: Temporarily locking a record being updated to prevent data inconsistency from concurrent edits.

Versioning: Keeping track of changes made to records over time.

Data Validation: Re-checking constraints and rules when updating data.

Maintain Relationships: Ensuring foreign keys and relations remain valid post-update.

Trigger Events: Actions that might occur post-update, like alerting a user of changes to their profile.

OLTP Operations

Delete (Removal of data)

•**Data Removal**: Permanently or temporarily removing one or more records from the database.

•**Referential Integrity**: Ensuring that no orphaned records exist post-deletion. This may require cascading deletes or nullifying related records.

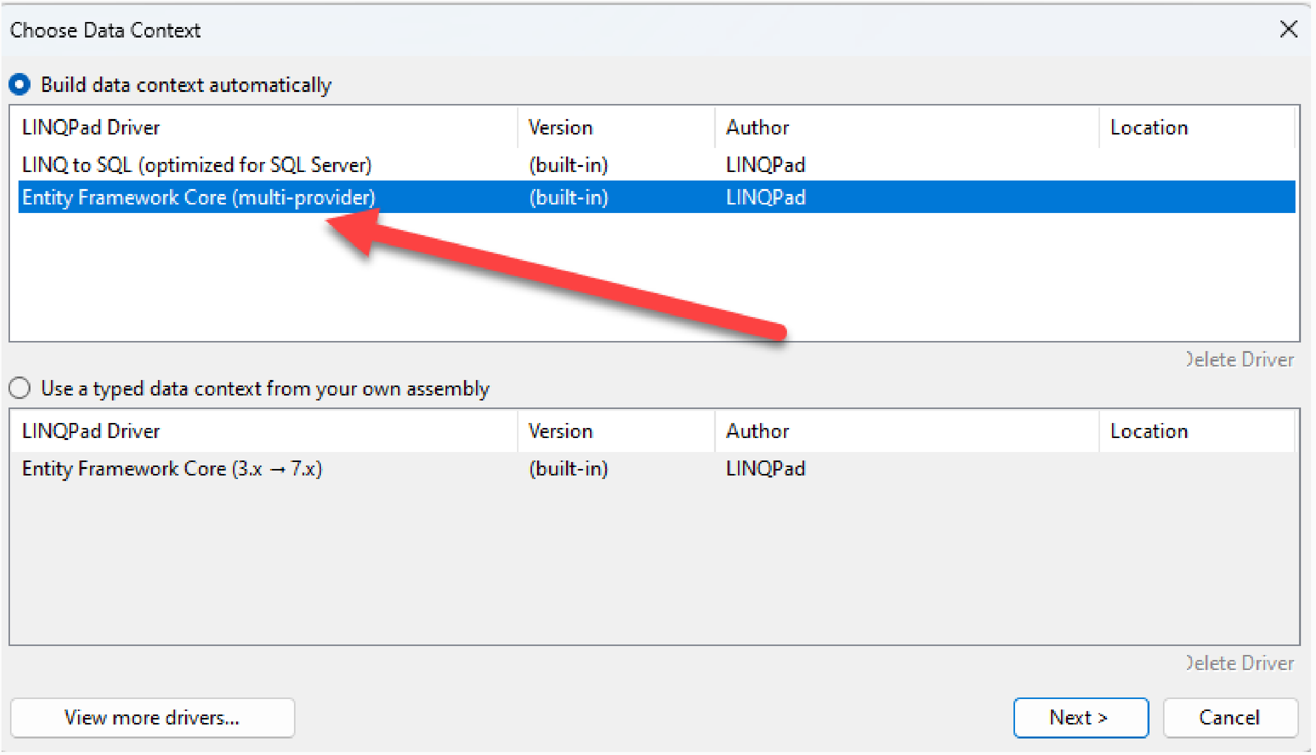
•**Archiving**: Instead of fully deleting, moving data to an archive for historical purposes.

•**Safe Deletion**: Marking records as 'deleted' without removing them from the database, to allow potential recovery.

•**Impacts on Indexes**: Deleting records might require updates to database indexes to maintain performance.

•**Trigger Events**: Actions post-deletion, such as sending a confirmation of account deletion.

CREATING LINQPAD ENTITY FRAMEWORK CONNECTION



Choose Data Context

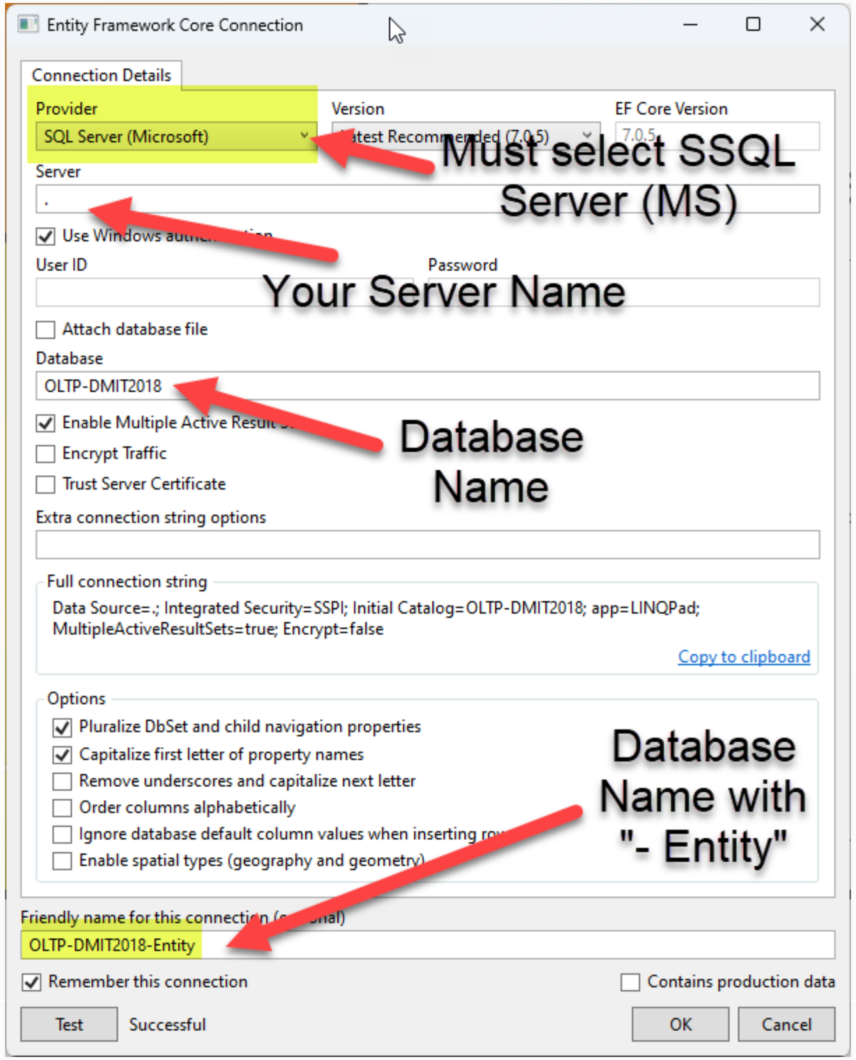
● Build data context automatically

LINQPad Driver	Version	Author	Location
LINQ to SQL (optimized for SQL Server)	(built-in)	LINQPad	
Entity Framework Core (multi-provider)	(built-in)	LINQPad	

○ Use a typed data context from your own assembly

LINQPad Driver	Version	Author	Location
Entity Framework Core (3.x → 7.x)	(built-in)	LINQPad	

CREATING LINQPAD ENTITY FRAMEWORK CONNECTION



Provider: SQL Server (Microsoft)

Version: Latest Recommended (7.0.5)

EF Core Version: 7.0.5

Server: .

☑ Use Windows authentication

User ID

Password

☐ Attach database file

Database: OLTP-DMIT2018

☑ Enable Multiple Active Result Sets

☐ Encrypt Traffic

☐ Trust Server Certificate

Extra connection string options

Full connection string

Data Source=.; Integrated Security=SSPI; Initial Catalog=OLTP-DMIT2018; app=LINQPad; MultipleActiveResultSets=true; Encrypt=false

Options

☑ Pluralize DbSet and child navigation properties

☑ Capitalize first letter of property names

☐ Remove underscores and capitalize next letter

☐ Order columns alphabetically

☐ Ignore database default column values when inserting rows

☐ Enable spatial types (geography and geometry)

Friendly name for this connection (optional): OLTP-DMIT2018-Entity

☑ Remember this connection

☐ Contains production data

Successful

Must select SSQL Server (MS)

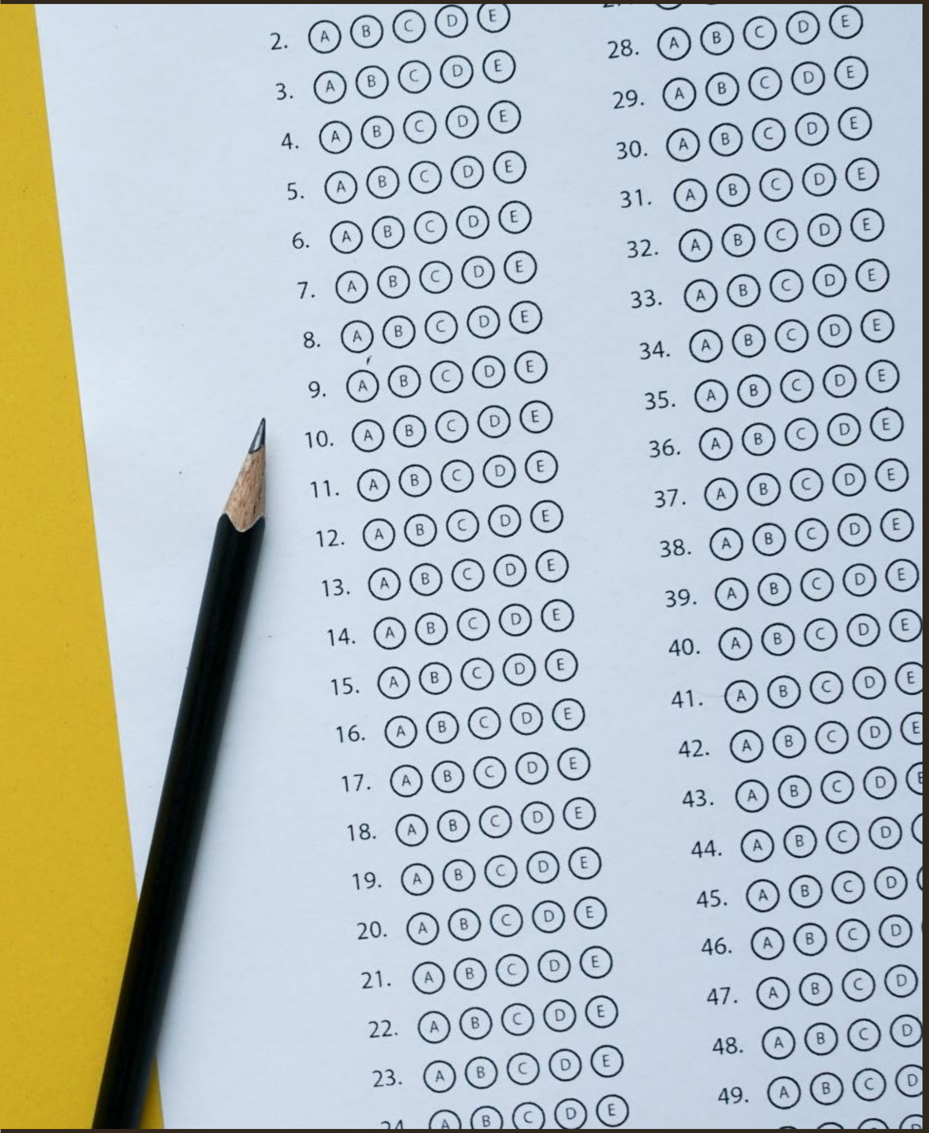
Your Server Name

Database Name

Database Name with "- Entity"

Customer Add & Edit – Defining the Rules!

What are our rules?



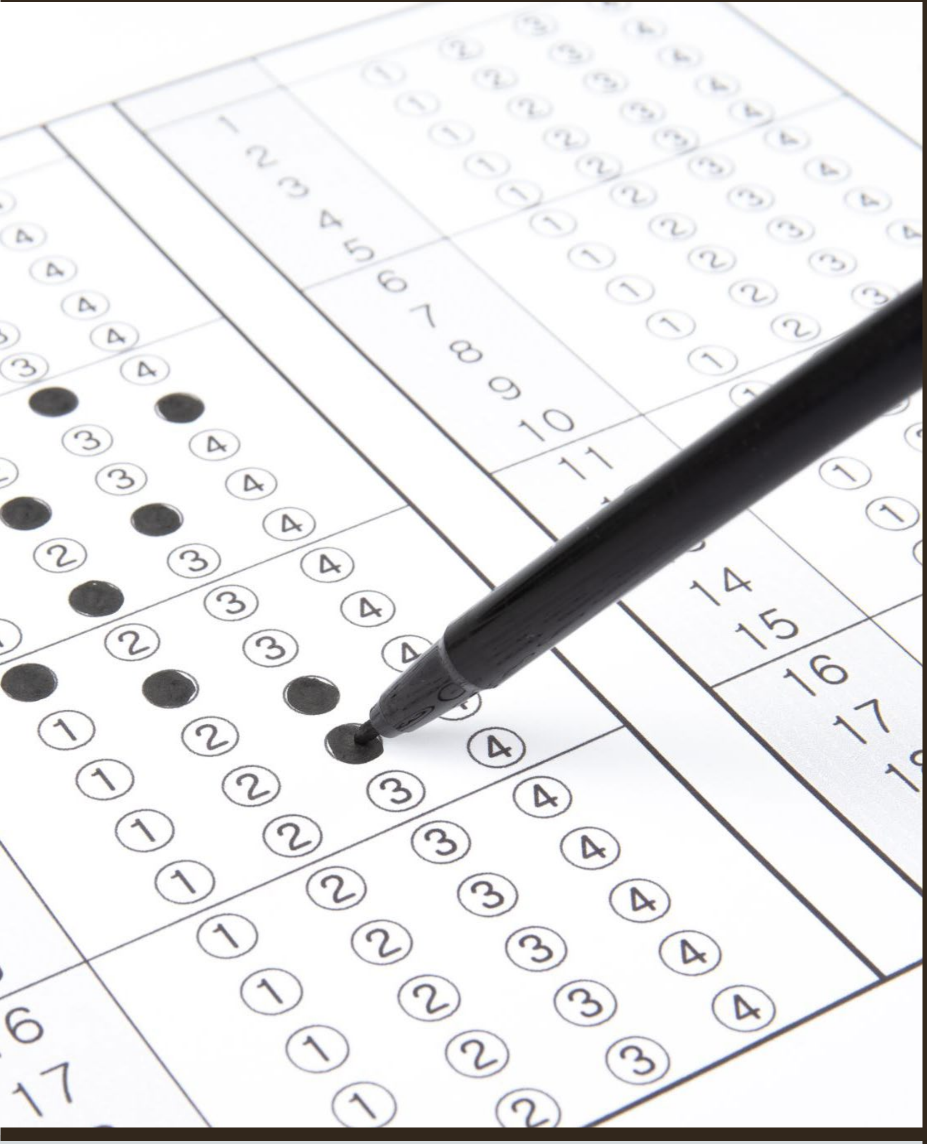
Business Rules

Add:

