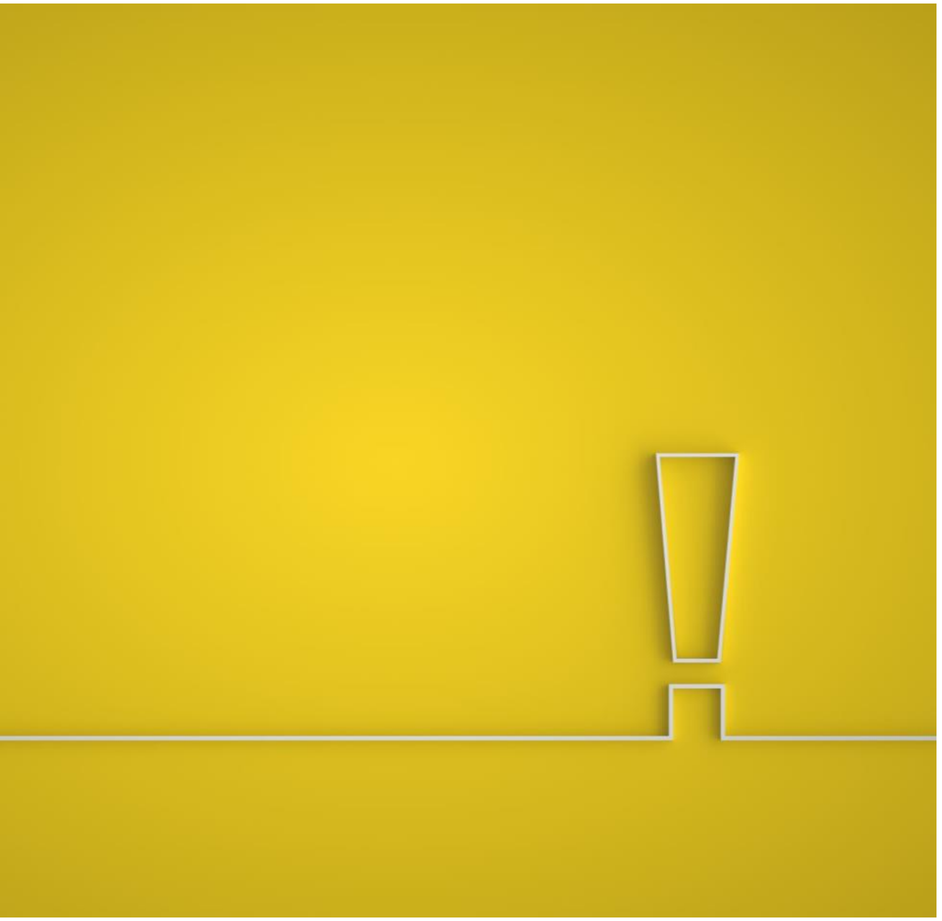
Session 6: OLTP Single Record

Adding & Editing Single Record



Session's Objective

- Review OLTP
- Setup LINQPad for Entity Framework Core (Westwind Database)
- Reviewing the OLTP Parts
- Simple CRUD Methods
  - Define the rules.
  - Create “Unit Tests” based on the rules using Arrange, Act, and Assert patterns.
  - Create the method for CRUD

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.

- **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.

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.

- **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

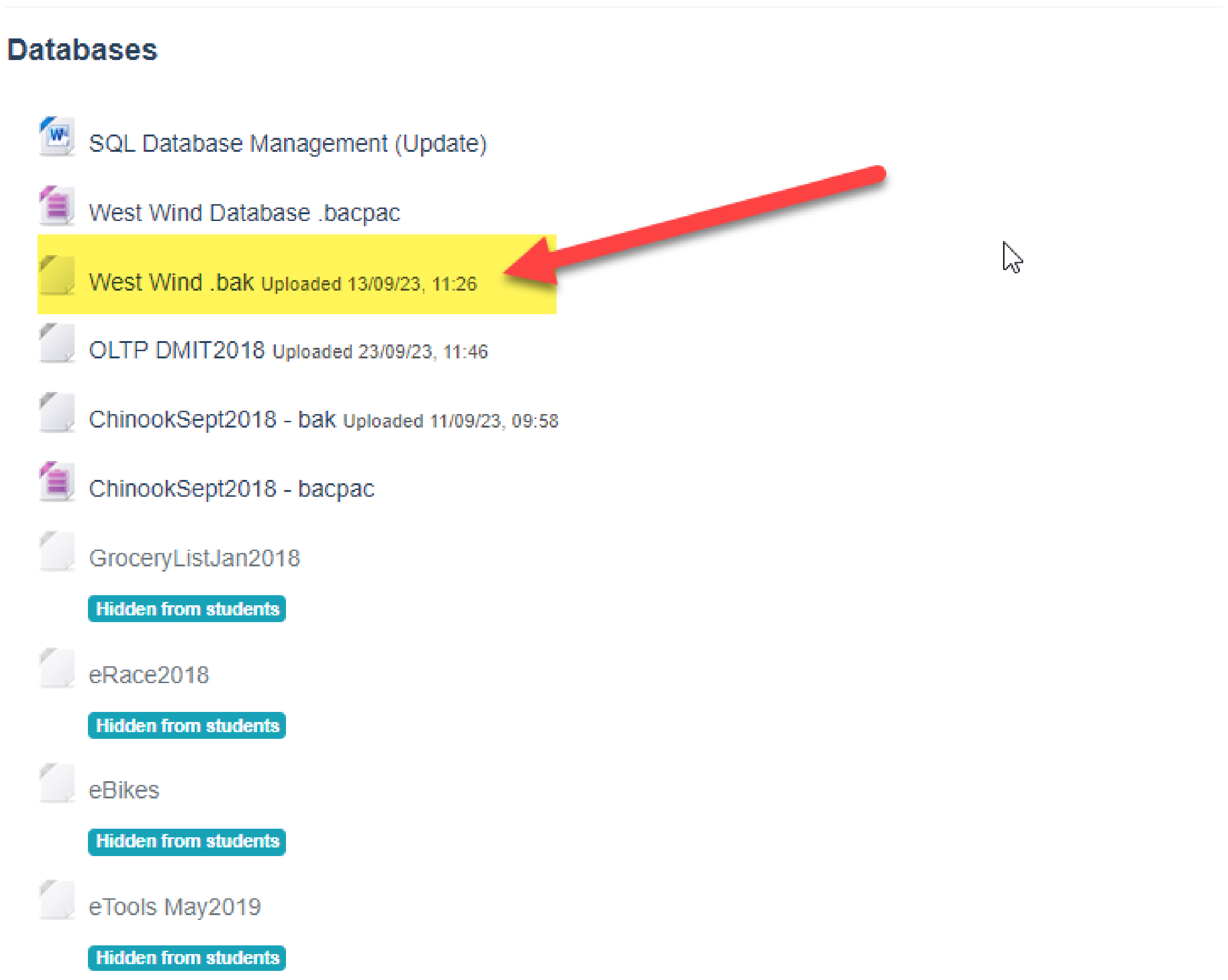
Delete (Removal of data)

- **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.

Entity Framework
Connection

Westwind Database

Install Westwind Database


Databases

SQL Database Management (Update)