- **Cannot add a customer if their phone number already exists in the database.**

Add/Edit:

- **All fields except for “Address2” are required.**



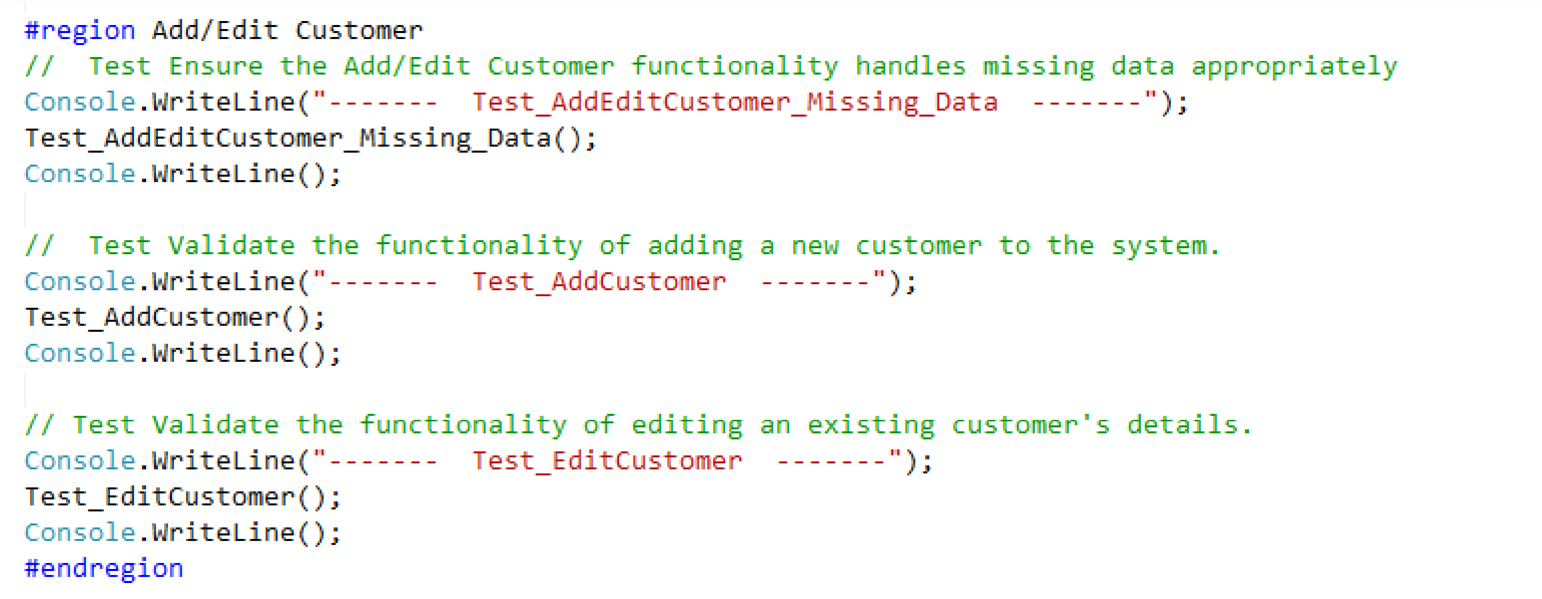
Creating Unit Test Placeholders for Add/Edit Customers

- Test_AddEditCustomer_Missing_Data
- Test_AddCustomer
- Test_EditCustomer

- NOTE: Add/edit customer test must retrieve a single record for comparison. We will be creating the GetCustomer method in a few minutes.

CREATING UNIT TEST PLACEHOLDERS FOR ADD/EDIT CUSTOMERS

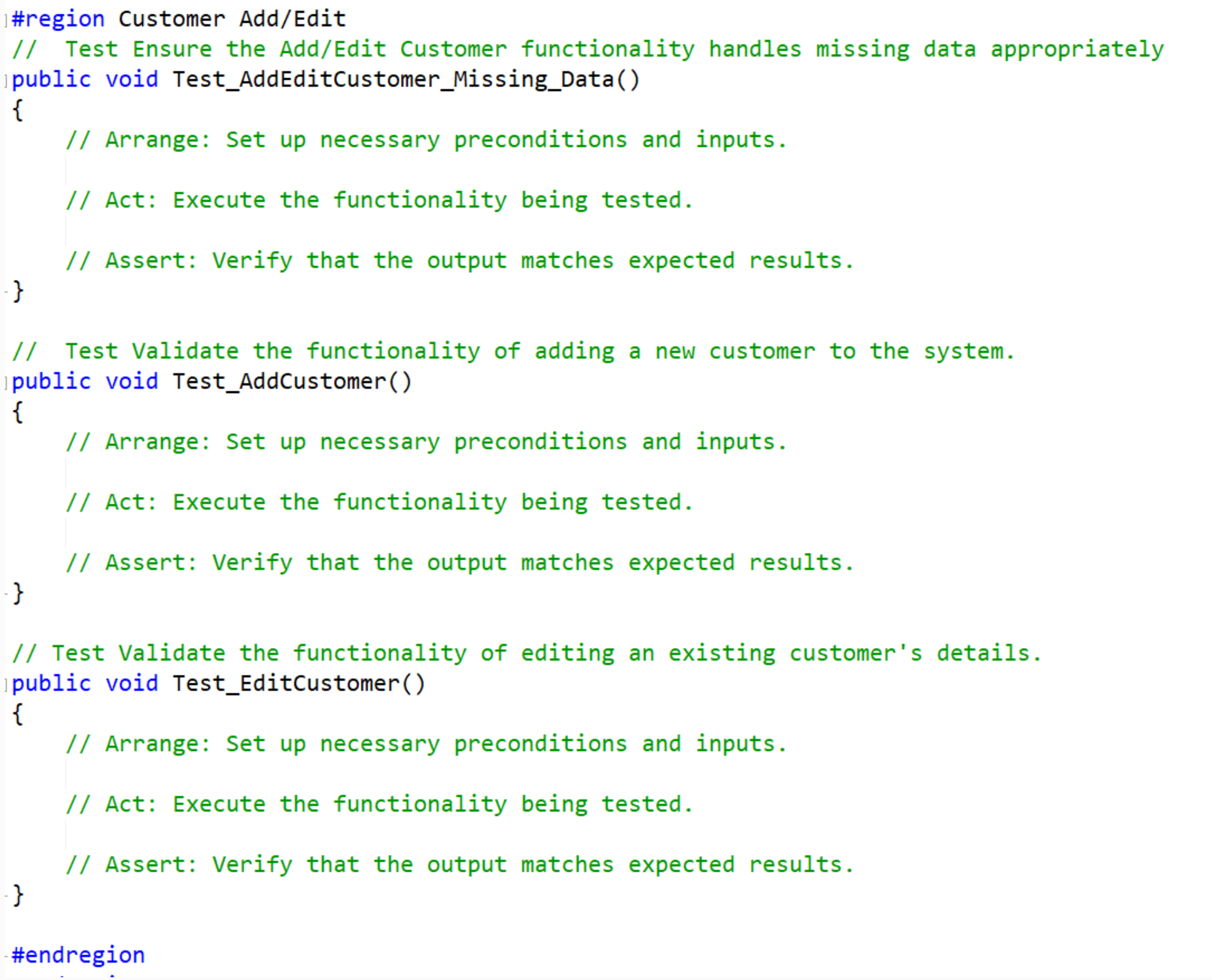
Copy Code to Students



```
#region Add/Edit Customer
//  Test Ensure the Add/Edit Customer functionality handles missing data appropriately
Console.WriteLine("-------  Test_AddEditCustomer_Missing_Data  -------");
Test_AddEditCustomer_Missing_Data();
Console.WriteLine();

//  Test Validate the functionality of adding a new customer to the system.
Console.WriteLine("-------  Test_AddCustomer  -------");
Test_AddCustomer();
Console.WriteLine();

// Test Validate the functionality of editing an existing customer's details.
Console.WriteLine("-------  Test_EditCustomer  -------");
Test_EditCustomer();
Console.WriteLine();
#endregion
```

UPDATE CUSTOMERTEST.LINQ FILE

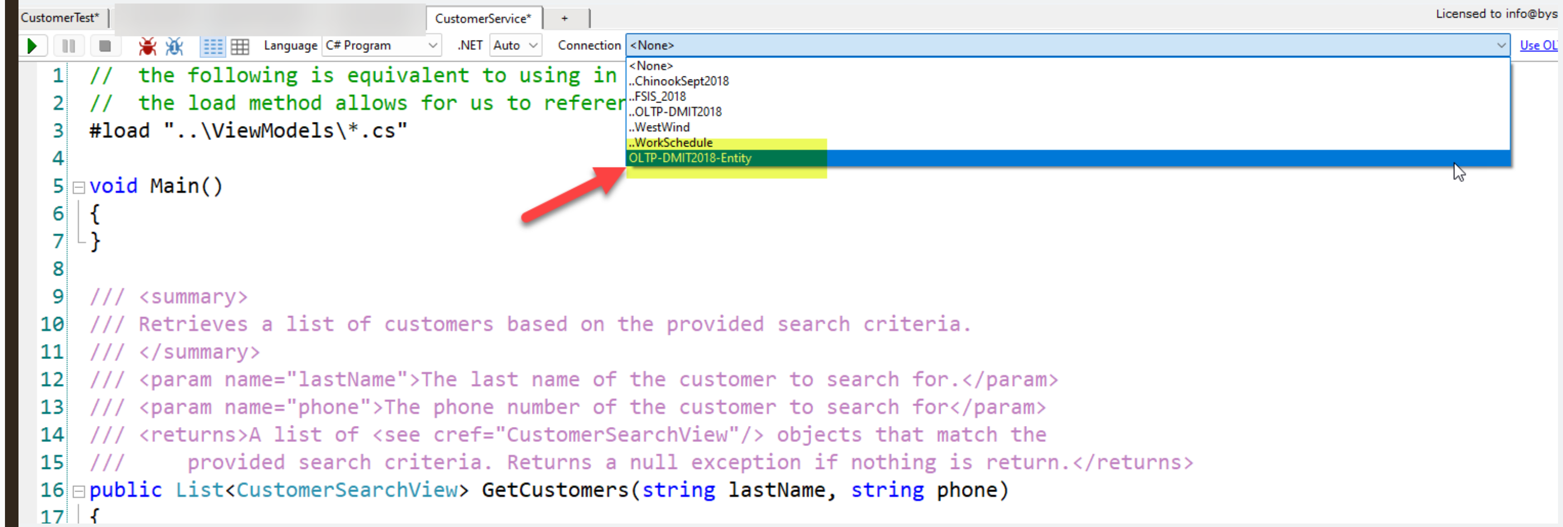
Copy Code to Students



```
#region Customer Add/Edit
//  Test Ensure the Add/Edit Customer functionality handles missing data appropriately
public void Test_AddEditCustomer_Missing_Data()
{
    // Arrange: Set up necessary preconditions and inputs.

    // Act: Execute the functionality being tested.

    // Assert: Verify that the output matches expected results.
}

//  Test Validate the functionality of adding a new customer to the system.
public void Test_AddCustomer()
{
    // Arrange: Set up necessary preconditions and inputs.

    // Act: Execute the functionality being tested.

    // Assert: Verify that the output matches expected results.
}

// Test Validate the functionality of editing an existing customer's details.
public void Test_EditCustomer()
{
    // Arrange: Set up necessary preconditions and inputs.

    // Act: Execute the functionality being tested.

    // Assert: Verify that the output matches expected results.
}

#endregion
```

Update Customer Service

Update the connection to “OLTP-DMIT2018-Entity”



Connection <None>

<None>
..ChinookSept2018
..FSIS_2018
..OLTP-DMIT2018
..WestWind
..WorkSchedule
OLTP-DMIT2018-Entity

```
//  the following is equivalent to using in
//  the load method allows for us to referer
#load "..\ViewModels\*.cs"

void Main()
{
}

/// <summary>
/// Retrieves a list of customers based on the provided search criteria.
/// </summary>
/// <param name="lastName">The last name of the customer to search for.</param>
/// <param name="phone">The phone number of the customer to search for</param>
/// <returns>A list of <see cref="CustomerSearchView"/> objects that match the
///     provided search criteria. Returns a null exception if nothing is return.</returns>
public List<CustomerSearchView> GetCustomers(string lastName, string phone)
{
```

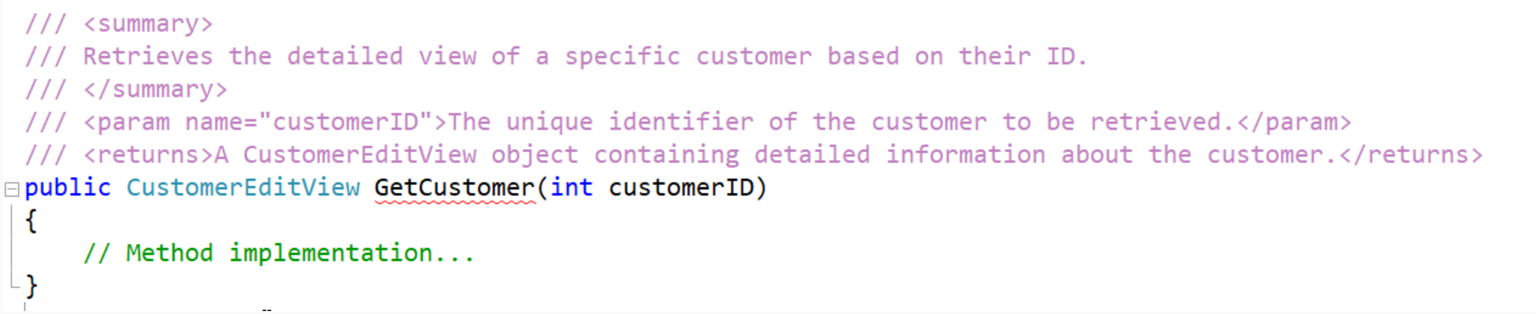
Add Method GetCustomer (int customerID)

Business Rule:

- Customer ID must be valid.
- Customer must exist in the database.

- NOTE: We will not be creating a Unit Test for this method due to the time limit.

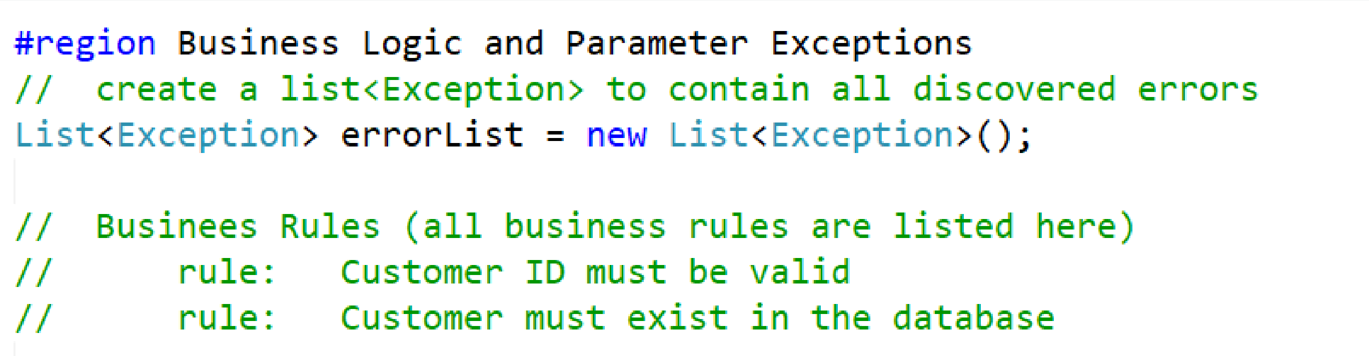
GET CUSTOMER METHOD



```
/// <summary>
/// Retrieves the detailed view of a specific customer based on their ID.
/// </summary>
/// <param name="customerID">The unique identifier of the customer to be retrieved.</param>
/// <returns>A CustomerEditView object containing detailed information about the customer.</returns>
public CustomerEditView GetCustomer(int customerID)
{
    // Method implementation...
}
```

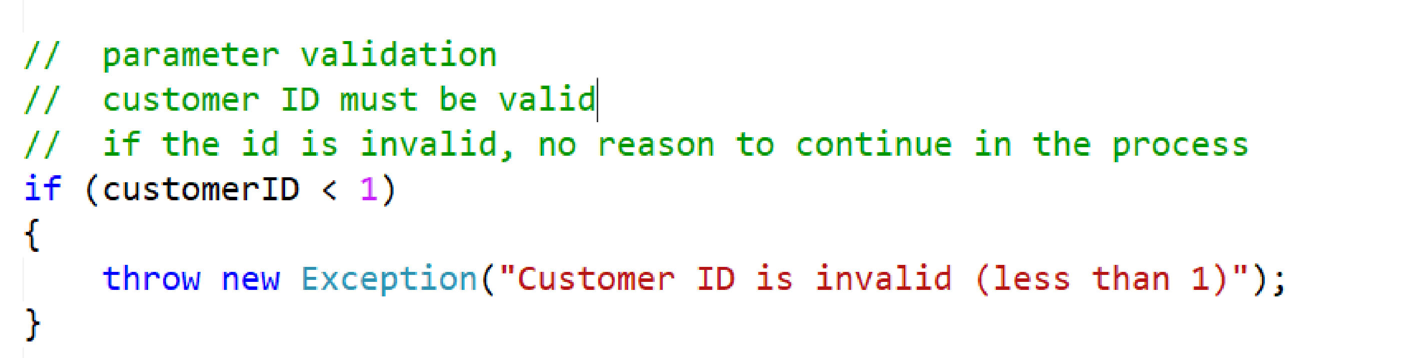
Build the Skeleton of our GetCustomers Method

Stub out the “Business Logic and Parameter Exceptions” area



```
#region Business Logic and Parameter Exceptions
//  create a list<Exception> to contain all discovered errors
List<Exception> errorList = new List<Exception>();

//  Businees Rules (all business rules are listed here)
//      rule:   Customer ID must be valid
//      rule:   Customer must exist in the database
```

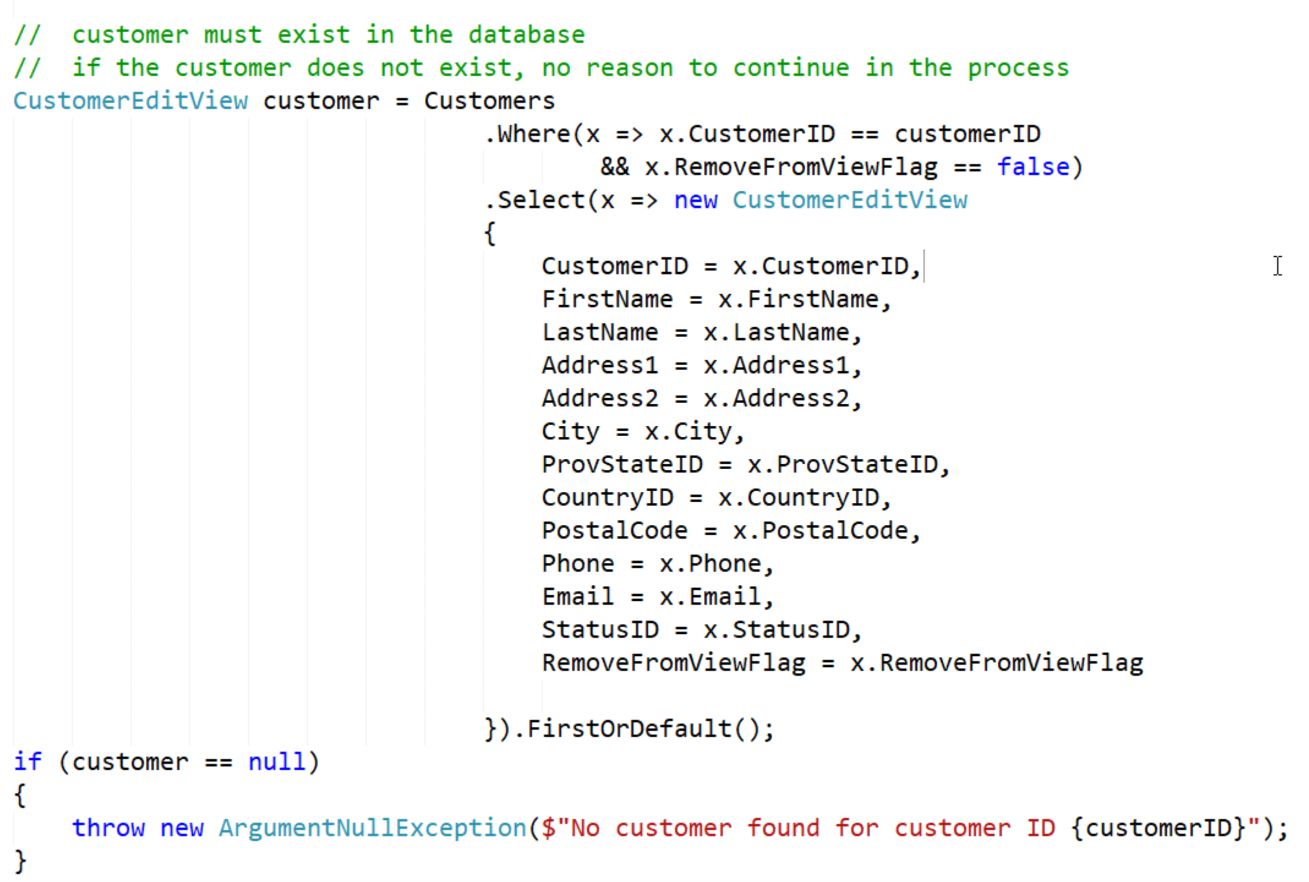
ADD “CUSTOMER ID MUST BE VALID” RULE



```
//  parameter validation
//  customer ID must be valid
//  if the id is invalid, no reason to continue in the process
if (customerID < 1)
{
    throw new Exception("Customer ID is invalid (less than 1)");
}
```

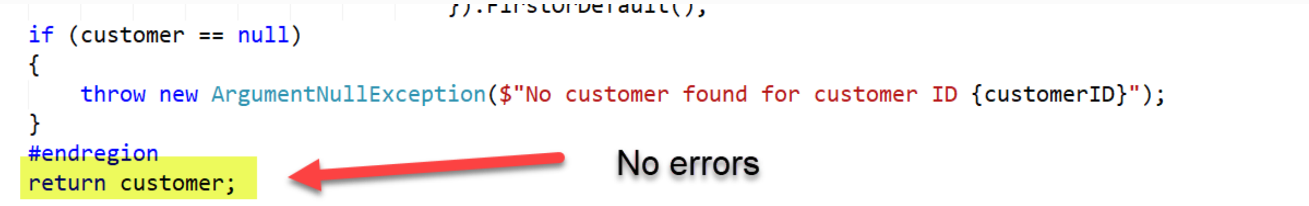
ADD “CUSTOMER MUST EXIST IN THE DATABASE” RULE

Copy Code to Students



```
//  customer must exist in the database
//  if the customer does not exist, no reason to continue in the process
CustomerEditView customer = Customers
                                .Where(x => x.CustomerID == customerID
                                        && x.RemoveFromViewFlag == false)
                                .Select(x => new CustomerEditView
                                {
                                    CustomerID = x.CustomerID,
                                    FirstName = x.FirstName,
                                    LastName = x.LastName,
                                    Address1 = x.Address1,
                                    Address2 = x.Address2,
                                    City = x.City,
                                    ProvStateID = x.ProvStateID,
                                    CountryID = x.CountryID,
                                    PostalCode = x.PostalCode,
                                    Phone = x.Phone,
                                    Email = x.Email,
                                    StatusID = x.StatusID,
                                    RemoveFromViewFlag = x.RemoveFromViewFlag

                                }).FirstOrDefault();
if (customer == null)
{
    throw new ArgumentNullException($"No customer found for customer ID {customerID}");
}
```

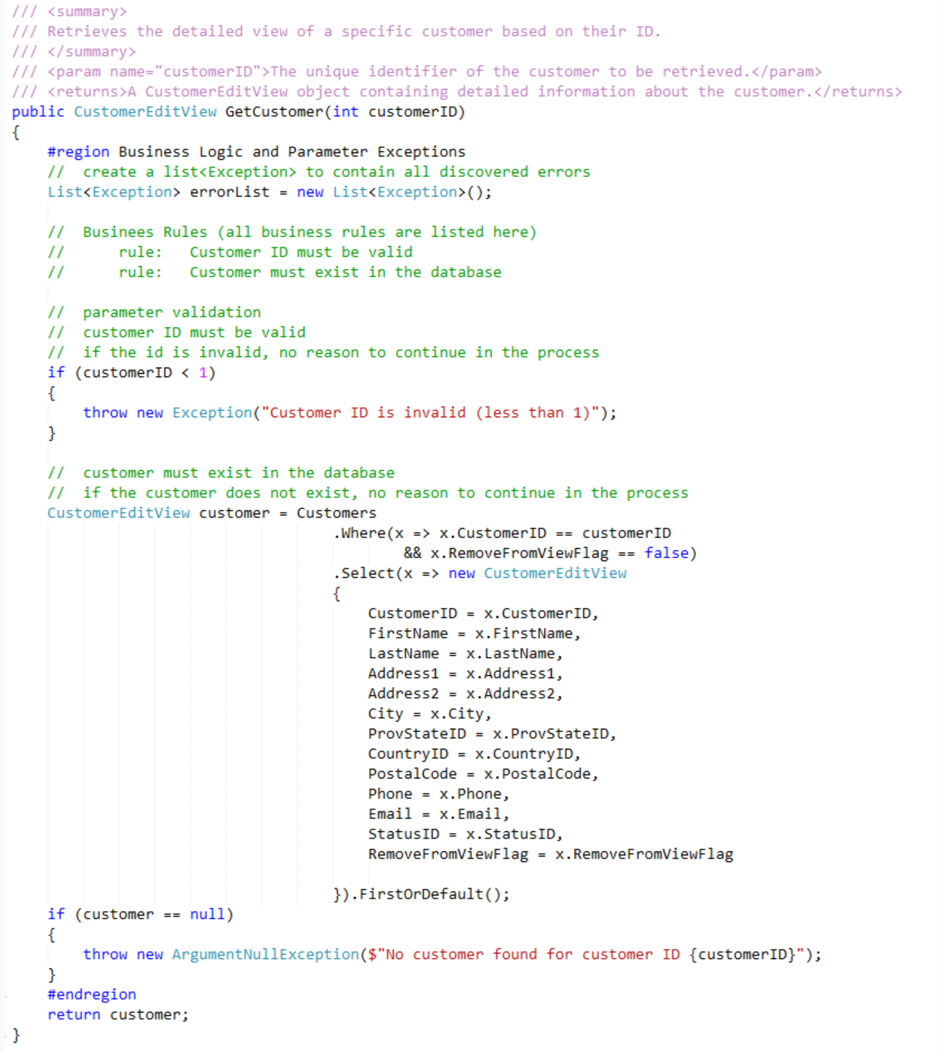
RETURN VALID CUSTOMER



```
                        }).FirstOrDefault();
if (customer == null)
{
    throw new ArgumentNullException($"No customer found for customer ID {customerID}");
}
#endregion
return customer;
```