West Wind Database .bacpac

West Wind .bak Uploaded 13/09/23, 11:26

OLTP DMIT2018 Uploaded 23/09/23, 11:46

ChinookSept2018 - bak Uploaded 11/09/23, 09:58

ChinookSept2018 - bacpac

GroceryListJan2018

Hidden from students

eRace2018

Hidden from students

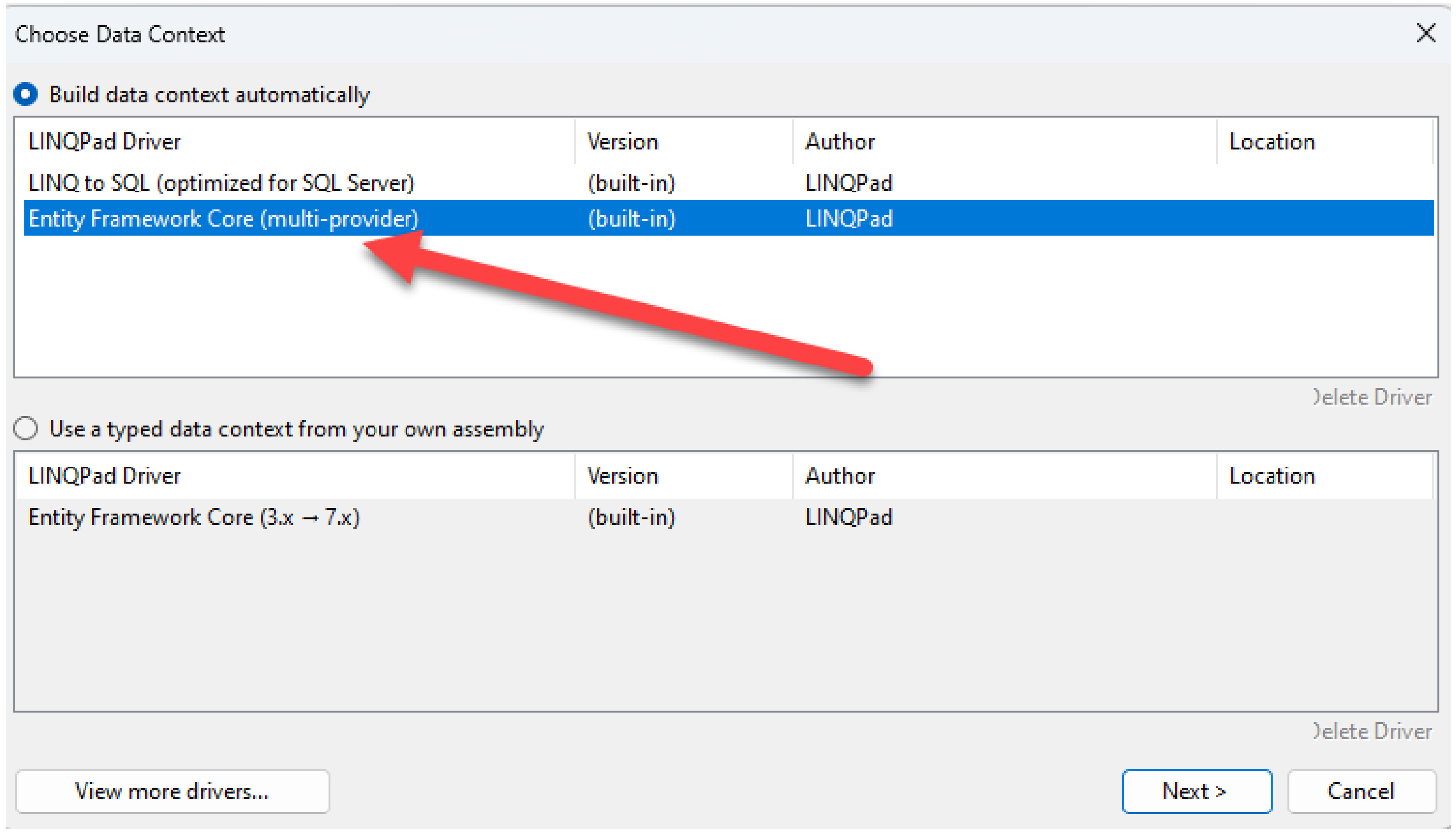
eBikes

Hidden from students

eTools May2019

Hidden from students

CREATING LINQPAD ENTITY FRAMEWORK CONNECTION


Choose Data Context

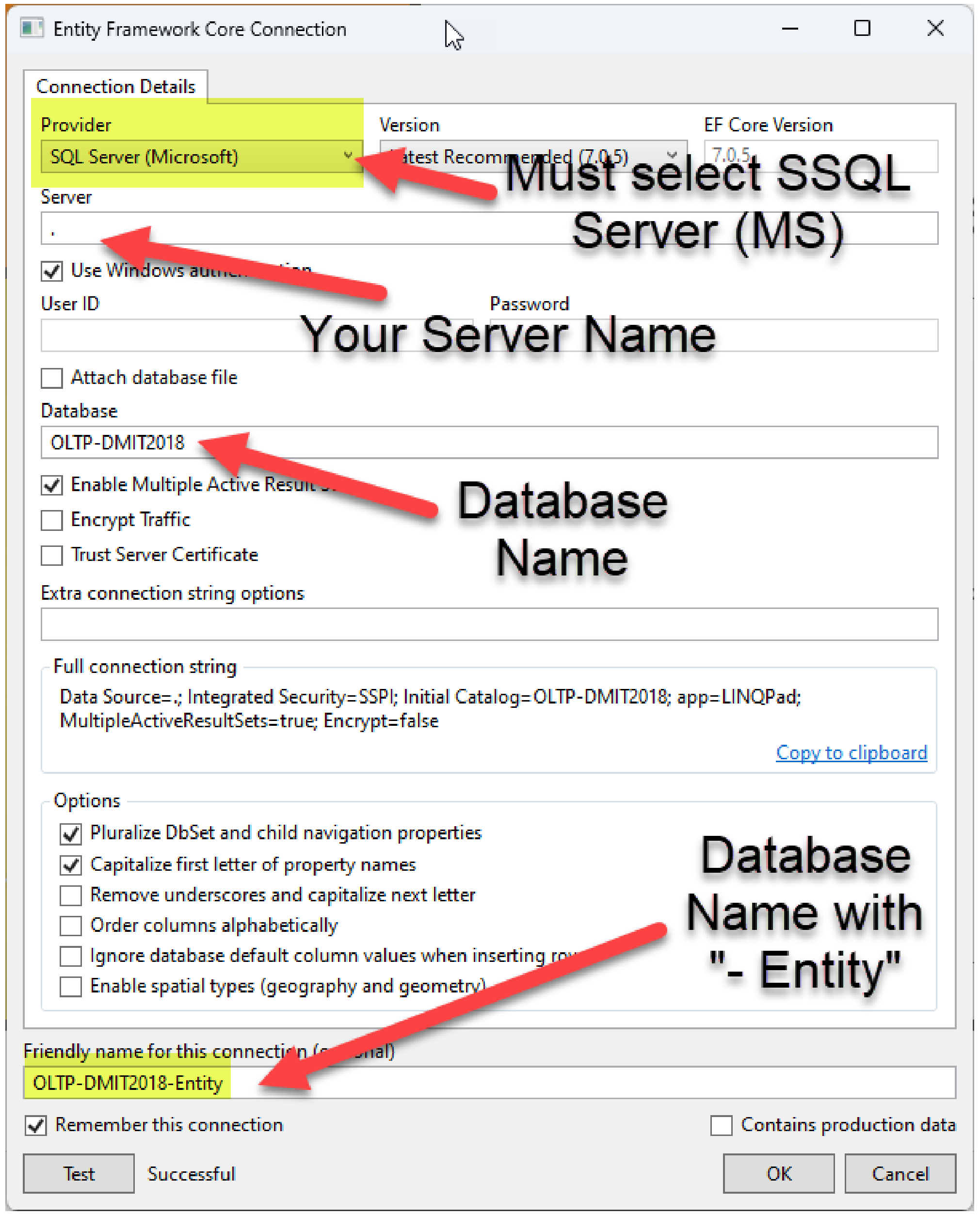
[✓ Build data context automatically]

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



Entity Framework Core Connection

Connection Details

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

Copy to clipboard

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

Test Successful

OK Cancel

Must select SSQL Server (MS)

Your Server Name

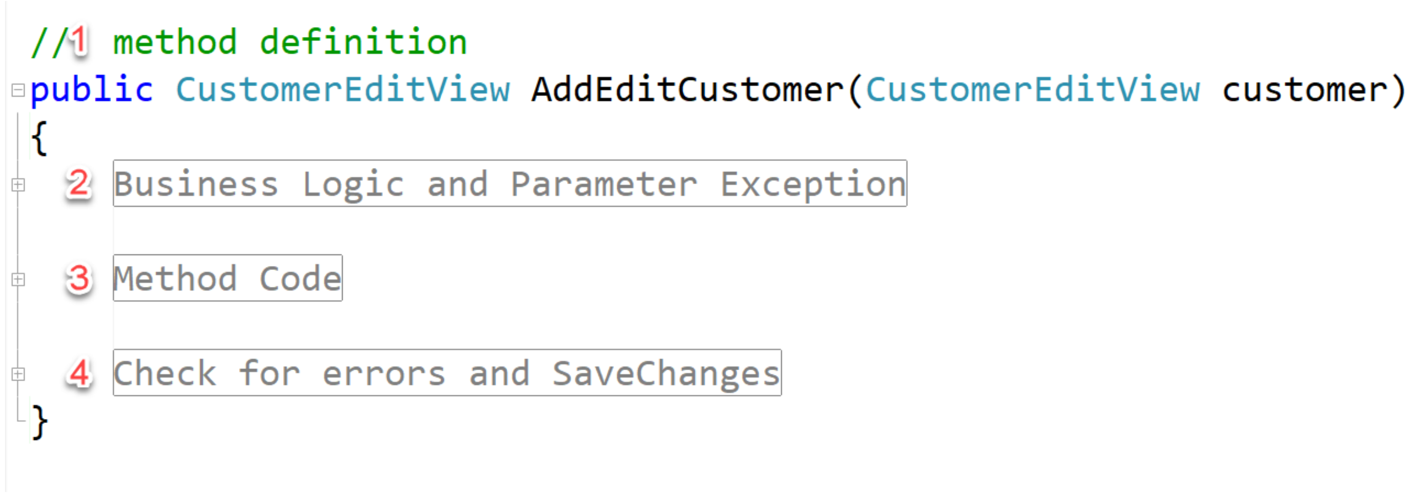
Database Name

Database Name with "- Entity"

OLTP Parts

- Four parts make up the OLTP Method

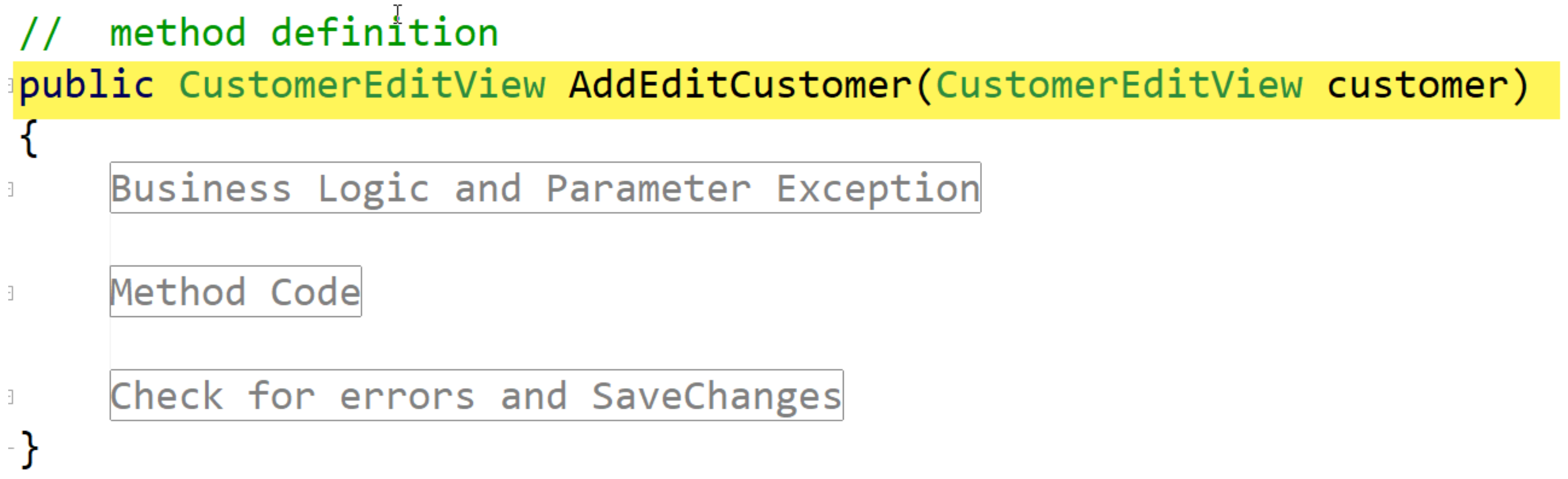
1. Method Definition.
2. Business Logic and Parameter Exception.
3. Method Code.
4. Final Error Checking and Saving of Data.



```
//1 method definition
public CustomerEditView AddEditCustomer(CustomerEditView customer)
{
   2 Business Logic and Parameter Exception

   3 Method Code

   4 Check for errors and SaveChanges
}
```

OLTP Parts - Method Definition

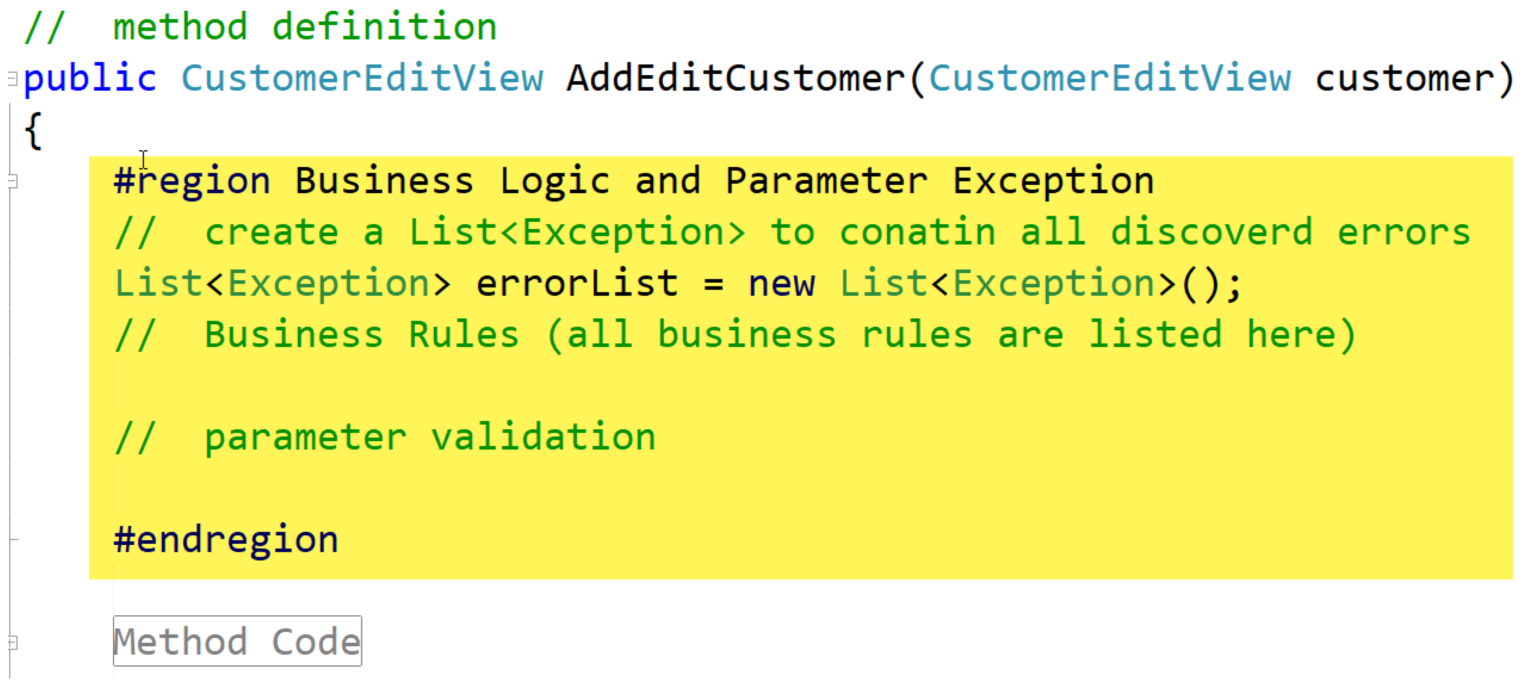
The method definition is the entry point to your code.



```
//  method definition
public CustomerEditView AddEditCustomer(CustomerEditView customer)
{
    Business Logic and Parameter Exception

    Method Code

    Check for errors and SaveChanges
}
```

OLTP Parts - Business Logic and Parameter Exception

All business rules are listed here, and any pre-validation is performed before entering the "Method Code" area.