No errors

COMPLETED METHOD



```
/// <summary>
/// Retrieves the detailed view of a specific customer based on their ID.
/// </summary>
/// <param name="customerID">The unique identifier of the customer to be retrieved.</param>
/// <returns>A CustomerEditView object containing detailed information about the customer.</returns>
public CustomerEditView GetCustomer(int customerID)
{
    #region Business Logic and Parameter Exceptions
    //  create a list<Exception> to contain all discovered errors
    List<Exception> errorList = new List<Exception>();

    //  Businees Rules (all business rules are listed here)
    //      rule:   Customer ID must be valid
    //      rule:   Customer must exist in the database

    //  parameter validation
    //  customer ID must be valid
    //  if the id is invalid, no reason to continue in the process
    if (customerID < 1)
    {
        throw new Exception("Customer ID is invalid (less than 1)");
    }

    //  customer must exist in the database
    //  if the customer does not exist, no reason to continue in the process
    CustomerEditView customer = Customers
                                    .Where(x => x.CustomerID == customerID
                                            && x.RemoveFromViewFlag == false)
                                    .Select(x => new CustomerEditView
                                    {
                                        CustomerID = x.CustomerID,
                                        FirstName = x.FirstName,
                                        LastName = x.LastName,
                                        Address1 = x.Address1,
                                        Address2 = x.Address2,
                                        City = x.City,
                                        ProvStateID = x.ProvStateID,
                                        CountryID = x.CountryID,
                                        PostalCode = x.PostalCode,
                                        Phone = x.Phone,
                                        Email = x.Email,
                                        StatusID = x.StatusID,
                                        RemoveFromViewFlag = x.RemoveFromViewFlag

                                    }).FirstOrDefault();
    if (customer == null)
    {
        throw new ArgumentNullException($"No customer found for customer ID {customerID}");
    }
    #endregion
    return customer;
}
```

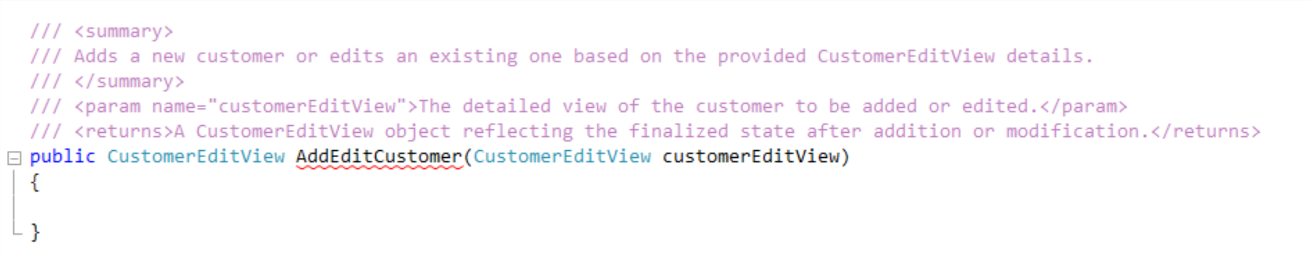
Add Method AddEditCustomer (CustomerEditView customerEditView)

When we used this method, the only difference between the add and edit functions is that the add functionality is triggered when the customer ID is equal to zero.

Business Rule:

- Excluding the customer ID and address 2, all fields must be valid.
- Cannot add a customer if their phone number already exists in the database.

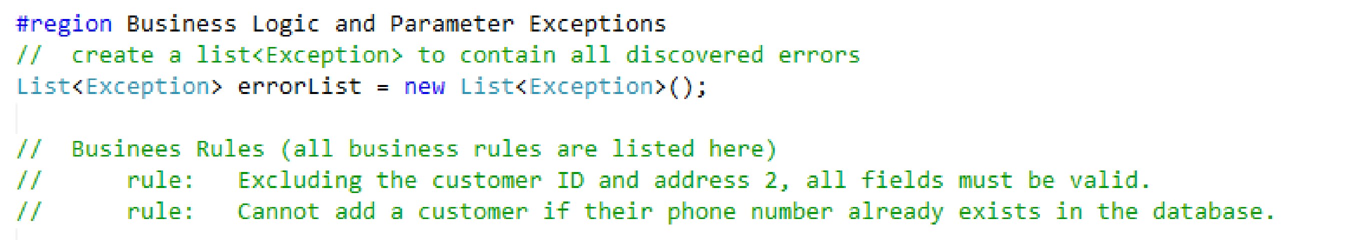
ADD/EDIT CUSTOMER METHOD



```
/// <summary>
/// Adds a new customer or edits an existing one based on the provided CustomerEditView details.
/// </summary>
/// <param name="customerEditView">The detailed view of the customer to be added or edited.</param>
/// <returns>A CustomerEditView object reflecting the finalized state after addition or modification.</returns>
public CustomerEditView AddEditCustomer(CustomerEditView customerEditView)
{

}
```

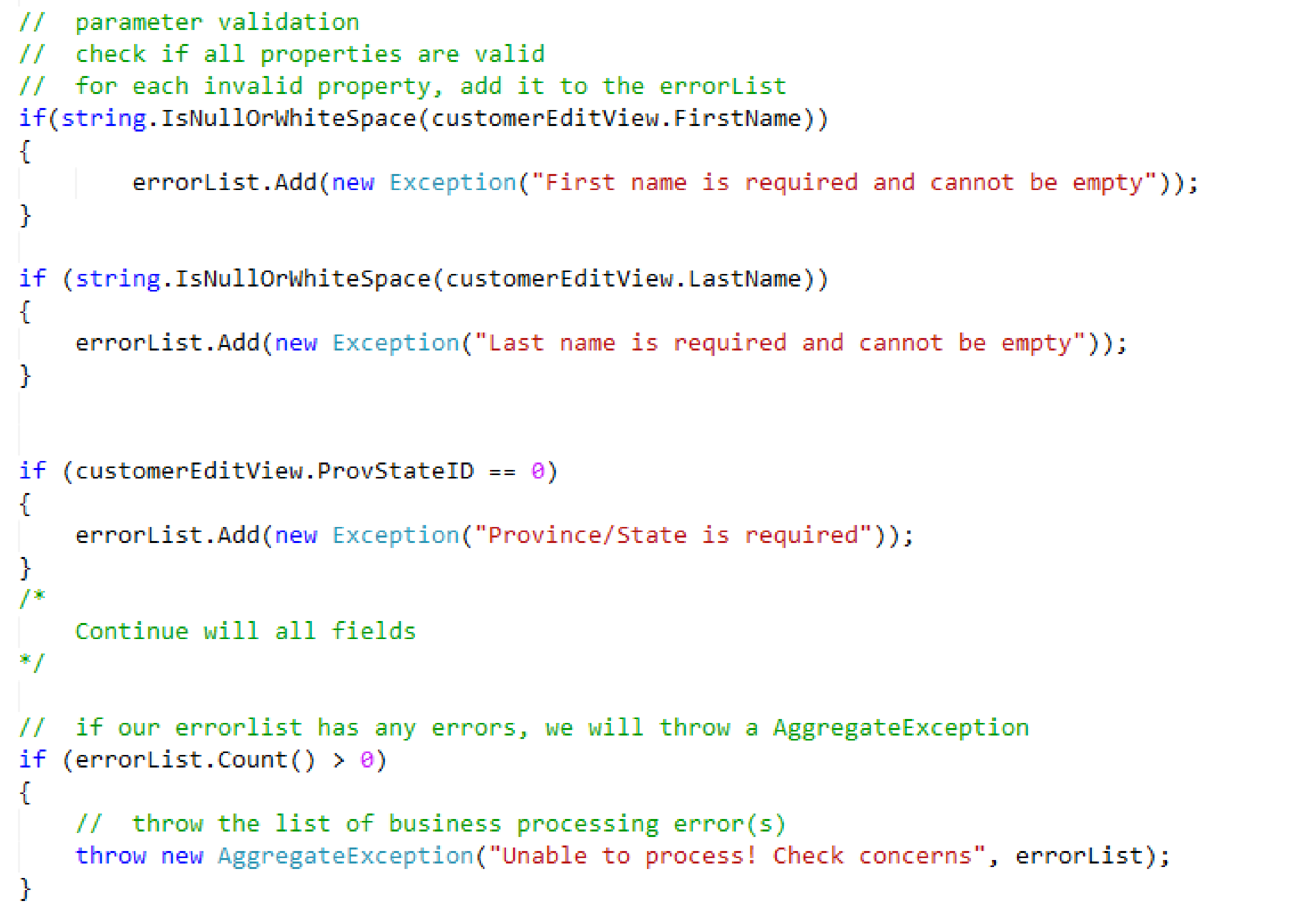
Build the Skeleton of our Add Edit Customer Method

Stub out the “Business Logic and Parameter Exceptions” area



```
#region Business Logic and Parameter Exceptions
//  create a list<Exception> to contain all discovered errors
List<Exception> errorList = new List<Exception>();

//  Businees Rules (all business rules are listed here)
//      rule:   Excluding the customer ID and address 2, all fields must be valid.
//      rule:   Cannot add a customer if their phone number already exists in the database.
```

Parameter Validation – Properties are Valid (Long Way)

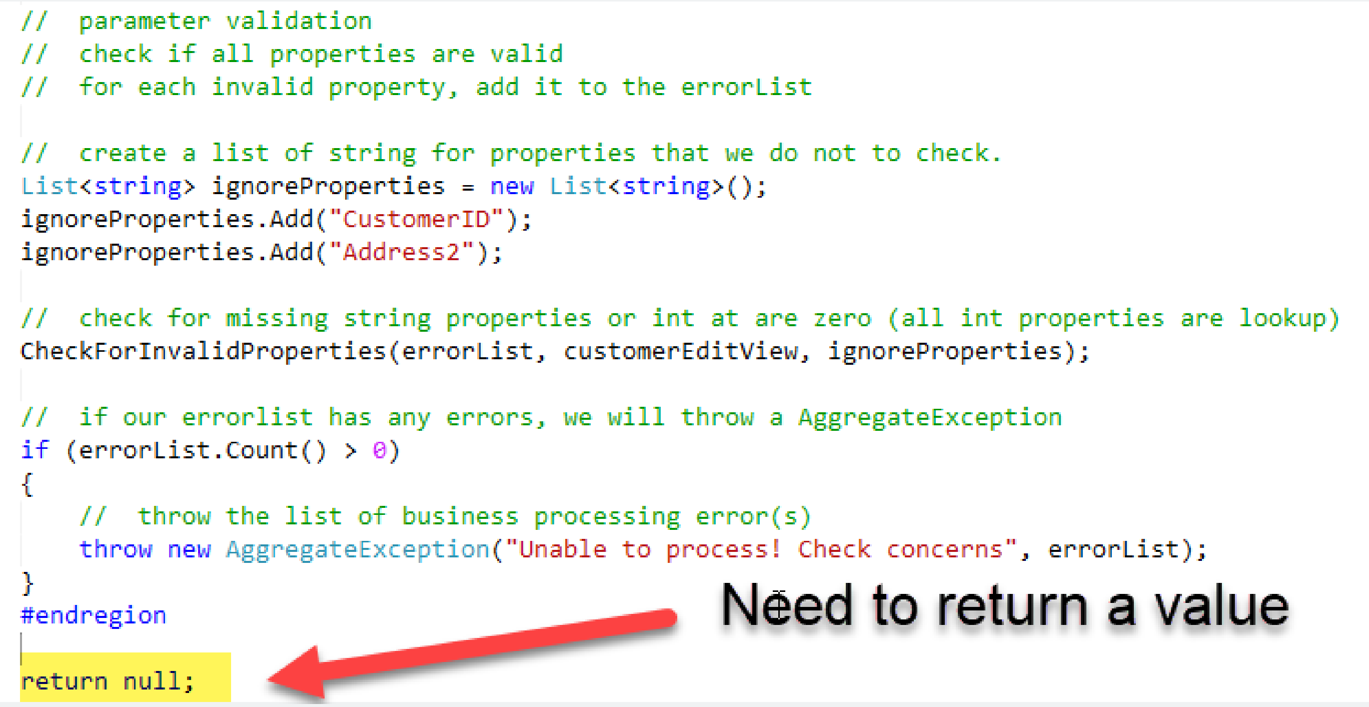


```
//  parameter validation
//  check if all properties are valid
//  for each invalid property, add it to the errorList
if(string.IsNullOrWhiteSpace(customerEditView.FirstName))
{
        errorList.Add(new Exception("First name is required and cannot be empty"));
}

if (string.IsNullOrWhiteSpace(customerEditView.LastName))
{
    errorList.Add(new Exception("Last name is required and cannot be empty"));
}


if (customerEditView.ProvStateID == 0)
{
    errorList.Add(new Exception("Province/State is required"));
}
/*
    Continue will all fields
*/

//  if our errorlist has any errors, we will throw a AggregateException
if (errorList.Count() > 0)
{
    //  throw the list of business processing error(s)
    throw new AggregateException("Unable to process! Check concerns", errorList);
}
```

Parameter Validation – Properties are Valid (Better Way)

Pass the customer view to a method that check each property to see if it is valid.