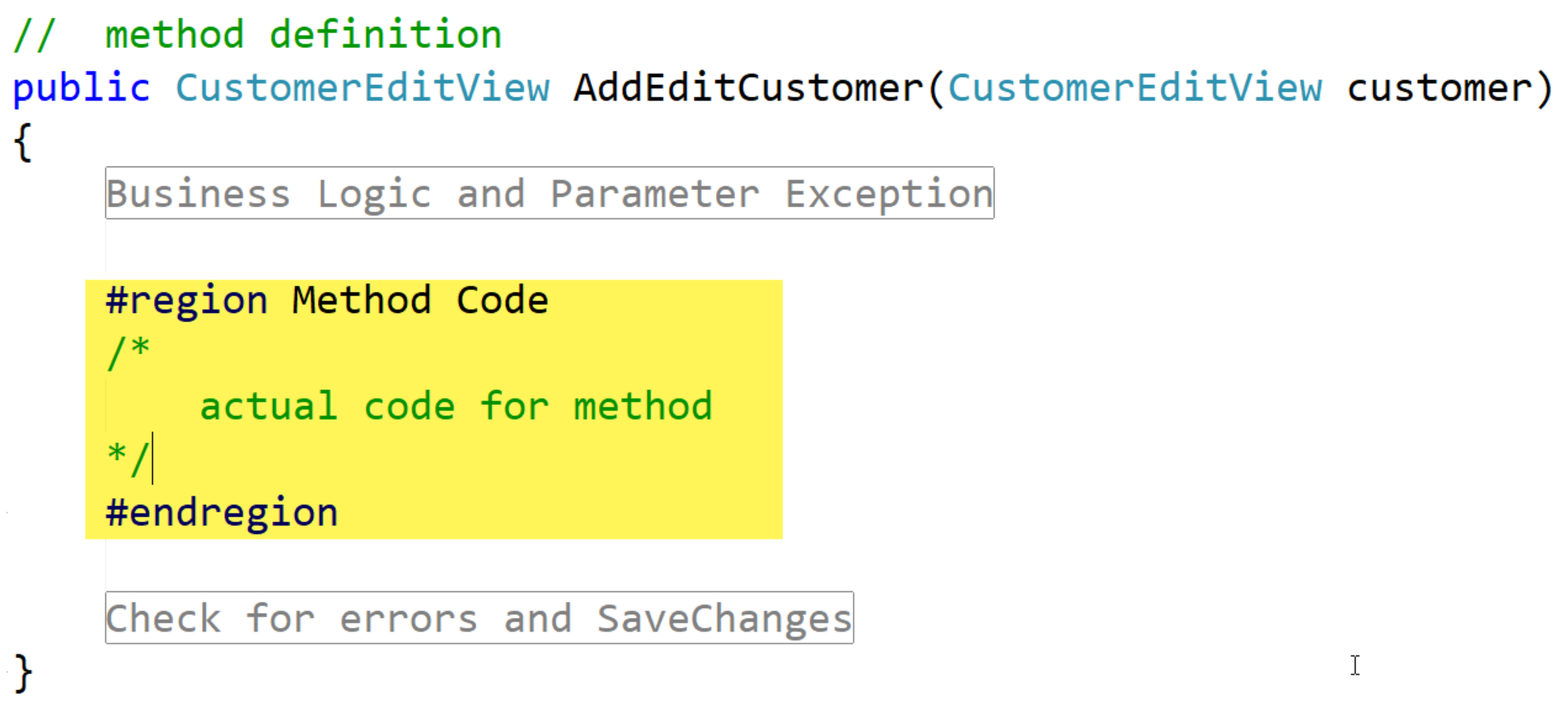
```
//  method definition
public CustomerEditView AddEditCustomer(CustomerEditView customer)
{
    #region Business Logic and Parameter Exception
    //  create a List<Exception> to conatin all discoverd errors
    List<Exception> errorList = new List<Exception>();
    //  Business Rules (all business rules are listed here)

    //  parameter validation

    #endregion

    Method Code
```

OLTP Parts - Method Code

Your actual code is handled here. This is where the changes to the database is coded. i.e., Add, Update or Deleting of data

NOTE: You may still run into exceptions you will handle here. This can be handled by throwing an "Exception" or adding the exception to the "error list."

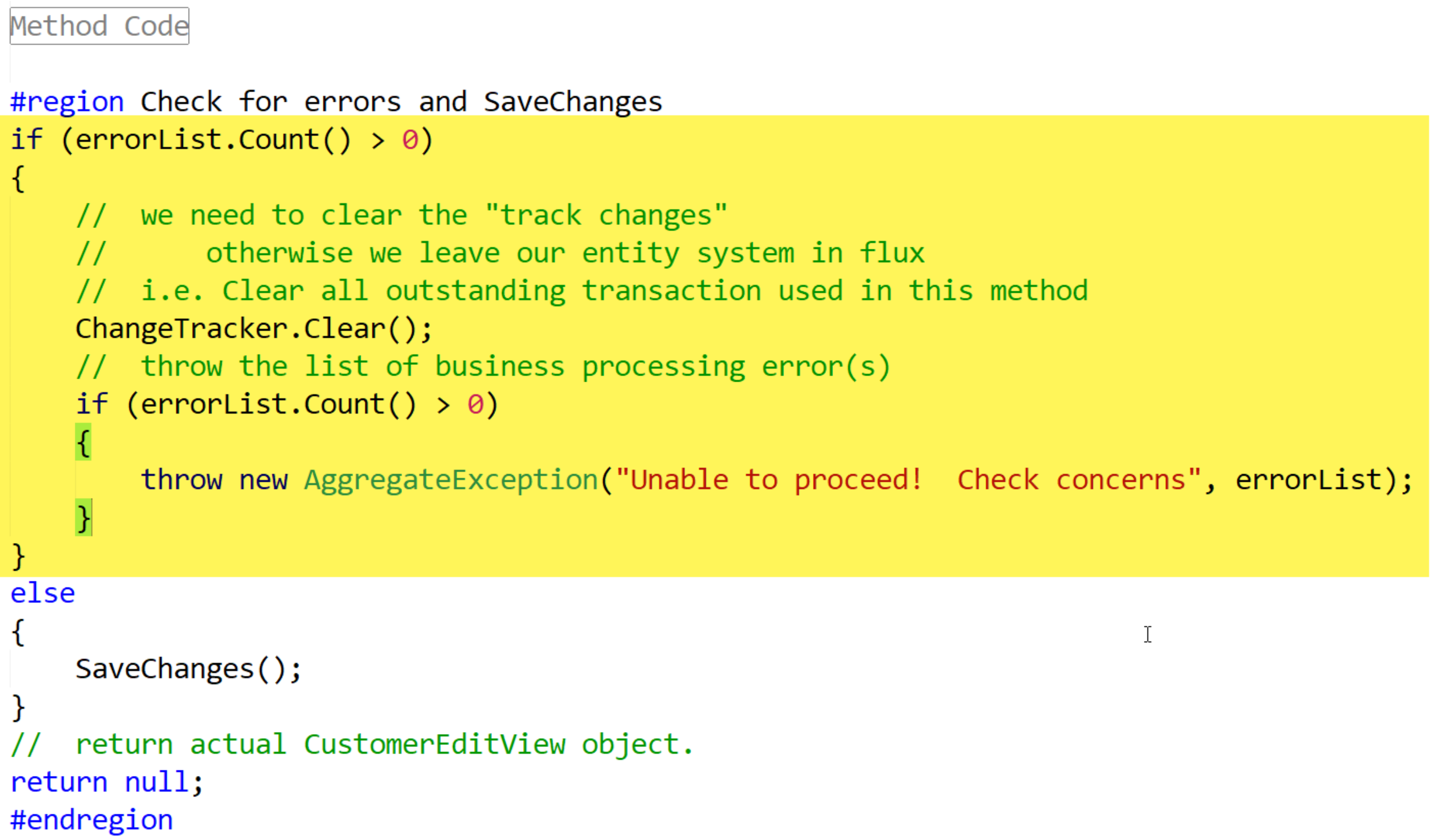
i.e., The sale price has expired.



```
//  method definition
public CustomerEditView AddEditCustomer(CustomerEditView customer)
{
    Business Logic and Parameter Exception

    #region Method Code
    /*
        actual code for method
    */
    #endregion

    Check for errors and SaveChanges
}
```

OLTP Parts - Final Error Checking

Final check for any outstanding errors in the “error list.”

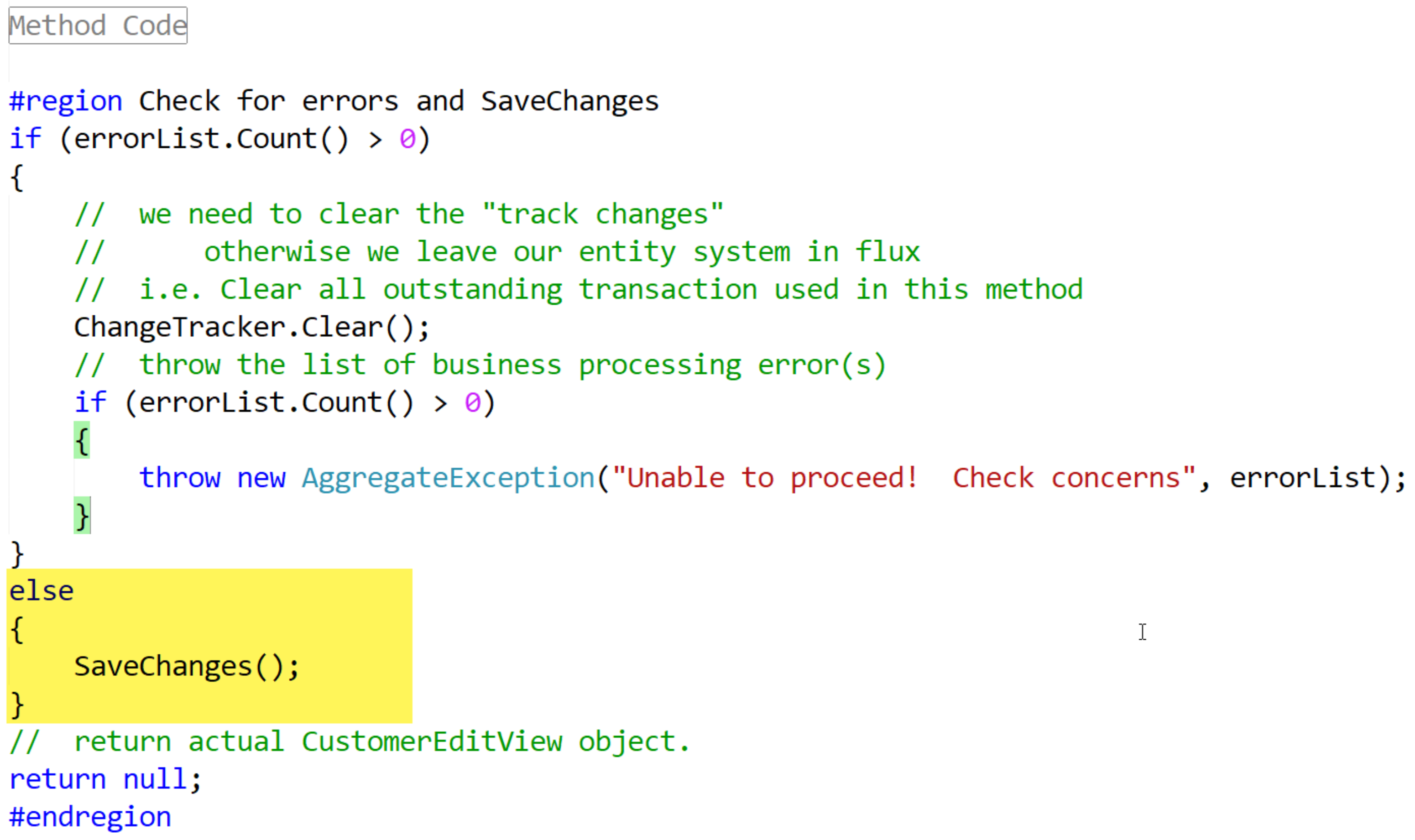


Method Code

```
#region Check for errors and SaveChanges
if (errorList.Count() > 0)
{
    //  we need to clear the "track changes"
    //      otherwise we leave our entity system in flux
    //  i.e. Clear all outstanding transaction used in this method
    ChangeTracker.Clear();
    //  throw the list of business processing error(s)
    if (errorList.Count() > 0)
    {
        throw new AggregateException("Unable to proceed!  Check concerns", errorList);
    }
}
else
{
    SaveChanges();
}
//  return actual CustomerEditView object.
return null;
#endregion
```

OLTP Parts - Saving of Data.