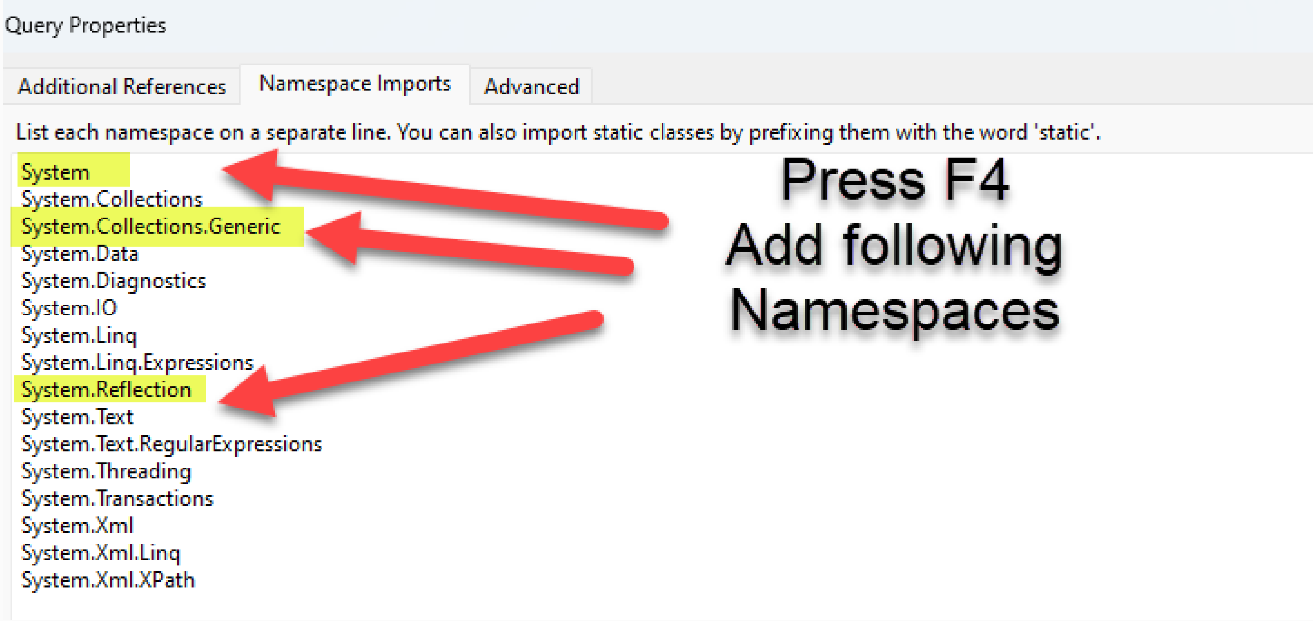
Copy Code to Students



```
//  parameter validation
//  check if all properties are valid
//  for each invalid property, add it to the errorList

//  create a list of string for properties that we do not to check.
List<string> ignoreProperties = new List<string>();
ignoreProperties.Add("CustomerID");
ignoreProperties.Add("Address2");

//  check for missing string properties or int at are zero (all int properties are lookup)
CheckForInvalidProperties(errorList, customerEditView, ignoreProperties);

//  if our errorlist has any errors, we will throw a AggregateException
if (errorList.Count() > 0)
{
    //  throw the list of business processing error(s)
    throw new AggregateException("Unable to process! Check concerns", errorList);
}
#endregion

return null;
```

Need to return a value

ADD NAMESPACES FOR CHECK FOR INVALID PROPERTIES METHOD



Query Properties

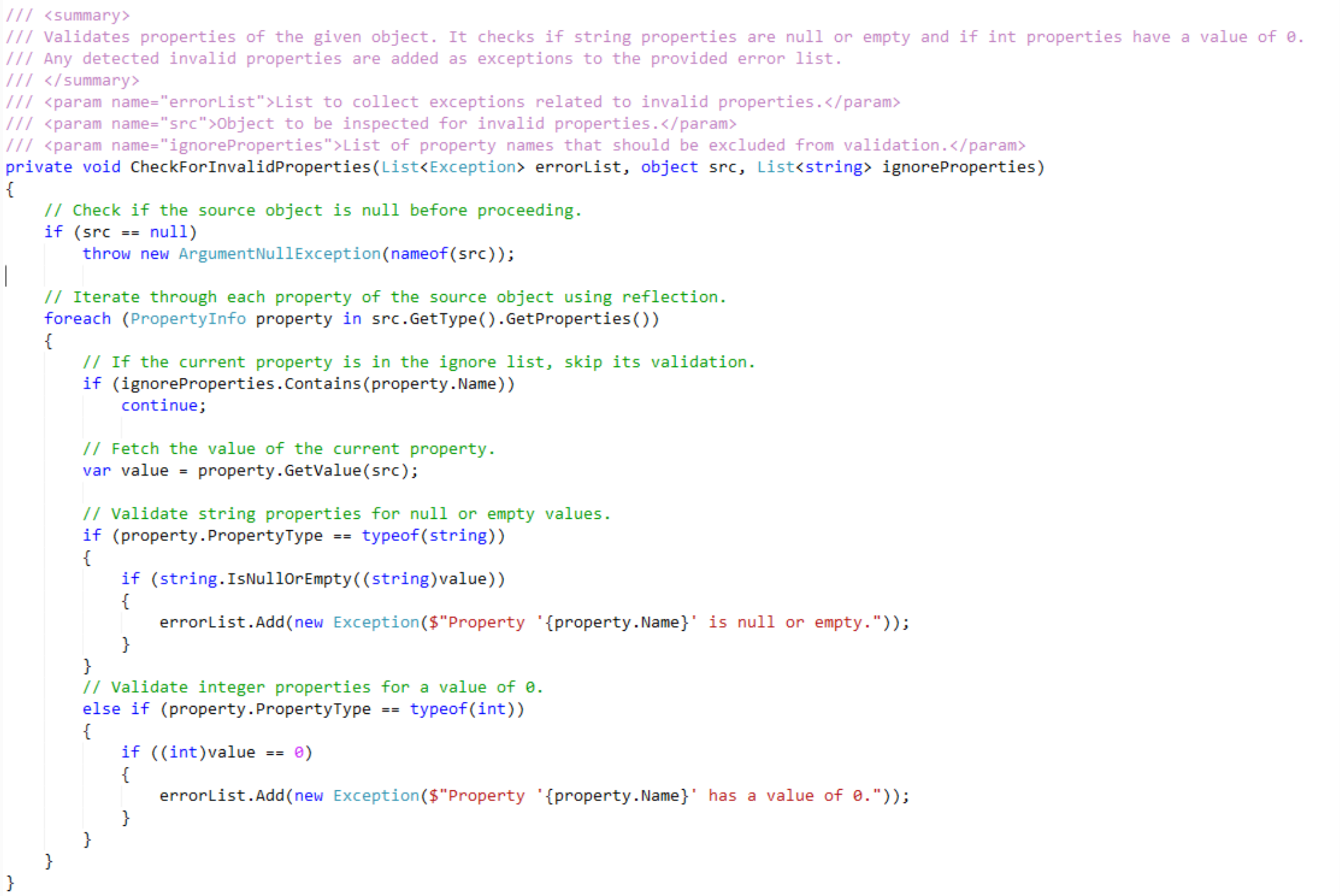
List each namespace on a separate line. You can also import static classes by prefixing them with the word 'static'.

```
System
System.Collections
System.Collections.Generic
System.Data
System.Diagnostics
System.IO
System.Linq
System.Linq.Expressions
System.Reflection
System.Text
System.Text.RegularExpressions
System.Threading
System.Transactions
System.Xml
System.Xml.Linq
System.Xml.XPath
```

Press F4
Add following
Namespaces

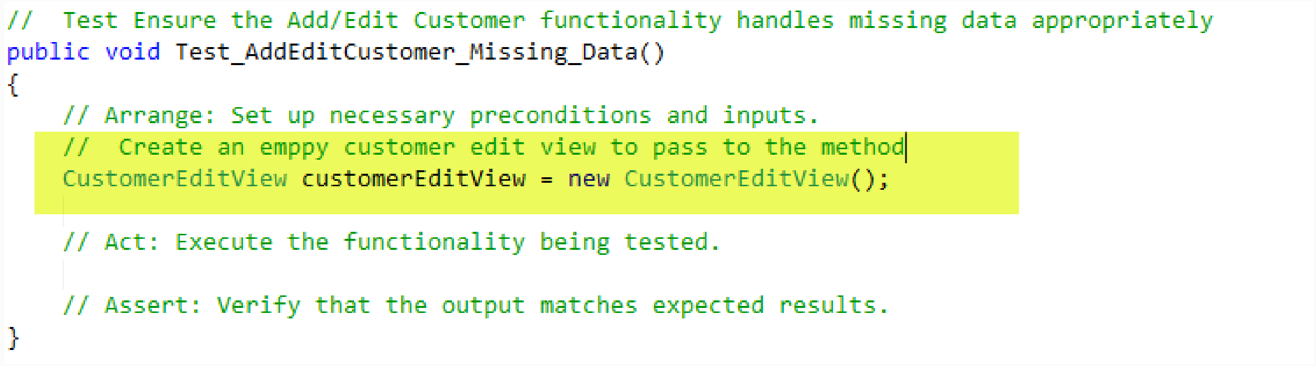
ADD CHECK FOR INVALID PROPERTIES

Copy Code to Students



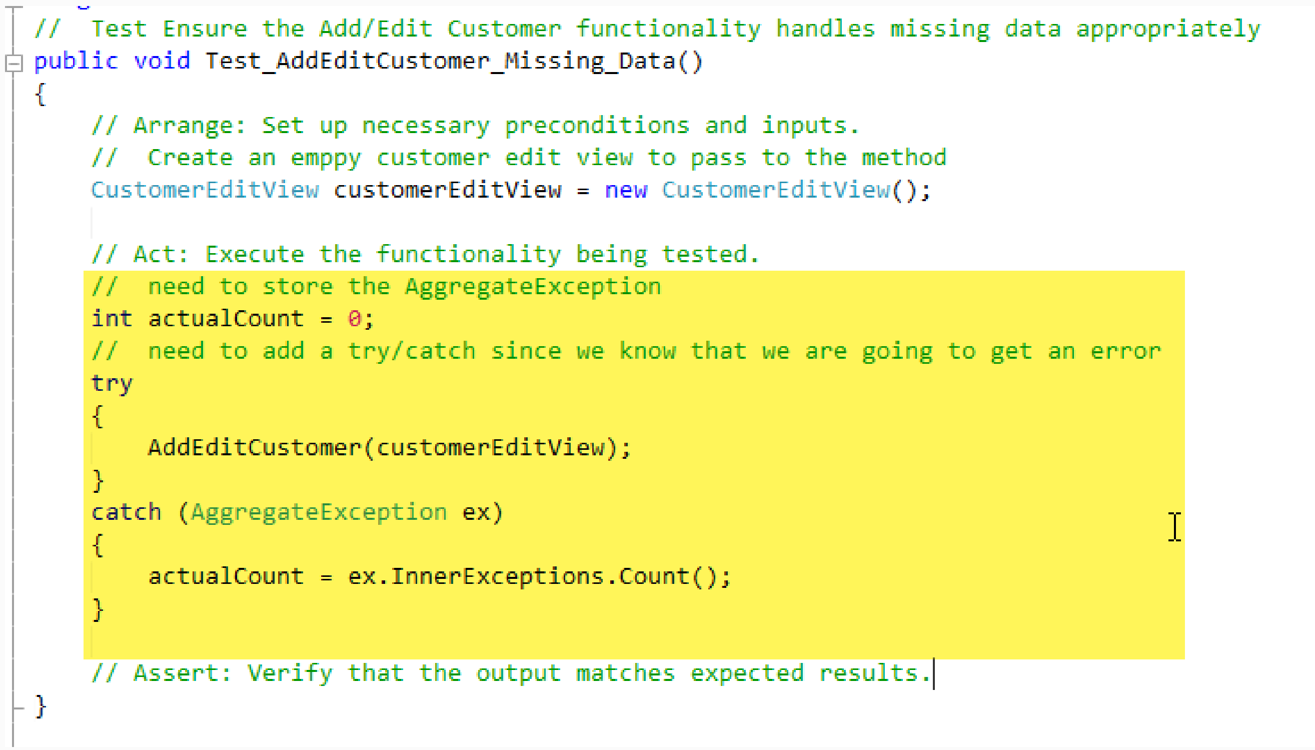
```
/// <summary>
/// Validates properties of the given object. It checks if string properties are null or empty and if int properties have a value of 0.
/// Any detected invalid properties are added as exceptions to the provided error list.
/// </summary>
/// <param name="errorList">List to collect exceptions related to invalid properties.</param>
/// <param name="src">Object to be inspected for invalid properties.</param>
/// <param name="ignoreProperties">List of property names that should be excluded from validation.</param>
private void CheckForInvalidProperties(List<Exception> errorList, object src, List<string> ignoreProperties)
{
    // Check if the source object is null before proceeding.
    if (src == null)
        throw new ArgumentNullException(nameof(src));

    // Iterate through each property of the source object using reflection.
    foreach (PropertyInfo property in src.GetType().GetProperties())
    {
        // If the current property is in the ignore list, skip its validation.
        if (ignoreProperties.Contains(property.Name))
            continue;

        // Fetch the value of the current property.
        var value = property.GetValue(src);

        // Validate string properties for null or empty values.
        if (property.PropertyType == typeof(string))
        {
            if (string.IsNullOrEmpty((string)value))
            {
                errorList.Add(new Exception($"Property '{property.Name}' is null or empty."));
            }
        }
        // Validate integer properties for a value of 0.
        else if (property.PropertyType == typeof(int))
        {
            if ((int)value == 0)
            {
                errorList.Add(new Exception($"Property '{property.Name}' has a value of 0."));
            }
        }
    }
}
```

REFACTOR ADD/EDIT CUSTOMER MISSING DATA (ARRANGE)



```
//  Test Ensure the Add/Edit Customer functionality handles missing data appropriately
public void Test_AddEditCustomer_Missing_Data()
{
    // Arrange: Set up necessary preconditions and inputs.
    //  Create an emppy customer edit view to pass to the method
    CustomerEditView customerEditView = new CustomerEditView();

    // Act: Execute the functionality being tested.

    // Assert: Verify that the output matches expected results.
}
```

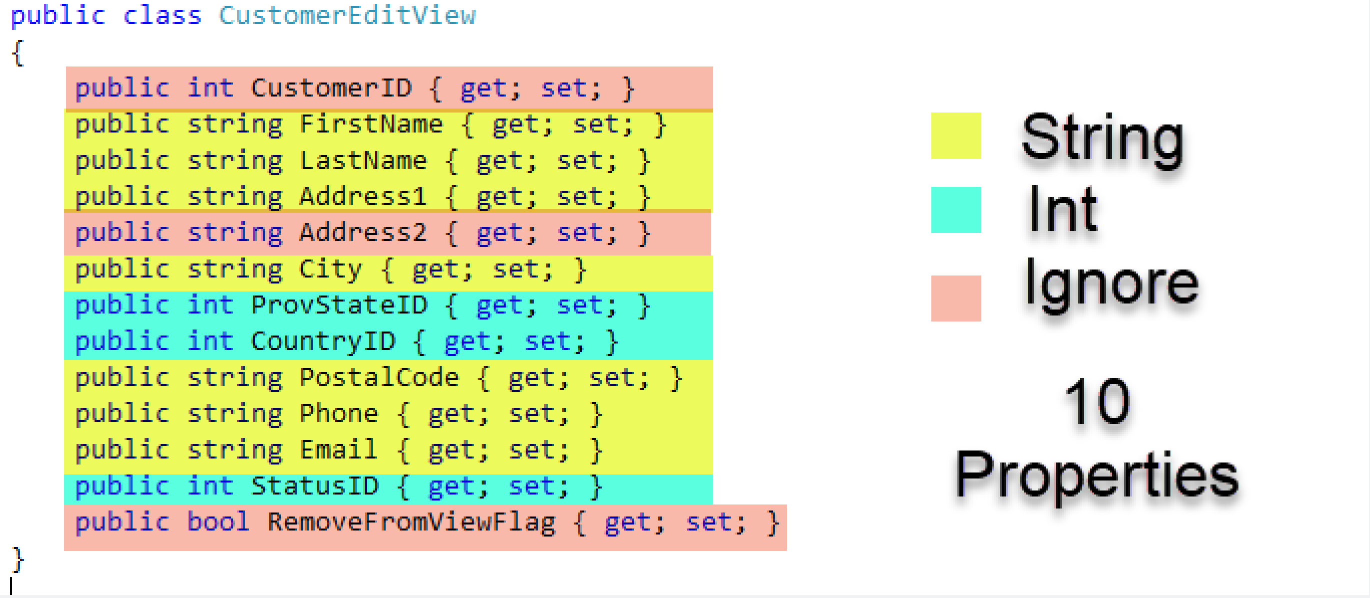
REFACTOR ADD/EDIT CUSTOMER MISSING DATA (ACT)



```
//  Test Ensure the Add/Edit Customer functionality handles missing data appropriately
public void Test_AddEditCustomer_Missing_Data()
{
    // Arrange: Set up necessary preconditions and inputs.
    //  Create an emppy customer edit view to pass to the method
    CustomerEditView customerEditView = new CustomerEditView();

    // Act: Execute the functionality being tested.
    //  need to store the AggregateException
    int actualCount = 0;
    //  need to add a try/catch since we know that we are going to get an error
    try
    {
        AddEditCustomer(customerEditView);
    }
    catch (AggregateException ex)
    {
        actualCount = ex.InnerExceptions.Count();
    }

    // Assert: Verify that the output matches expected results.
}
```

Refactor Add/Edit Customer Missing Data (Assert)

Review the CustomerEditView Properties



```
public class CustomerEditView
{
    public int CustomerID { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Address1 { get; set; }
    public string Address2 { get; set; }
    public string City { get; set; }
    public int ProvStateID { get; set; }
    public int CountryID { get; set; }
    public string PostalCode { get; set; }
    public string Phone { get; set; }
    public string Email { get; set; }
    public int StatusID { get; set; }
    public bool RemoveFromViewFlag { get; set; }
}
```

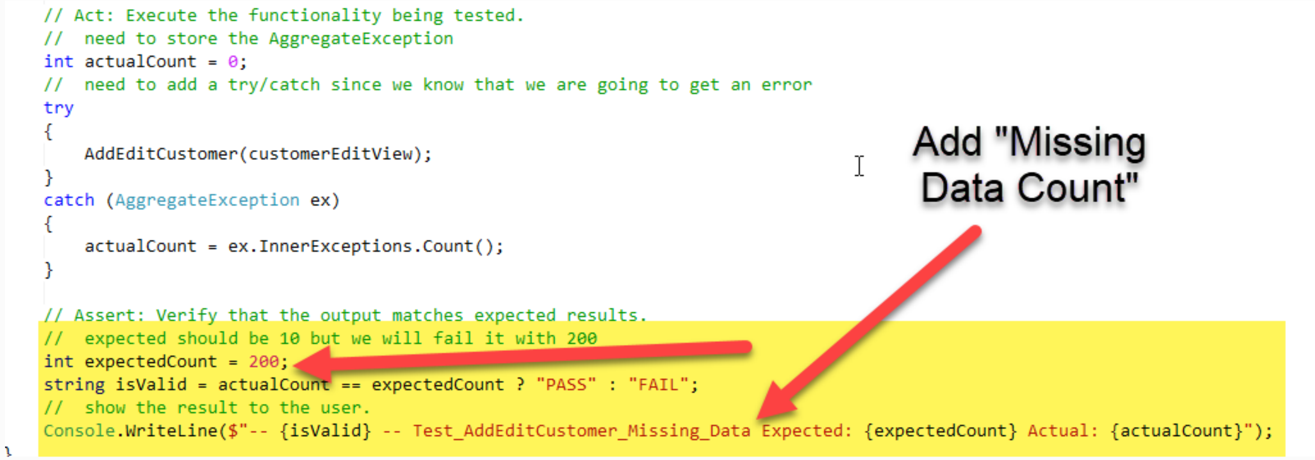
🟨 String

🟦 Int

🟧 Ignore

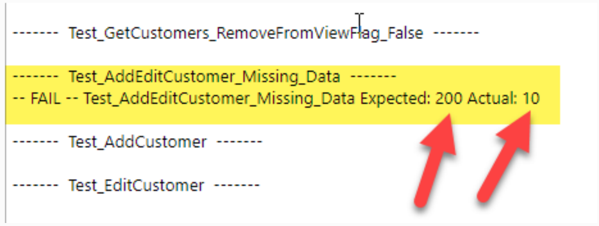
10 Properties

REFACTOR ADD/EDIT CUSTOMER MISSING DATA (ASSERT) – ERROR LIST COUNT (FAIL)



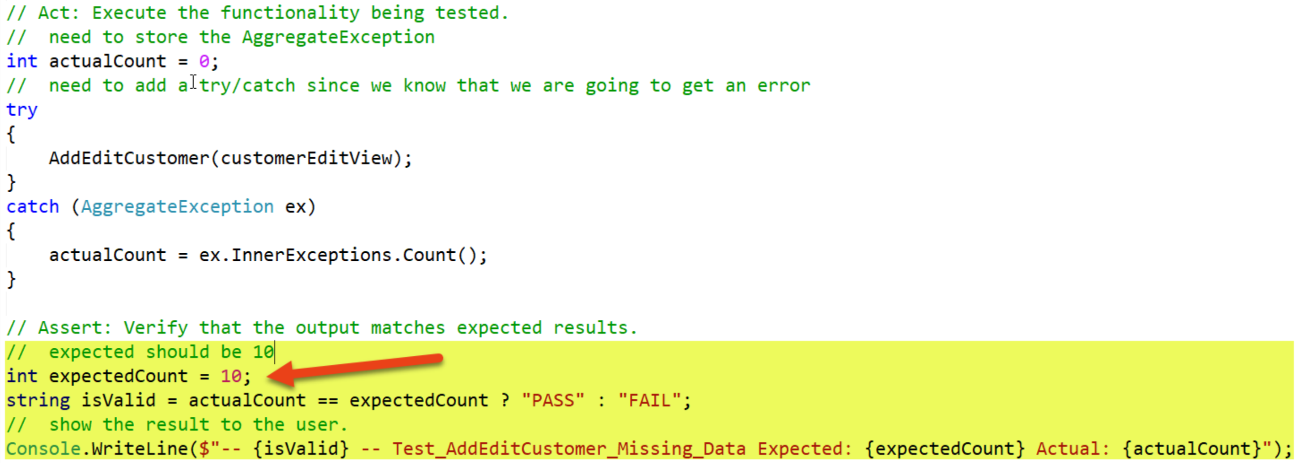
```
// Act: Execute the functionality being tested.
//  need to store the AggregateException
int actualCount = 0;
//  need to add a try/catch since we know that we are going to get an error
try
{
    AddEditCustomer(customerEditView);
}
catch (AggregateException ex)
{
    actualCount = ex.InnerExceptions.Count();
}

// Assert: Verify that the output matches expected results.
//  expected should be 10 but we will fail it with 200
int expectedCount = 200;
string isValid = actualCount == expectedCount ? "PASS" : "FAIL";
//  show the result to the user.
Console.WriteLine($"-- {isValid} -- Test_AddEditCustomer_Missing_Data Expected: {expectedCount} Actual: {actualCount}");
```

Add "Missing Data Count"

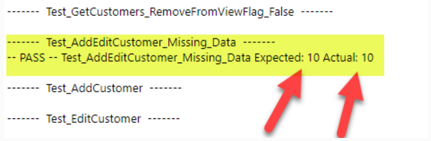


```
------- Test_GetCustomers_RemoveFromViewFlag_False -------

------- Test_AddEditCustomer_Missing_Data -------
-- FAIL -- Test_AddEditCustomer_Missing_Data Expected: 200 Actual: 10

------- Test_AddCustomer -------

------- Test_EditCustomer -------
```

REFACTOR ADD/EDIT CUSTOMER MISSING DATA (ASSERT) – ERROR LIST COUNT (PASS)



```
// Act: Execute the functionality being tested.
//  need to store the AggregateException
int actualCount = 0;
//  need to add a try/catch since we know that we are going to get an error
try
{
    AddEditCustomer(customerEditView);
}
catch (AggregateException ex)
{
    actualCount = ex.InnerExceptions.Count();
}

// Assert: Verify that the output matches expected results.
//  expected should be 10
int expectedCount = 10;
string isValid = actualCount == expectedCount ? "PASS" : "FAIL";
//  show the result to the user.
Console.WriteLine($"-- {isValid} -- Test_AddEditCustomer_Missing_Data Expected: {expectedCount} Actual: {actualCount}");
```



```
------- Test_GetCustomers_RemoveFromViewFlag_False -------

------- Test_AddEditCustomer_Missing_Data -------
-- PASS -- Test_AddEditCustomer_Missing_Data Expected: 10 Actual: 10

------- Test_AddCustomer -------

------- Test_EditCustomer -------
```

REVIEWING THE ADD/EDIT CUSTOMER MISSING DATA ERROR LIST

- There are three types of errors that we must deal with for missing data.

1. String property has an empty value.
2. Lookup value has not been set and has a value of “0”.
3. Property was returned that we were not aware of.

EXAMPLE OF ERROR MESSAGES FROM “CHECK FOR INVALID PROPERTIES” METHOD

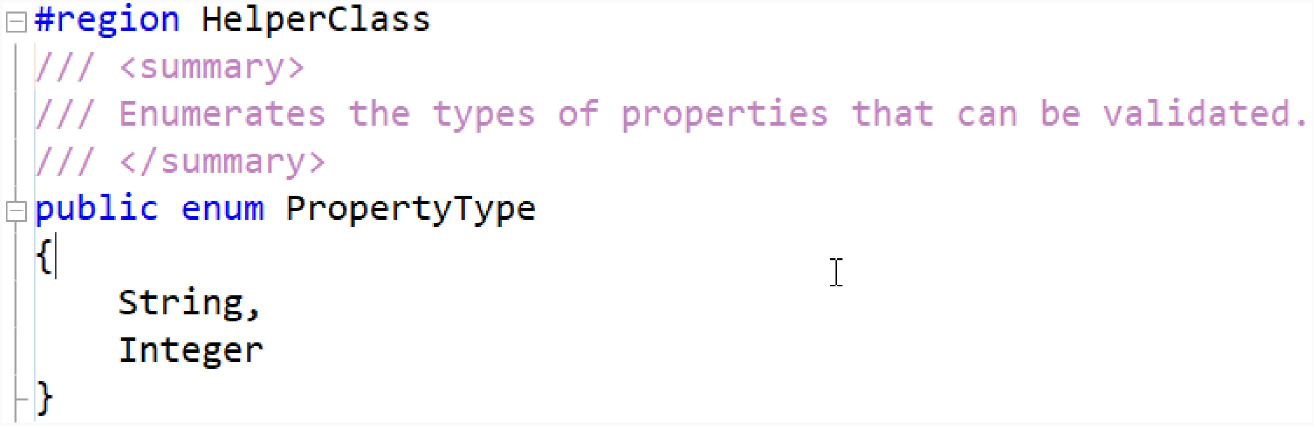
Property ‘FirstName' is null or empty.

Property ‘CountryID’ has a value of 0.

Property ‘Fax’ was found in the error list, not your test.

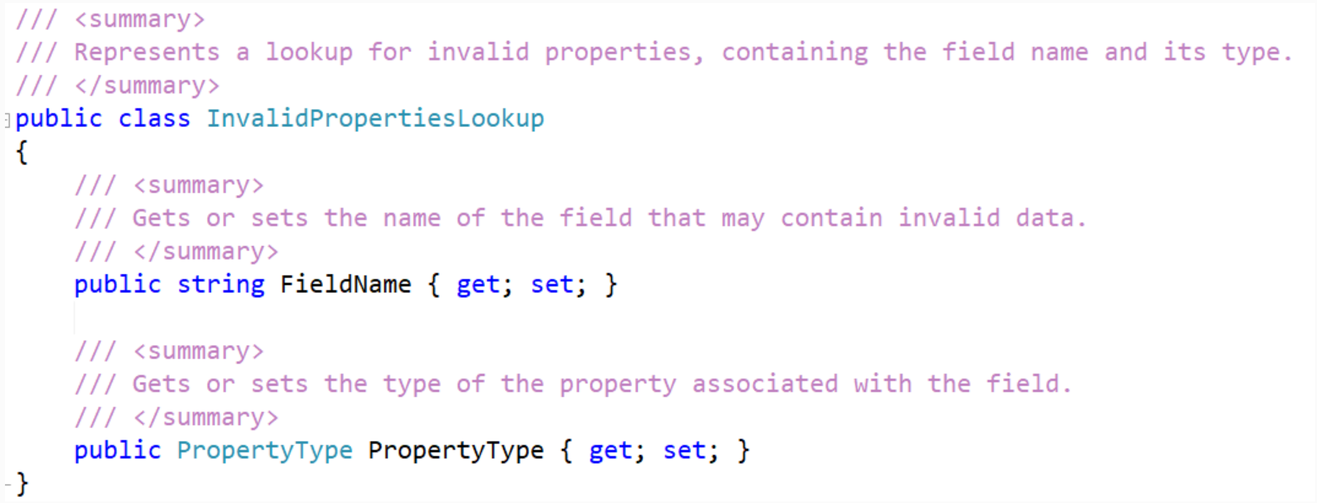
NOTE: This will be return from the “Helper” method.