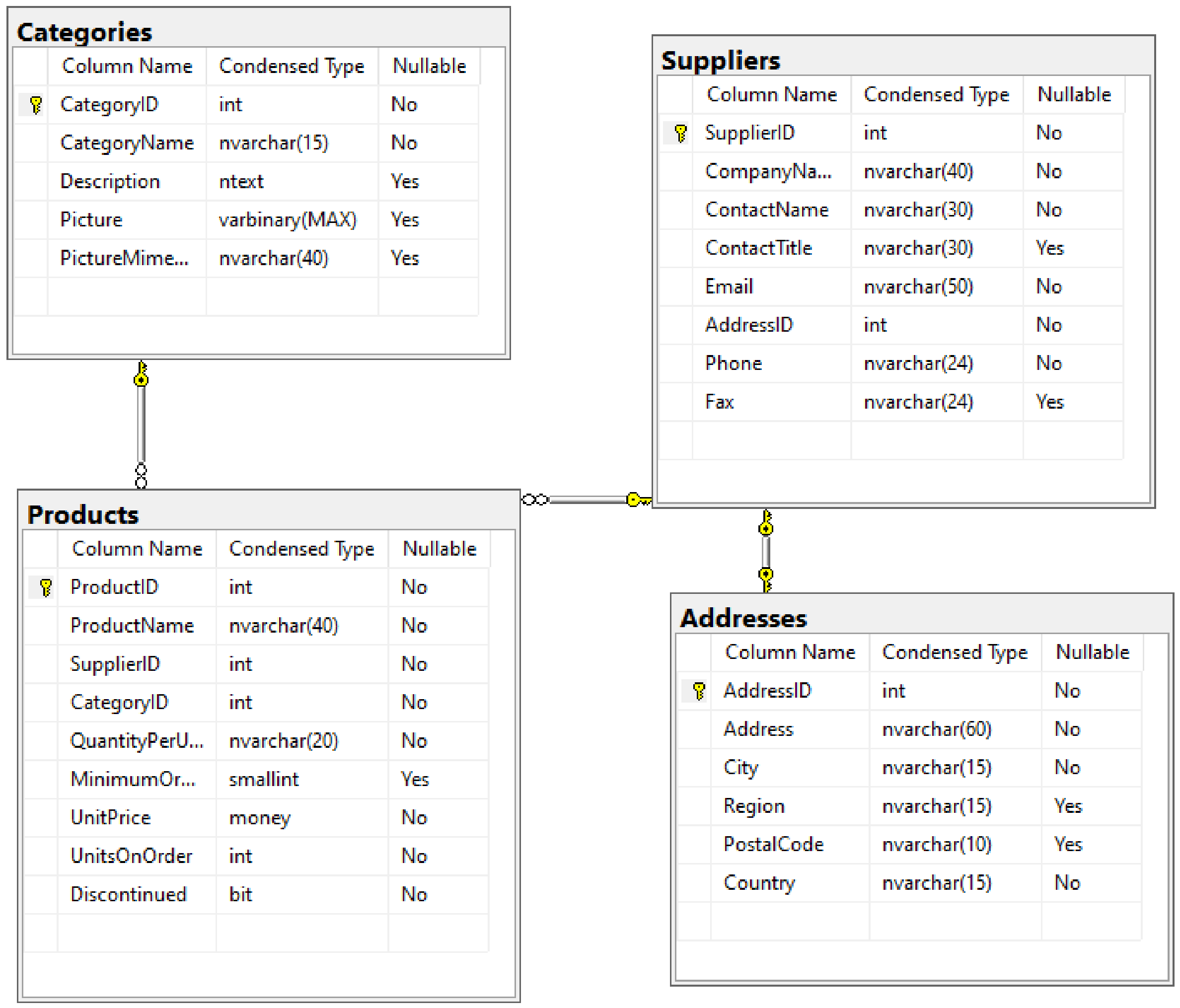
This is where we commit all changes to the database using the “Entity Framework” method called “SaveChanages().”



Method Code

```
#region Check for errors and SaveChanges
if (errorList.Count() > 0)
{
    //  we need to clear the "track changes"
    //      otherwise we leave our entity system in flux
    //  i.e. Clear all outstanding transaction used in this method
    ChangeTracker.Clear();
    //  throw the list of business processing error(s)
    if (errorList.Count() > 0)
    {
        throw new AggregateException("Unable to proceed!  Check concerns", errorList);
    }
}
else
{
    SaveChanges();
}
//  return actual CustomerEditView object.
return null;
#endregion
```

Initial Design



Categories

	Column Name	Condensed Type	Nullable
🔑	CategoryID	int	No
	CategoryName	nvarchar(15)	No
	Description	ntext	Yes
	Picture	varbinary(MAX)	Yes
	PictureMime...	nvarchar(40)	Yes

Suppliers

	Column Name	Condensed Type	Nullable
🔑	SupplierID	int	No
	CompanyNa...	nvarchar(40)	No
	ContactName	nvarchar(30)	No
	ContactTitle	nvarchar(30)	Yes
	Email	nvarchar(50)	No
	AddressID	int	No
	Phone	nvarchar(24)	No
	Fax	nvarchar(24)	Yes

Products

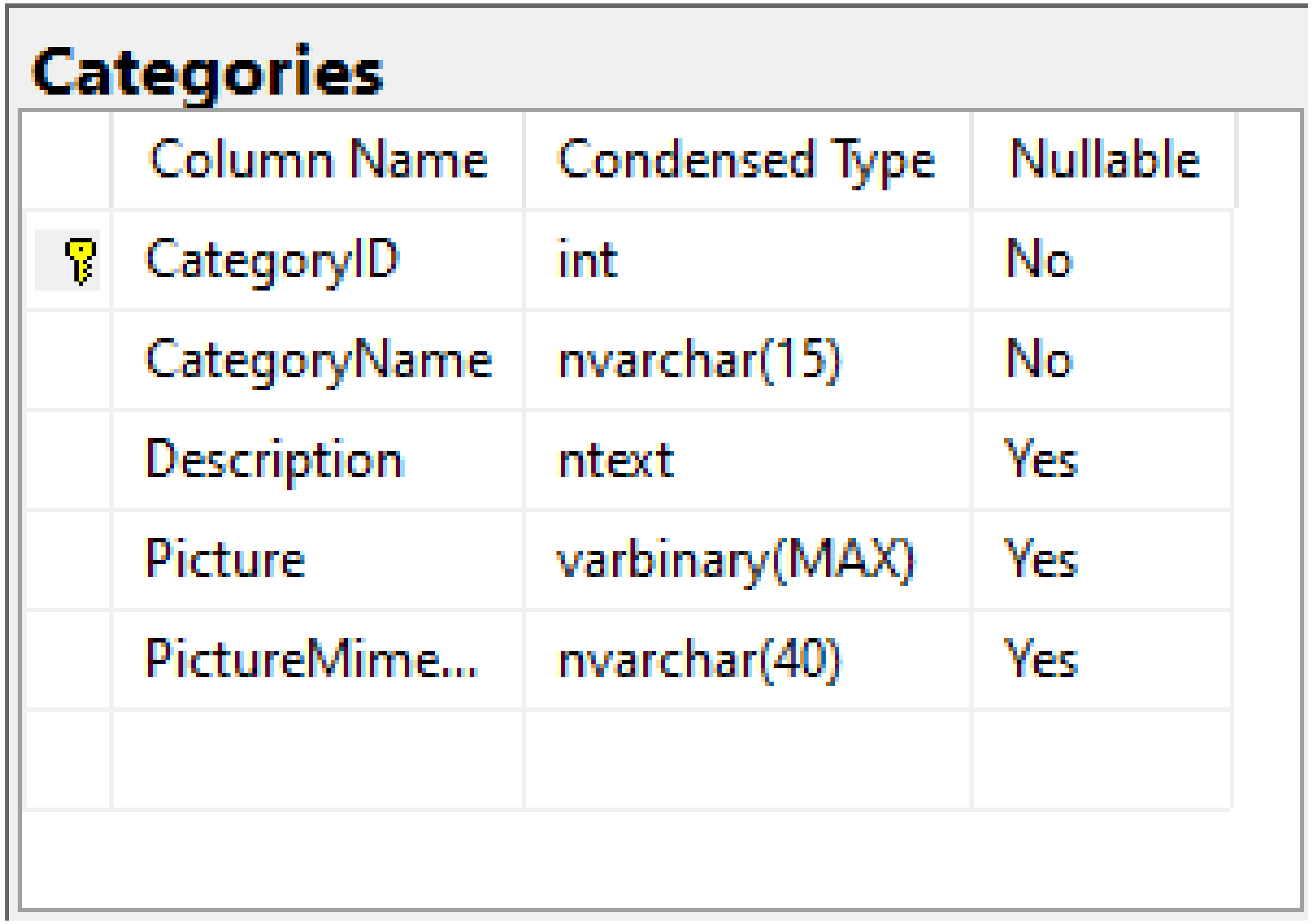
	Column Name	Condensed Type	Nullable
🔑	ProductID	int	No
	ProductName	nvarchar(40)	No
	SupplierID	int	No
	CategoryID	int	No
	QuantityPerU...	nvarchar(20)	No
	MinimumOr...	smallint	Yes
	UnitPrice	money	No
	UnitsOnOrder	int	No
	Discontinued	bit	No

Addresses

	Column Name	Condensed Type	Nullable
🔑	AddressID	int	No
	Address	nvarchar(60)	No
	City	nvarchar(15)	No
	Region	nvarchar(15)	Yes
	PostalCode	nvarchar(10)	Yes
	Country	nvarchar(15)	No

Category - Add

What are our rules for the adding a new category?