CREATING HELPER CLASS & METHODS - ENUM PROPERTYTYPE



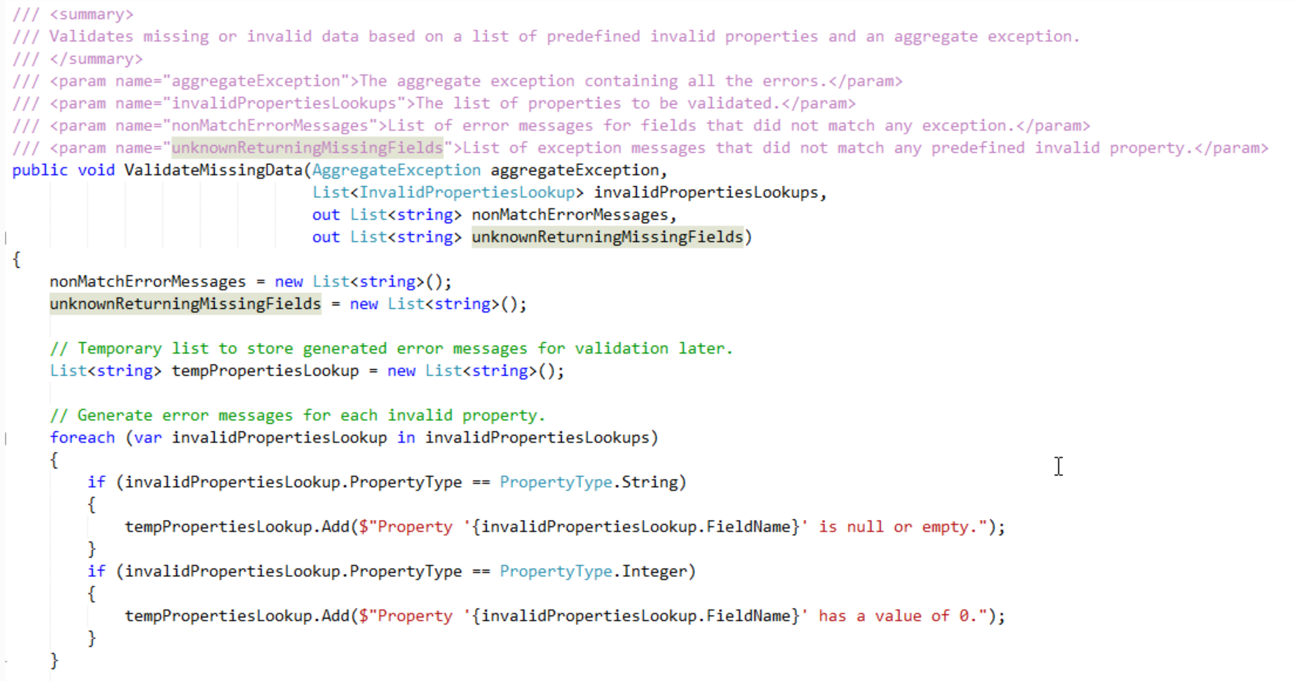
```
#region HelperClass
/// <summary>
/// Enumerates the types of properties that can be validated.
/// </summary>
public enum PropertyType
{
    String,
    Integer
}
```

CREATING HELPER CLASS & METHODS - CLASS “INVALID PROPERTIES LOOKUP”



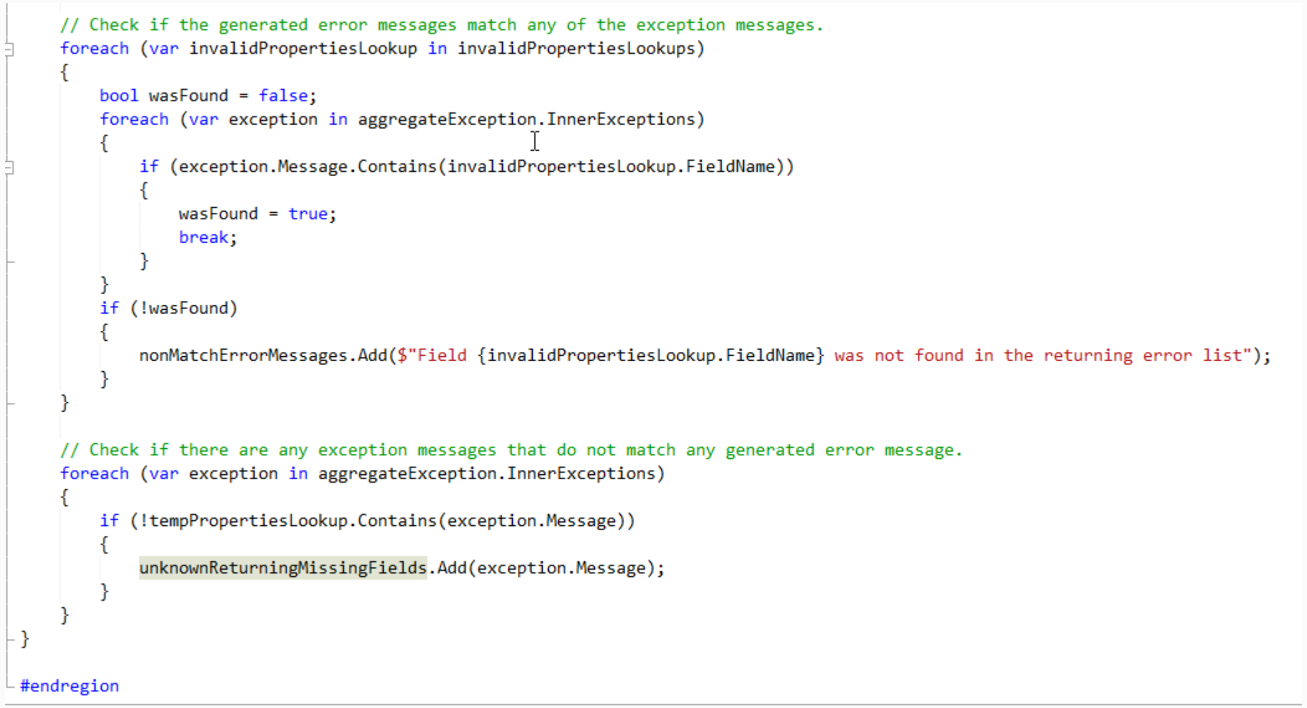
```
/// <summary>
/// Represents a lookup for invalid properties, containing the field name and its type.
/// </summary>
public class InvalidPropertiesLookup
{
    /// <summary>
    /// Gets or sets the name of the field that may contain invalid data.
    /// </summary>
    public string FieldName { get; set; }

    /// <summary>
    /// Gets or sets the type of the property associated with the field.
    /// </summary>
    public PropertyType PropertyType { get; set; }
}
```

CREATING HELPER CLASS & METHODS - METHOD “VALIDATE MISSING DATA” 1/2



```
/// <summary>
/// Validates missing or invalid data based on a list of predefined invalid properties and an aggregate exception.
/// </summary>
/// <param name="aggregateException">The aggregate exception containing all the errors.</param>
/// <param name="invalidPropertiesLookups">The list of properties to be validated.</param>
/// <param name="nonMatchErrorMessages">List of error messages for fields that did not match any exception.</param>
/// <param name="unknownReturningMissingFields">List of exception messages that did not match any predefined invalid property.</param>
public void ValidateMissingData(AggregateException aggregateException,
                                List<InvalidPropertiesLookup> invalidPropertiesLookups,
                                out List<string> nonMatchErrorMessages,
                                out List<string> unknownReturningMissingFields)
{
    nonMatchErrorMessages = new List<string>();
    unknownReturningMissingFields = new List<string>();

    // Temporary list to store generated error messages for validation later.
    List<string> tempPropertiesLookup = new List<string>();

    // Generate error messages for each invalid property.
    foreach (var invalidPropertiesLookup in invalidPropertiesLookups)
    {
        if (invalidPropertiesLookup.PropertyType == PropertyType.String)
        {
            tempPropertiesLookup.Add($"Property '{invalidPropertiesLookup.FieldName}' is null or empty.");
        }
        if (invalidPropertiesLookup.PropertyType == PropertyType.Integer)
        {
            tempPropertiesLookup.Add($"Property '{invalidPropertiesLookup.FieldName}' has a value of 0.");
        }
    }
```

CREATING HELPER CLASS & METHODS - METHOD “VALIDATE MISSING DATA” 2/2

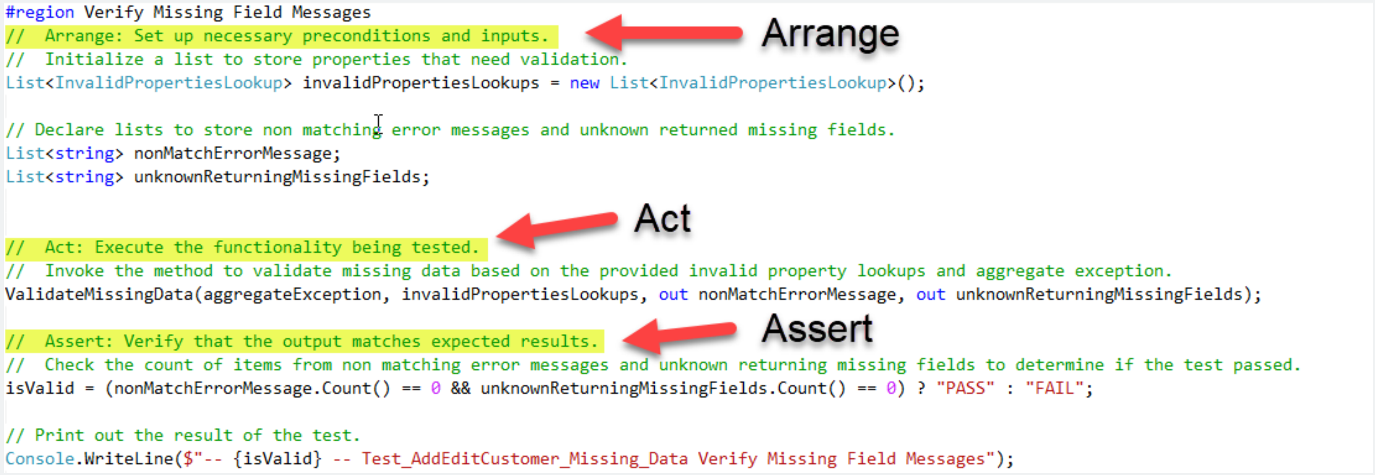


```
    // Check if the generated error messages match any of the exception messages.
    foreach (var invalidPropertiesLookup in invalidPropertiesLookups)
    {
        bool wasFound = false;
        foreach (var exception in aggregateException.InnerExceptions)
        {
            if (exception.Message.Contains(invalidPropertiesLookup.FieldName))
            {
                wasFound = true;
                break;
            }
        }
        if (!wasFound)
        {
            nonMatchErrorMessages.Add($"Field {invalidPropertiesLookup.FieldName} was not found in the returning error list");
        }
    }

    // Check if there are any exception messages that do not match any generated error message.
    foreach (var exception in aggregateException.InnerExceptions)
    {
        if (!tempPropertiesLookup.Contains(exception.Message))
        {
            unknownReturningMissingFields.Add(exception.Message);
        }
    }
}

#endregion
```

Let's Start by Showing Return Data That We Did Not Know About

The code for testing “Missing Field Messages”

Part 1 of 2

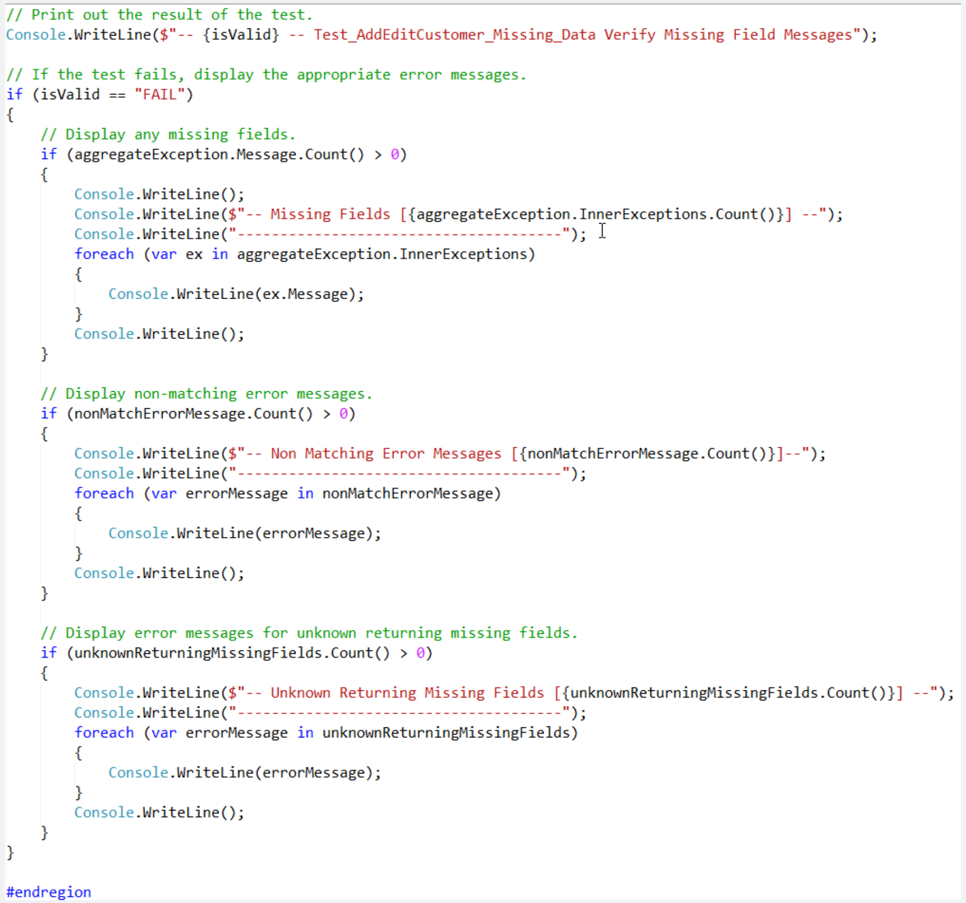


```
#region Verify Missing Field Messages
//  Arrange: Set up necessary preconditions and inputs.
//  Initialize a list to store properties that need validation.
List<InvalidPropertiesLookup> invalidPropertiesLookups = new List<InvalidPropertiesLookup>();

// Declare lists to store non matching error messages and unknown returned missing fields.
List<string> nonMatchErrorMessage;
List<string> unknownReturningMissingFields;


//  Act: Execute the functionality being tested.
//  Invoke the method to validate missing data based on the provided invalid property lookups and aggregate exception.
ValidateMissingData(aggregateException, invalidPropertiesLookups, out nonMatchErrorMessage, out unknownReturningMissingFields);

//  Assert: Verify that the output matches expected results.
//  Check the count of items from non matching error messages and unknown returning missing fields to determine if the test passed.
isValid = (nonMatchErrorMessage.Count() == 0 && unknownReturningMissingFields.Count() == 0) ? "PASS" : "FAIL";

// Print out the result of the test.
Console.WriteLine($"-- {isValid} -- Test_AddEditCustomer_Missing_Data Verify Missing Field Messages");
```

Let's Start by Showing Return Data That We Did Not Know About

The code for testing “Missing Field Messages”

Part 2 of 2

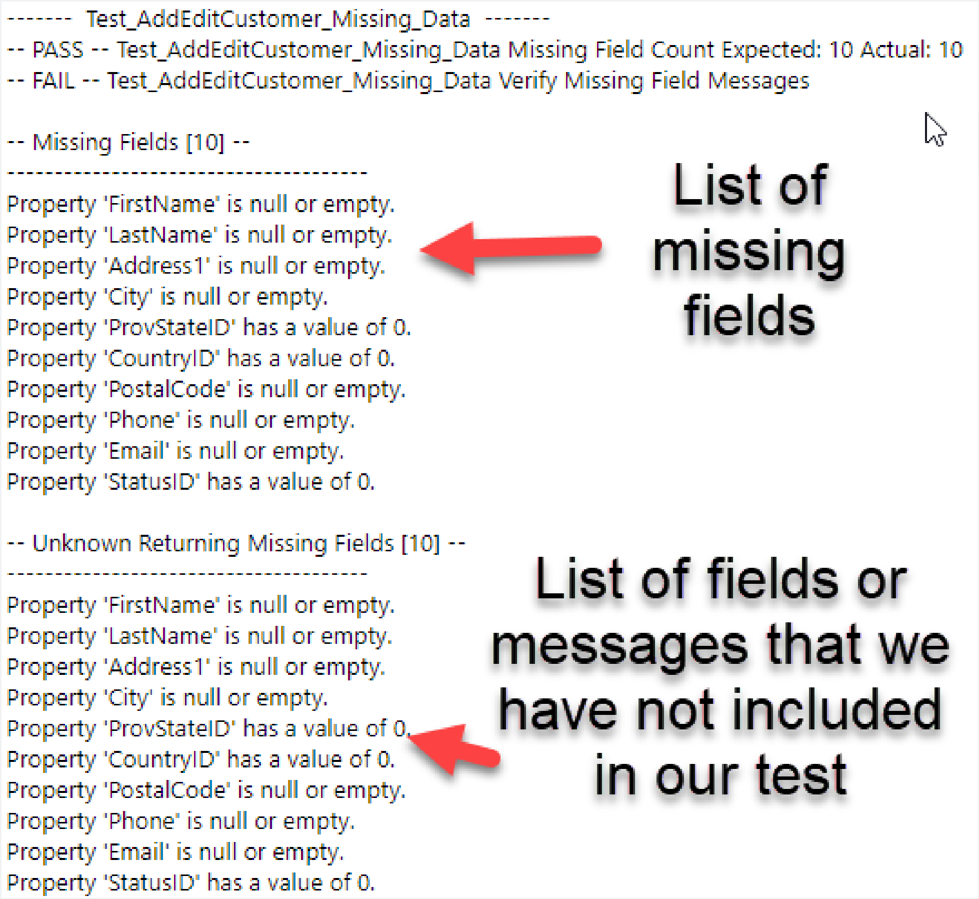


```
// Print out the result of the test.
Console.WriteLine($"-- {isValid} -- Test_AddEditCustomer_Missing_Data Verify Missing Field Messages");

// If the test fails, display the appropriate error messages.
if (isValid == "FAIL")
{
    // Display any missing fields.
    if (aggregateException.Message.Count() > 0)
    {
        Console.WriteLine();
        Console.WriteLine($"-- Missing Fields [{aggregateException.InnerExceptions.Count()}] --");
        Console.WriteLine("--------------------------------------");
        foreach (var ex in aggregateException.InnerExceptions)
        {
            Console.WriteLine(ex.Message);
        }
        Console.WriteLine();
    }

    // Display non-matching error messages.
    if (nonMatchErrorMessage.Count() > 0)
    {
        Console.WriteLine($"-- Non Matching Error Messages [{nonMatchErrorMessage.Count()}]--");
        Console.WriteLine("--------------------------------------");
        foreach (var errorMessage in nonMatchErrorMessage)
        {
            Console.WriteLine(errorMessage);
        }
        Console.WriteLine();
    }

    // Display error messages for unknown returning missing fields.
    if (unknownReturningMissingFields.Count() > 0)
    {
        Console.WriteLine($"-- Unknown Returning Missing Fields [{unknownReturningMissingFields.Count()}] --");
        Console.WriteLine("--------------------------------------");
        foreach (var errorMessage in unknownReturningMissingFields)
        {
            Console.WriteLine(errorMessage);
        }
        Console.WriteLine();
    }
}

#endregion
```

REVIEWING OUTPUT

Running the test without setting up the unit test for known missing fields.

FAIL

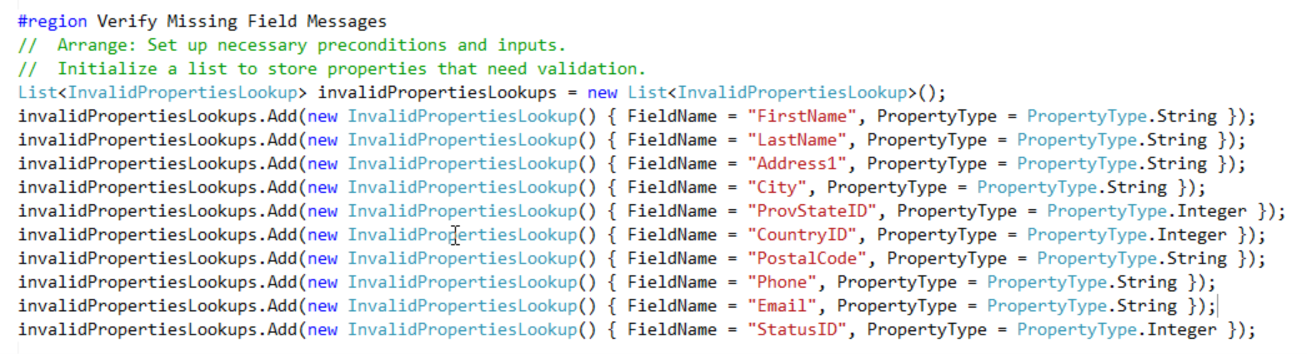


```
-------  Test_AddEditCustomer_Missing_Data  -------
-- PASS -- Test_AddEditCustomer_Missing_Data Missing Field Count Expected: 10 Actual: 10
-- FAIL -- Test_AddEditCustomer_Missing_Data Verify Missing Field Messages

-- Missing Fields [10] --
--------------------------------------
Property 'FirstName' is null or empty.
Property 'LastName' is null or empty.
Property 'Address1' is null or empty.
Property 'City' is null or empty.
Property 'ProvStateID' has a value of 0.
Property 'CountryID' has a value of 0.
Property 'PostalCode' is null or empty.
Property 'Phone' is null or empty.
Property 'Email' is null or empty.
Property 'StatusID' has a value of 0.

-- Unknown Returning Missing Fields [10] --
--------------------------------------
Property 'FirstName' is null or empty.
Property 'LastName' is null or empty.
Property 'Address1' is null or empty.
Property 'City' is null or empty.
Property 'ProvStateID' has a value of 0.
Property 'CountryID' has a value of 0.
Property 'PostalCode' is null or empty.
Property 'Phone' is null or empty.
Property 'Email' is null or empty.
Property 'StatusID' has a value of 0.
```

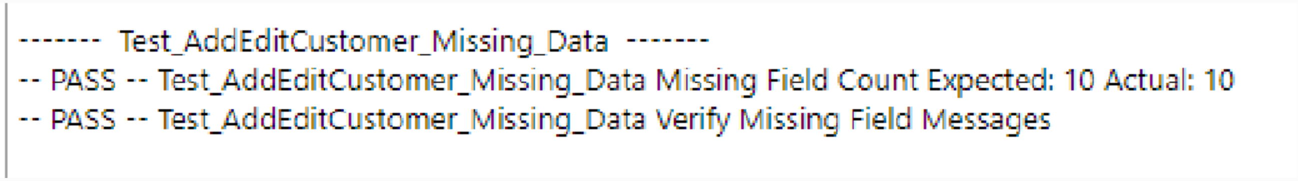
List of missing fields

List of fields or messages that we have not included in our test

UPDATE INVALID PROPERTIES LOOKUP WITH ALL PROPERTIES NAMES

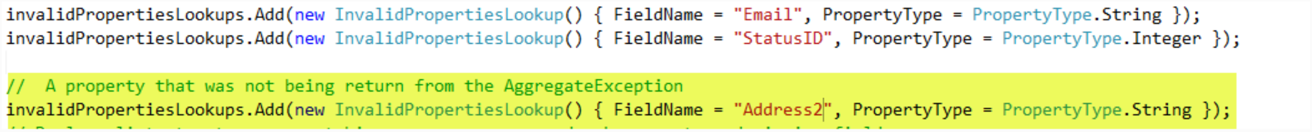


```
#region Verify Missing Field Messages
//  Arrange: Set up necessary preconditions and inputs.
//  Initialize a list to store properties that need validation.
List<InvalidPropertiesLookup> invalidPropertiesLookups = new List<InvalidPropertiesLookup>();
invalidPropertiesLookups.Add(new InvalidPropertiesLookup() { FieldName = "FirstName", PropertyType = PropertyType.String });
invalidPropertiesLookups.Add(new InvalidPropertiesLookup() { FieldName = "LastName", PropertyType = PropertyType.String });
invalidPropertiesLookups.Add(new InvalidPropertiesLookup() { FieldName = "Address1", PropertyType = PropertyType.String });
invalidPropertiesLookups.Add(new InvalidPropertiesLookup() { FieldName = "City", PropertyType = PropertyType.String });
invalidPropertiesLookups.Add(new InvalidPropertiesLookup() { FieldName = "ProvStateID", PropertyType = PropertyType.Integer });
invalidPropertiesLookups.Add(new InvalidPropertiesLookup() { FieldName = "CountryID", PropertyType = PropertyType.Integer });
invalidPropertiesLookups.Add(new InvalidPropertiesLookup() { FieldName = "PostalCode", PropertyType = PropertyType.String });
invalidPropertiesLookups.Add(new InvalidPropertiesLookup() { FieldName = "Phone", PropertyType = PropertyType.String });
invalidPropertiesLookups.Add(new InvalidPropertiesLookup() { FieldName = "Email", PropertyType = PropertyType.String });
invalidPropertiesLookups.Add(new InvalidPropertiesLookup() { FieldName = "StatusID", PropertyType = PropertyType.Integer });
```

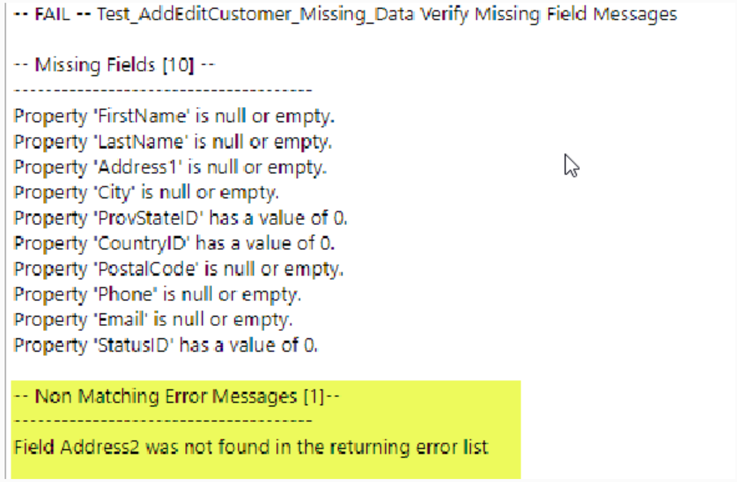


```
-------  Test_AddEditCustomer_Missing_Data  -------
-- PASS -- Test_AddEditCustomer_Missing_Data Missing Field Count Expected: 10 Actual: 10
-- PASS -- Test_AddEditCustomer_Missing_Data Verify Missing Field Messages
```

UPDATE INVALID PROPERTIES LOOKUP WITH A PROPERTY THAT IS NOT FOUND IN THE LIST OF AGGREGATE EXCEPTION



```
invalidPropertiesLookups.Add(new InvalidPropertiesLookup() { FieldName = "Email", PropertyType = PropertyType.String });
invalidPropertiesLookups.Add(new InvalidPropertiesLookup() { FieldName = "StatusID", PropertyType = PropertyType.Integer });

//  A property that was not being return from the AggregateException
invalidPropertiesLookups.Add(new InvalidPropertiesLookup() { FieldName = "Address2", PropertyType = PropertyType.String });
```



```
-- FAIL -- Test_AddEditCustomer_Missing_Data Verify Missing Field Messages

-- Missing Fields [10] --
------------------------------------
Property 'FirstName' is null or empty.
Property 'LastName' is null or empty.
Property 'Address1' is null or empty.
Property 'City' is null or empty.
Property 'ProvStateID' has a value of 0.
Property 'CountryID' has a value of 0.
Property 'PostalCode' is null or empty.
Property 'Phone' is null or empty.
Property 'Email' is null or empty.
Property 'StatusID' has a value of 0.

-- Non Matching Error Messages [1]--
------------------------------------
Field Address2 was not found in the returning error list
```

ResetTestData()