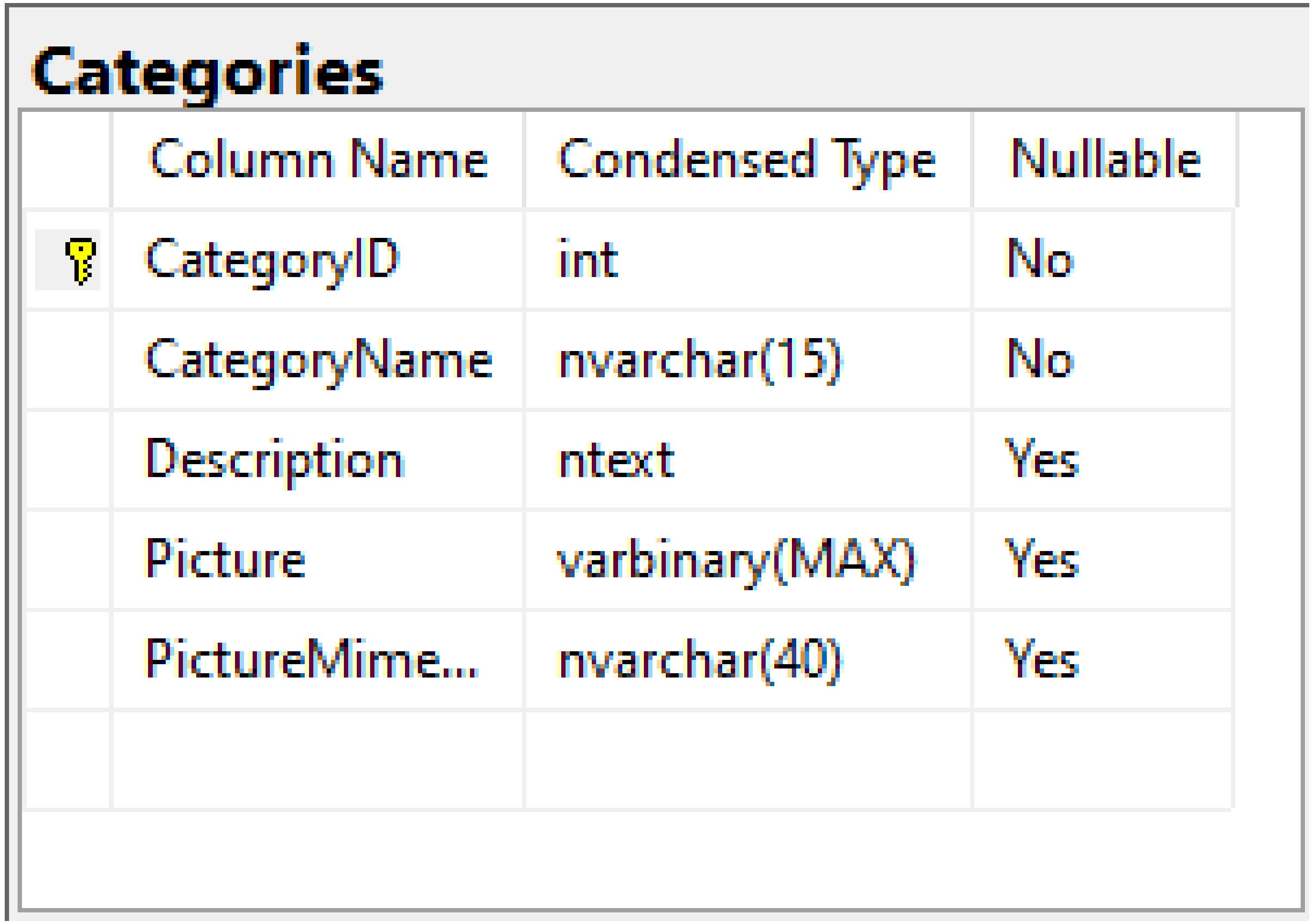
Categories

	Column Name	Condensed Type	Nullable
🔑	CategoryID	int	No
	CategoryName	nvarchar(15)	No
	Description	ntext	Yes
	Picture	varbinary(MAX)	Yes
	PictureMime...	nvarchar(40)	Yes

Category – Rules For Adding or Editing

1. “Category Name” cannot be empty.

What will our method look like? Our method will be used for adding and editing categories.

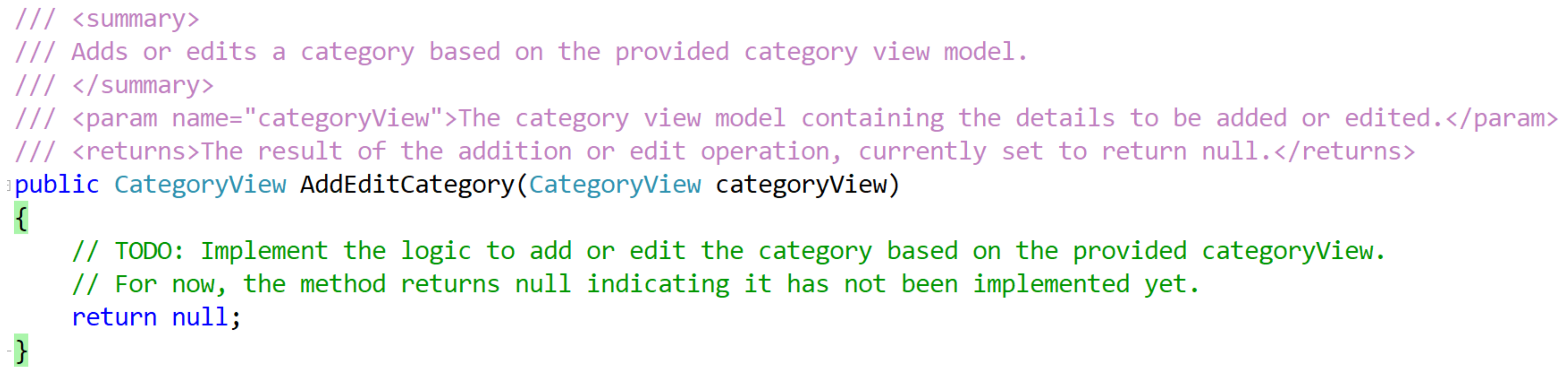


Categories

	Column Name	Condensed Type	Nullable
🔑	CategoryID	int	No
	CategoryName	nvarchar(15)	No
	Description	ntext	Yes
	Picture	varbinary(MAX)	Yes
	PictureMime...	nvarchar(40)	Yes

Category – Method Signature

- We will return a CategoryView.
- The method name will be AddEditCategory
- The parameter will accept a CategoryView.
- What will the “Unit Test” be called?

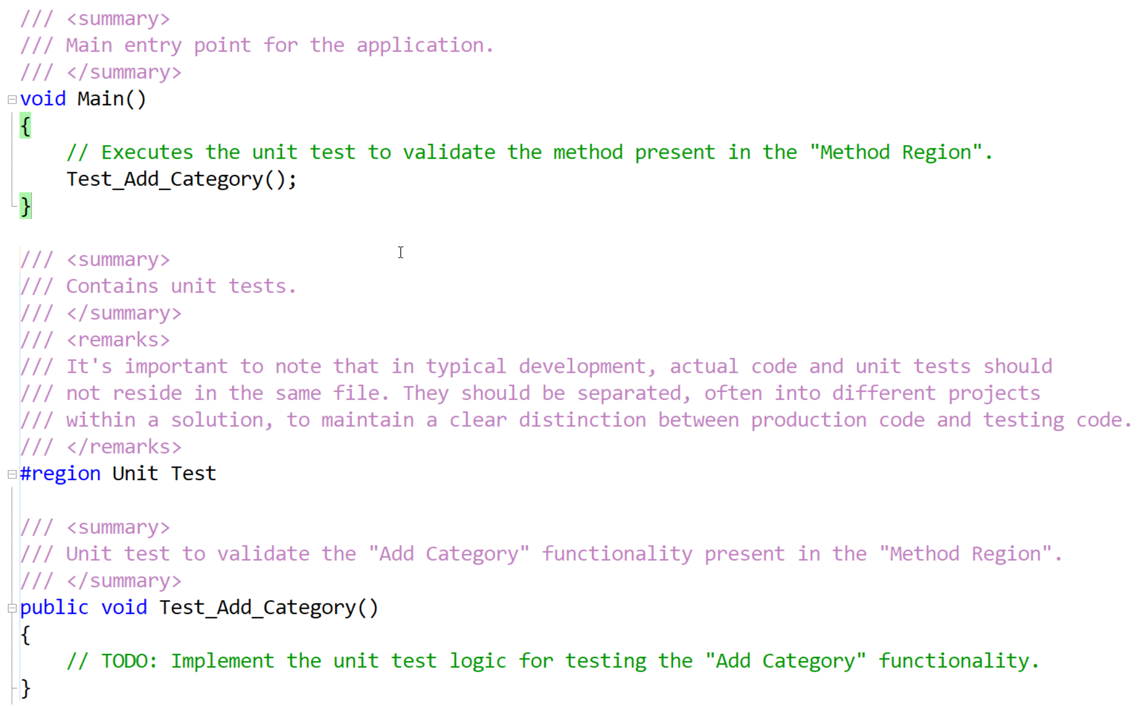


```csharp
/// <summary>
/// Adds or edits a category based on the provided category view model.
/// </summary>
/// <param name="categoryView">The category view model containing the details to be added or edited.</param>
/// <returns>The result of the addition or edit operation, currently set to return null.</returns>
public CategoryView AddEditCategory(CategoryView categoryView)
{
    // TODO: Implement the logic to add or edit the category based on the provided categoryView.
    // For now, the method returns null indicating it has not been implemented yet.
    return null;
}
```

Category – Unit Test

- Test that we have added a record to the database.
- Validate that the data we send in is the same data that is returned.
  - CategoryID is not “0.”
  - Category Name.
  - Description

- NOTE: We might have a check for an exception for “Category Name” being empty, but we will just do a high-level test at this time.



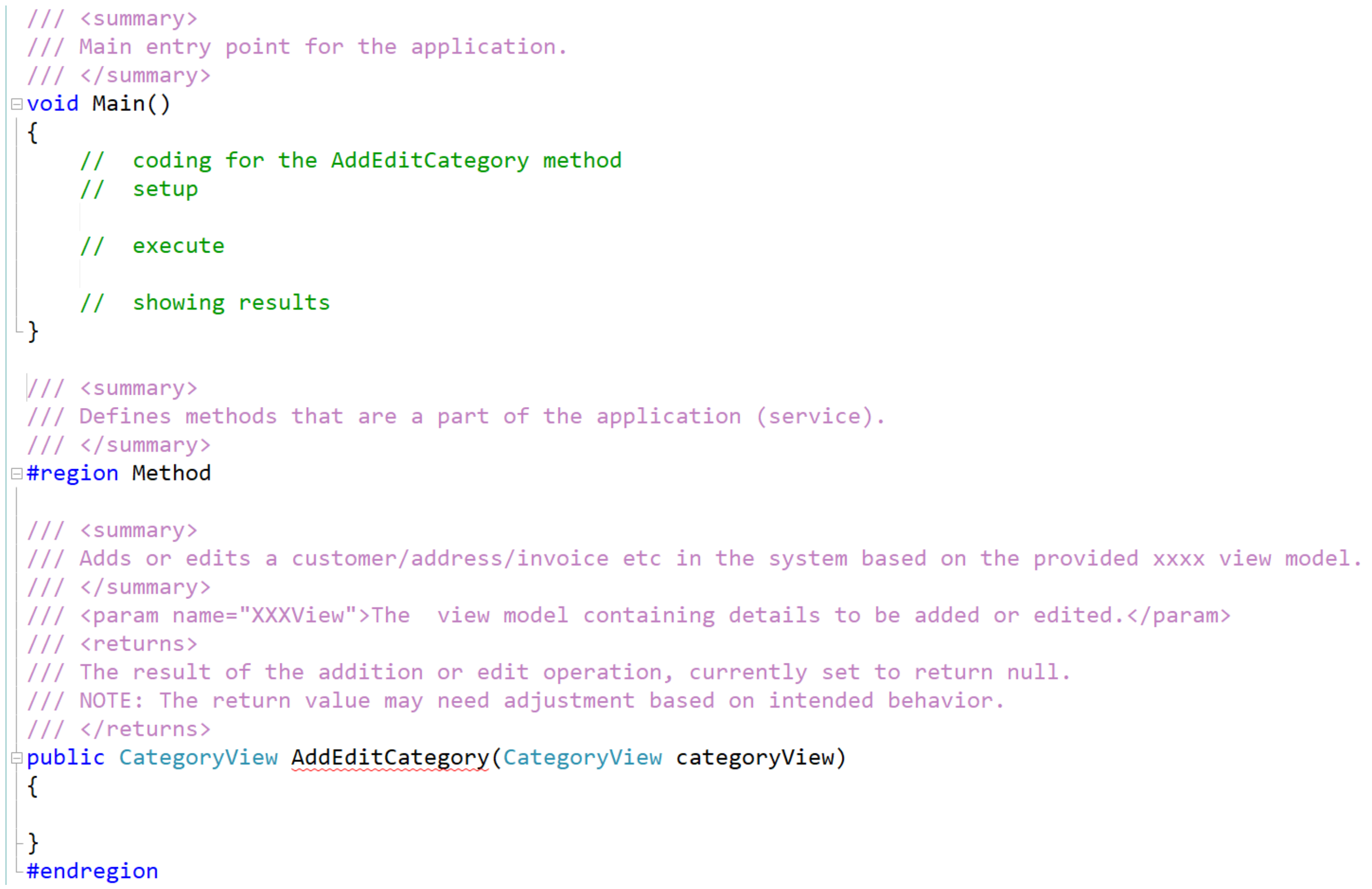
```
/// <summary>
/// Main entry point for the application.
/// </summary>
void Main()
{
    // Executes the unit test to validate the method present in the "Method Region".
    Test_Add_Category();
}

/// <summary>
/// Contains unit tests.
/// </summary>
/// <remarks>
/// It's important to note that in typical development, actual code and unit tests should
/// not reside in the same file. They should be separated, often into different projects
/// within a solution, to maintain a clear distinction between production code and testing code.
/// </remarks>
#region Unit Test

/// <summary>
/// Unit test to validate the "Add Category" functionality present in the "Method Region".
/// </summary>
public void Test_Add_Category()
{
    // TODO: Implement the unit test logic for testing the "Add Category" functionality.
}
```

Category – Driver (Main)

- If “Unit Test “is not required, all setup and execution will be done in the “Main” method.

- NOTE: Like “Unit Test,” we will do the following:

1. Setup any variables needed for the method.
2. Call the method with the assign variables.
3. Show the results .Dump()



```
/// <summary>
/// Main entry point for the application.
/// </summary>
void Main()
{
    //  coding for the AddEditCategory method
    //  setup

    //  execute

    //  showing results
}

/// <summary>
/// Defines methods that are a part of the application (service).
/// </summary>
#region Method

/// <summary>
/// Adds or edits a customer/address/invoice etc in the system based on the provided xxxx view model.
/// </summary>
/// <param name="XXXView">The  view model containing details to be added or edited.</param>
/// <returns>
/// The result of the addition or edit operation, currently set to return null.
/// NOTE: The return value may need adjustment based on intended behavior.
/// </returns>
public CategoryView AddEditCategory(CategoryView categoryView)
{

}
#endregion
```

Getting Started

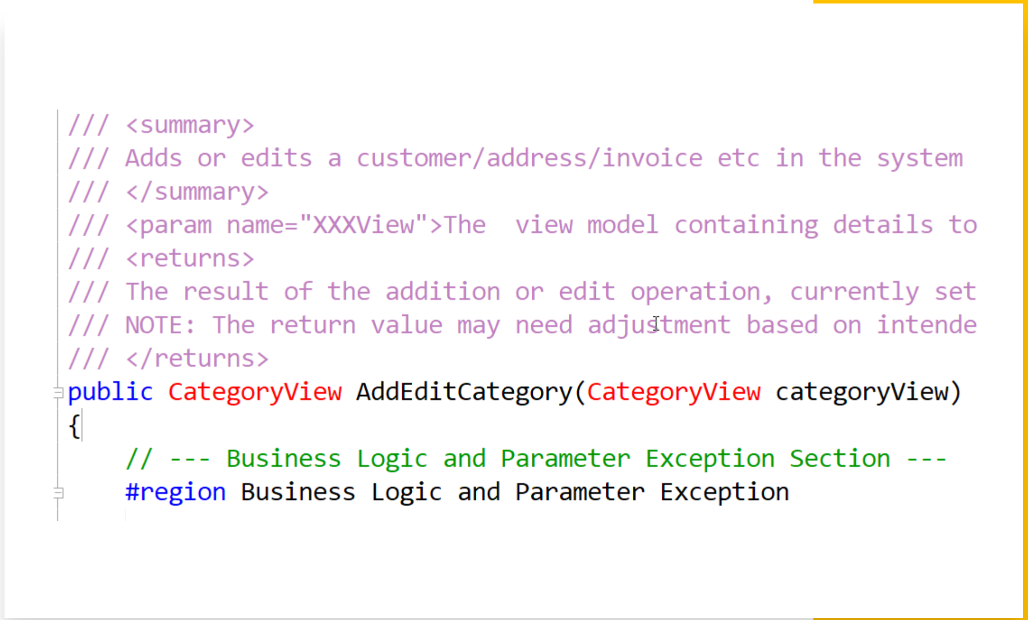
1. Start a new "C# Program" LINQ query.
2. Copy the OLTP Template into the LINQ query.
3. Set the "Connection" to Westwind-Entity.
4. Save the LINQ file as AddEditCategory.

Building Add/Edit Category

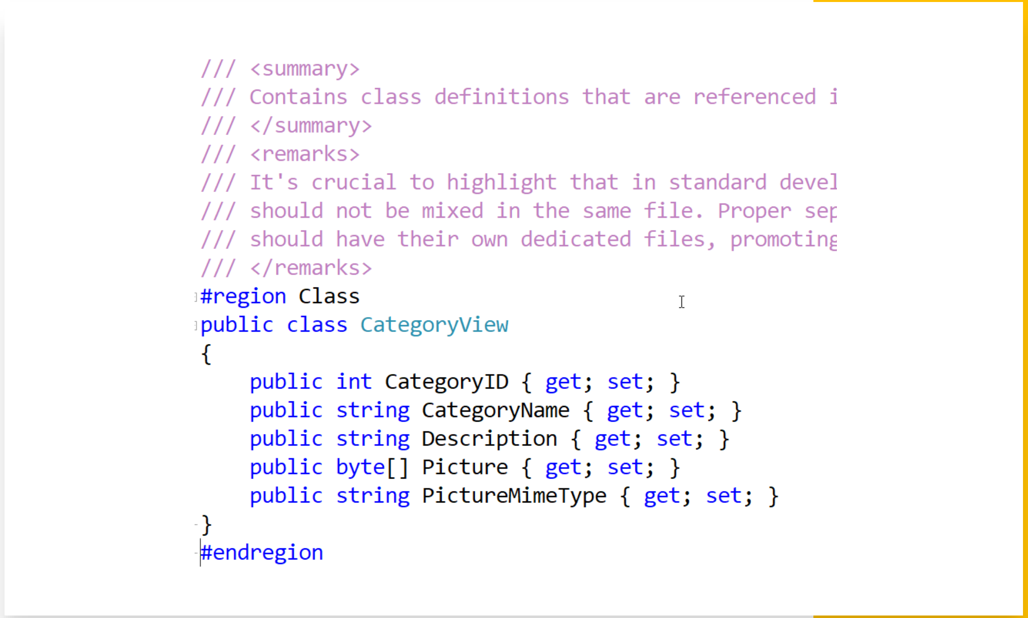
- Creating the method
- Add the business rules and validation
- Add code for adding/editing a entity
- Add test for “Adding a Category”
- Add test for “Editing a Category”

Creating the Method

- Let’s refactor our template method signature to our “Add/Edit” signature.
- NOTE: We are getting a red error on the CategoryView not being found. We will need to add the CategoryView to the class region at the bottom of the LINQ file.



```
/// <summary>
/// Adds or edits a customer/address/invoice etc in the system
/// </summary>
/// <param name="XXXView">The  view model containing details to
/// <returns>
/// The result of the addition or edit operation, currently set
/// NOTE: The return value may need adjustment based on intende
/// </returns>
public CategoryView AddEditCategory(CategoryView categoryView)
{
    // --- Business Logic and Parameter Exception Section ---
    #region Business Logic and Parameter Exception
```



```
/// <summary>
/// Contains class definitions that are referenced i
/// </summary>
/// <remarks>
/// It's crucial to highlight that in standard devel
/// should not be mixed in the same file. Proper sep
/// should have their own dedicated files, promoting
/// </remarks>
#region Class
public class CategoryView
{
    public int CategoryID { get; set; }
    public string CategoryName { get; set; }
    public string Description { get; set; }
    public byte[] Picture { get; set; }
    public string PictureMimeType { get; set; }
}
#endregion
```

Business Rules

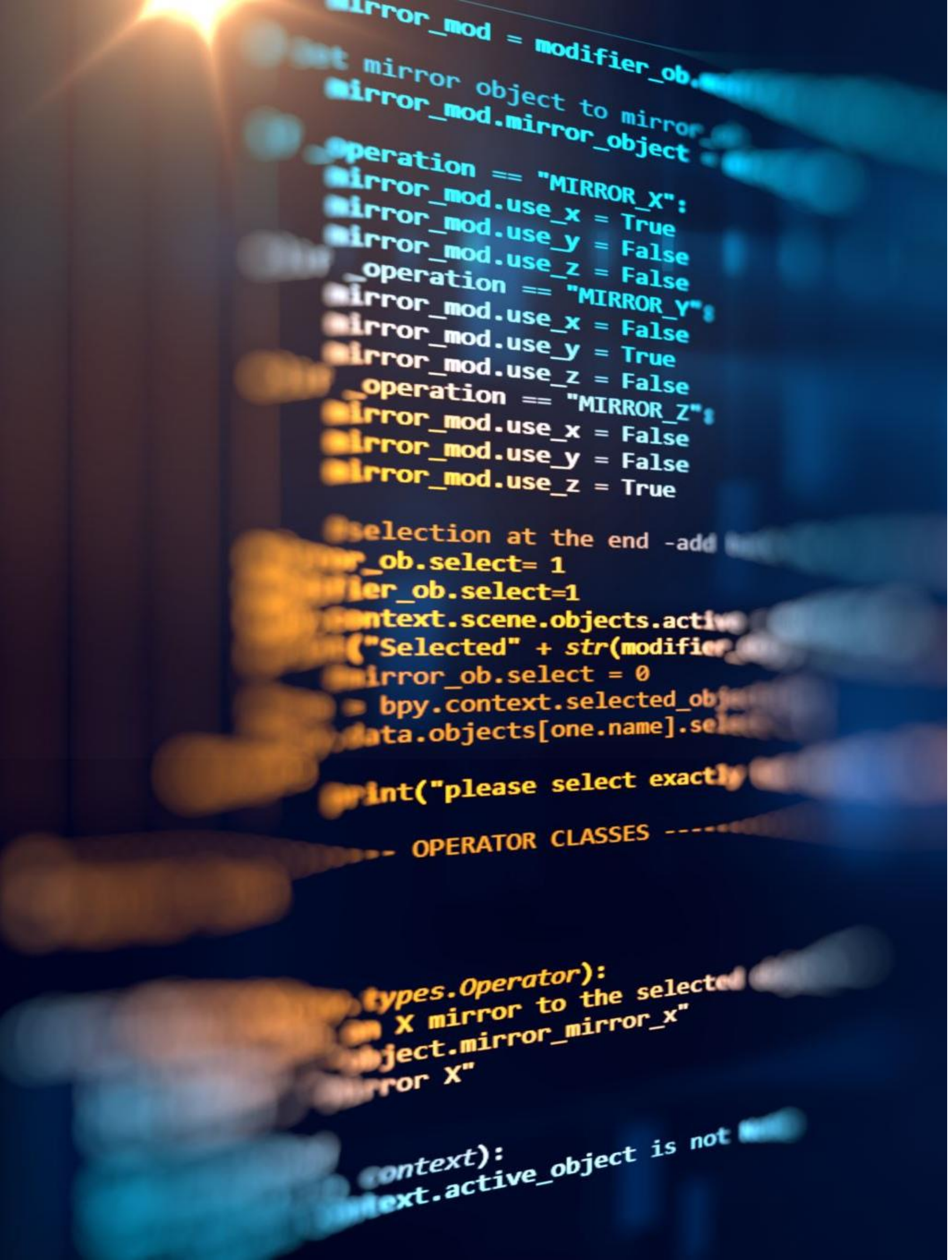
What are our known business rules?

Business Rules and Validation

- Rule: “Category Name” cannot be empty.

```
// --- Business Logic and Parameter Exception Section ---
#region Business Logic and Parameter Exception

// List initialization to capture potential errors during processing.
List<Exception> errorList = new List<Exception>();

// All business rules are placed here.
// Rule:    “Category Name” cannot be empty.

// The logic to validate incoming parameters goes here.
if (string.IsNullOrWhiteSpace(categoryView.CategoryName))
{
    throw new Exception("Category name is required and cannot be empty");
}
#endregion
```

Method



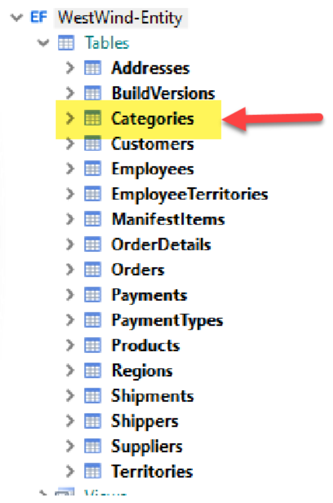
Overview of Adding or Editing Data

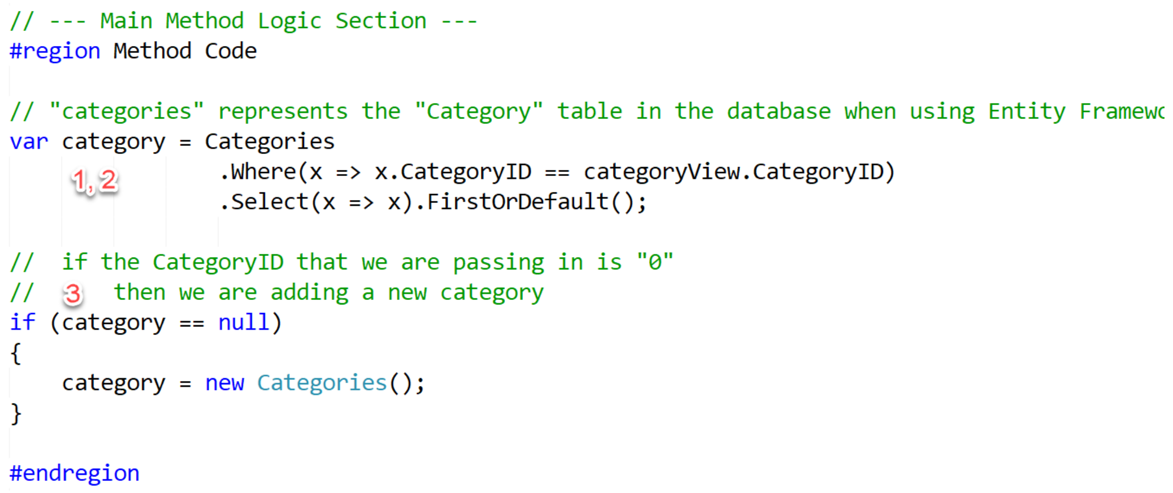
- Attempt to retrieve the existing entity from the database using the “Primary Key” value.
- If the entity exists, then we are editing the existing entity.
- Otherwise, we are creating a new entity, which we will use to add a new record to the database.
- Update the entity with data from the “View Model.”
- Handling Adding vs Editing entity for “SaveChanges”
- Do a final check for any outstanding errors in the “Error List.”
- Commit the changes to the database.

Working with the Categories Entity

1. Attempt to retrieve the existing entity from the database using the "Primary Key" value.
2. If the entity exists, then we are editing the existing entity.
3. Otherwise, we are creating a new entity, which we will use to add a new record to the database.

NOTE: The category is an entity of the "Categories" collection.

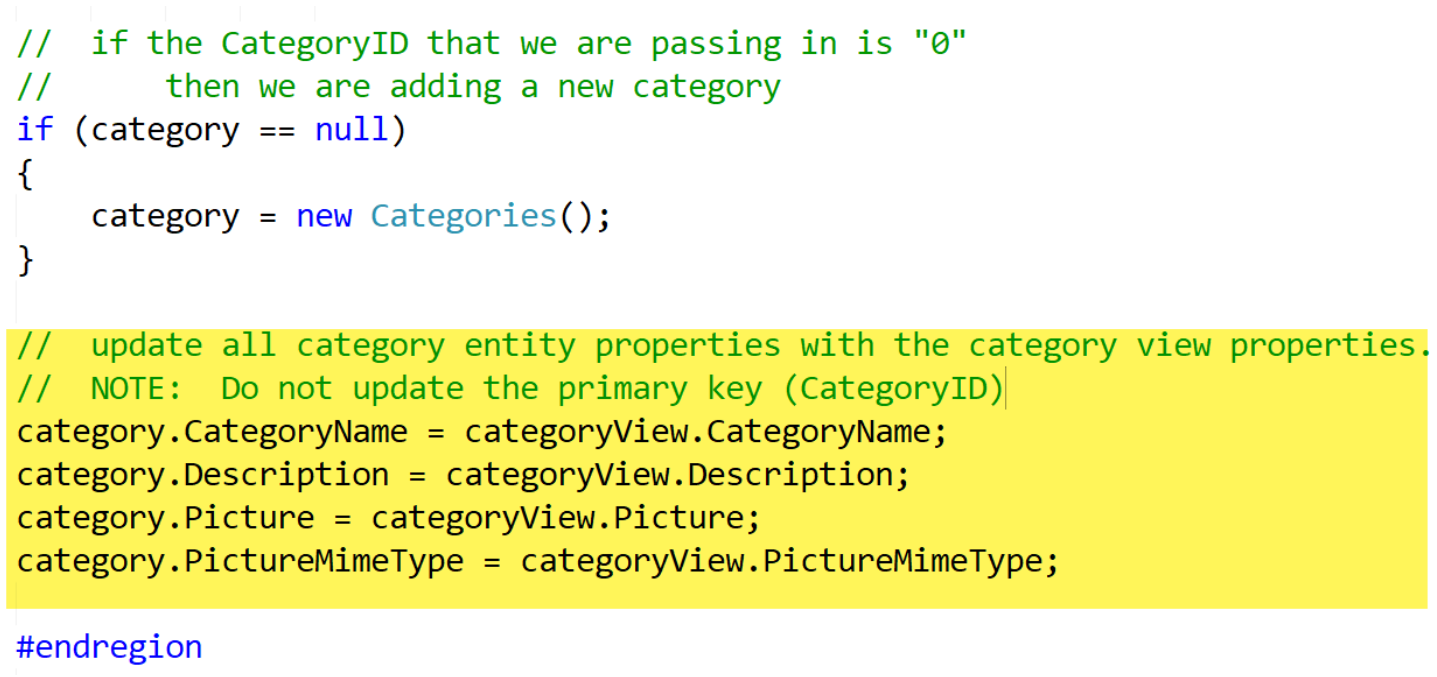




```
// --- Main Method Logic Section ---
#region Method Code

// "categories" represents the "Category" table in the database when using Entity Framew
var category = Categories
    1, 2        .Where(x => x.CategoryID == categoryView.CategoryID)
                .Select(x => x).FirstOrDefault();

//  if the CategoryID that we are passing in is "0"
//  3  then we are adding a new category
if (category == null)
{
    category = new Categories();
}

#endregion
```

Update the entity with data from the “View Model.”

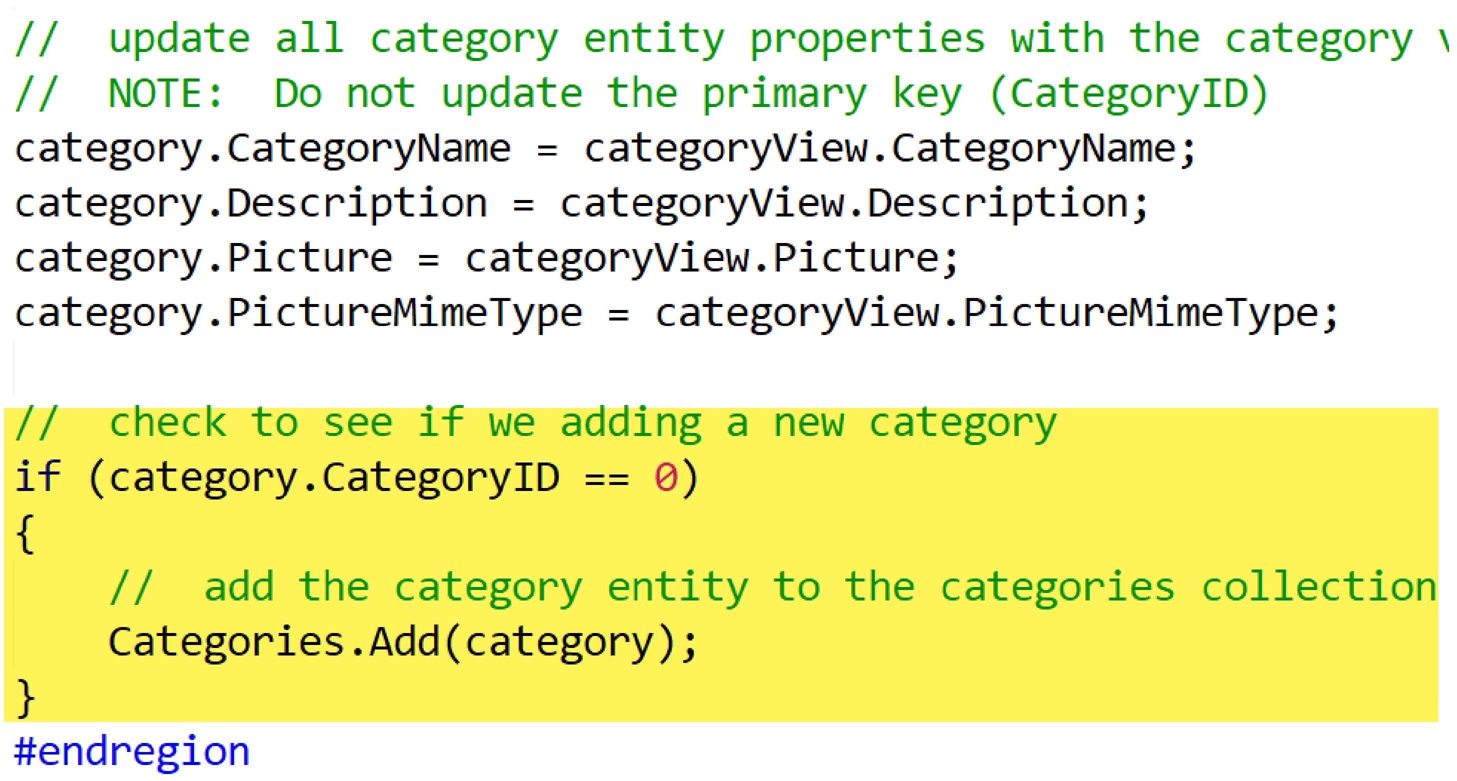
- For all properties except for the “Primary Key,” the entity must be updated from the “View Model.”



```
//  if the CategoryID that we are passing in is "0"
//      then we are adding a new category
if (category == null)
{
    category = new Categories();
}

//  update all category entity properties with the category view properties.
//  NOTE:  Do not update the primary key (CategoryID)
category.CategoryName = categoryView.CategoryName;
category.Description = categoryView.Description;
category.Picture = categoryView.Picture;
category.PictureMimeType = categoryView.PictureMimeType;

#endregion
```

Handling Adding vs Editing

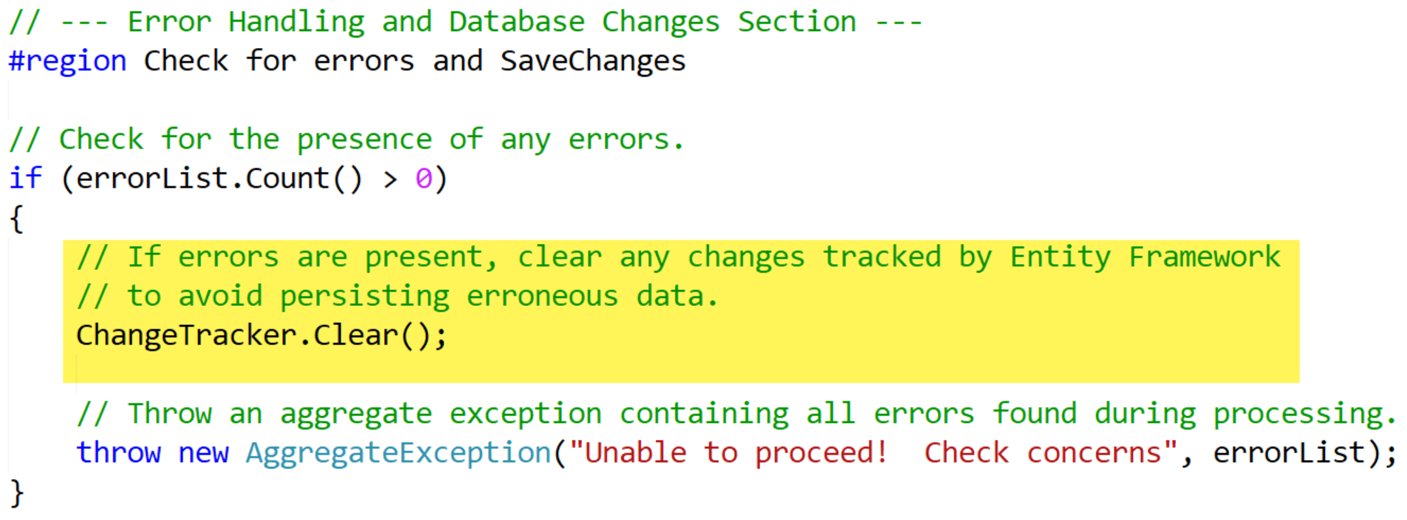
- When finding an existing entity from the “Categories” table, we are editing the actual “Category” entity that represents the data within the database. NOTE: We do not have to do anything else if code to handle the update.
- When we do not have an existing entity, we create a new “Category” entity to be added to the “Categories” collection. However, at this moment, the “Categories” is not aware that this entity exists. Because of this, we must add the new “Category” entity to the “Categories” collections.



```
//  update all category entity properties with the category
//  NOTE:  Do not update the primary key (CategoryID)
category.CategoryName = categoryView.CategoryName;
category.Description = categoryView.Description;
category.Picture = categoryView.Picture;
category.PictureMimeType = categoryView.PictureMimeType;

//  check to see if we adding a new category
if (category.CategoryID == 0)
{
    //  add the category entity to the categories collection
    Categories.Add(category);
}
#endregion
```

Final Check for any Outstanding Errors!

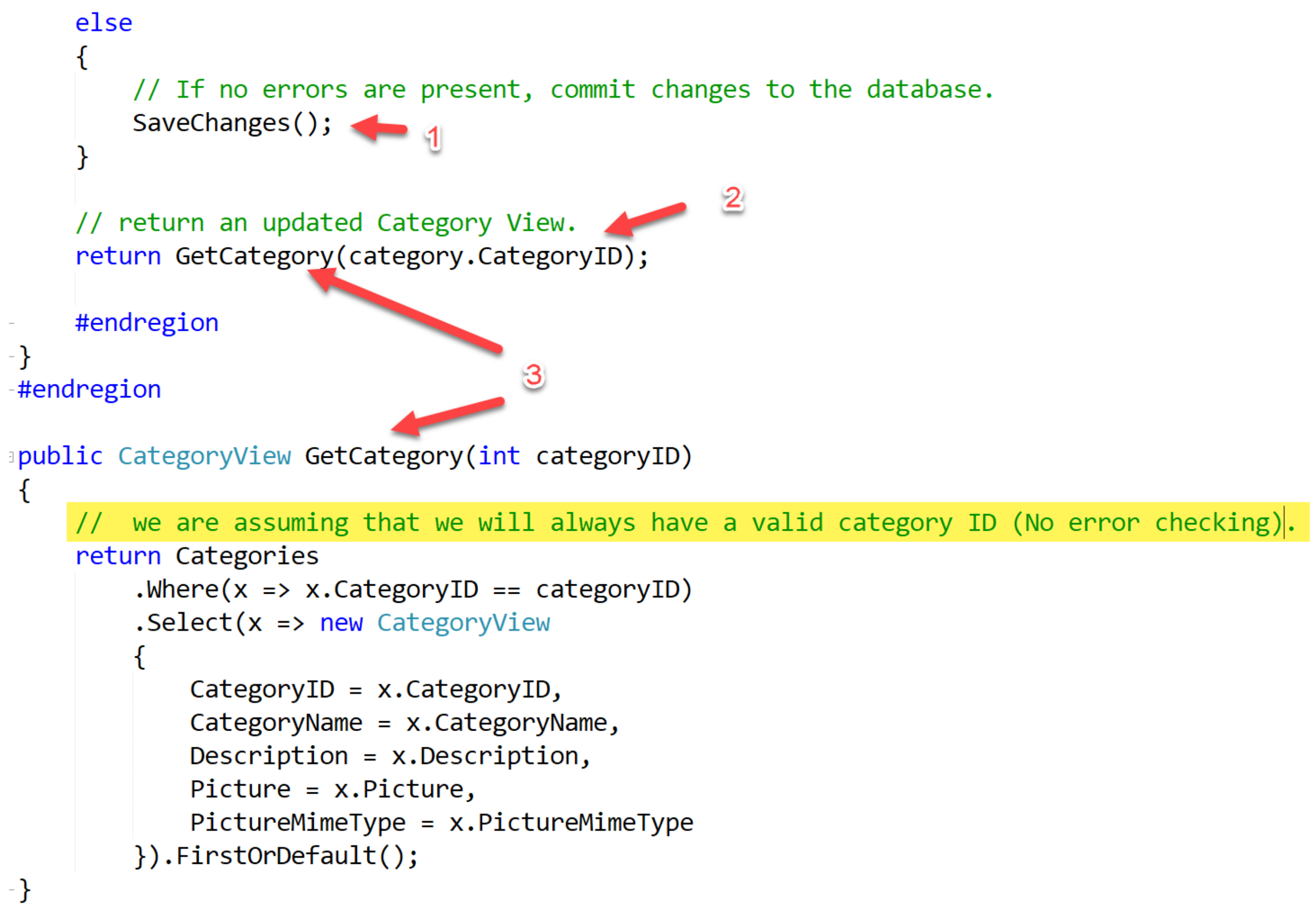
- Do a final check for any outstanding errors in the "Error List."
- NOTE: During the coding of your "Main Method Logic Section," you might find issues that you wish for the "End User" to be aware of. You might add these errors to the "Error List."
- WARNING: You must ensure that you include the "ChangeTracker.Clear()" when processing your "Error List."



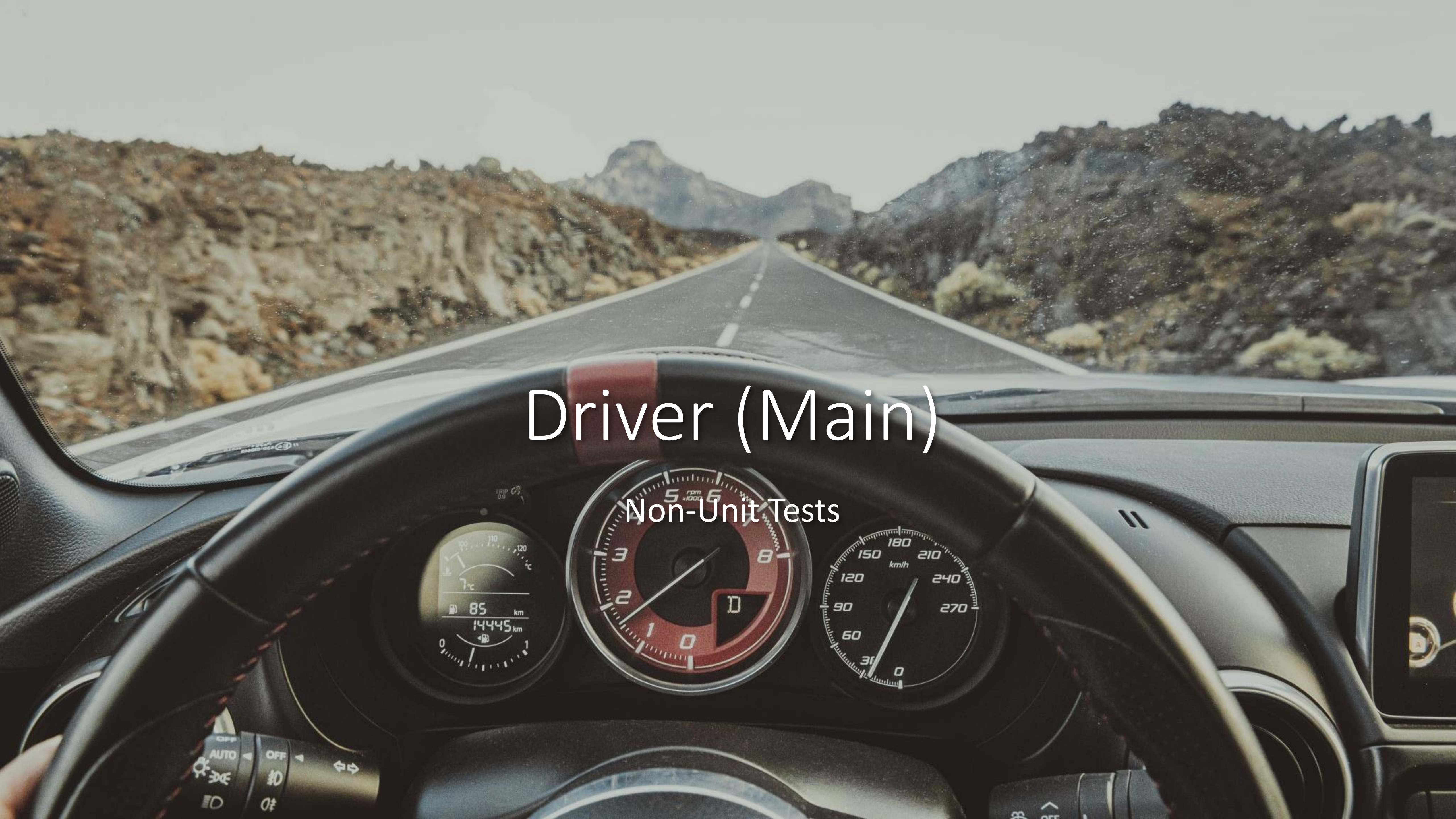
```
// --- Error Handling and Database Changes Section ---
#region Check for errors and SaveChanges

// Check for the presence of any errors.
if (errorList.Count() > 0)
{
    // If errors are present, clear any changes tracked by Entity Framework
    // to avoid persisting erroneous data.
    ChangeTracker.Clear();

    // Throw an aggregate exception containing all errors found during processing.
    throw new AggregateException("Unable to proceed!  Check concerns", errorList);
}
```

Commit Changes & Returning Results

- Commit the changes to the database.
  1. When you create a new entity in Entity Framework and call SaveChanges(), the entity's primary key will be updated if it is set as an auto-incrementing key in the database.
  2. After SaveChanges(), category.CategoryID will have the value assigned by the database.
  3. We will then use the "GetCategory" method to return an updated "CategoryView" for use on our"Web Page". This is because we might have business rules that are only run during either the "GetCategory" or from the database store procedure or trigger.

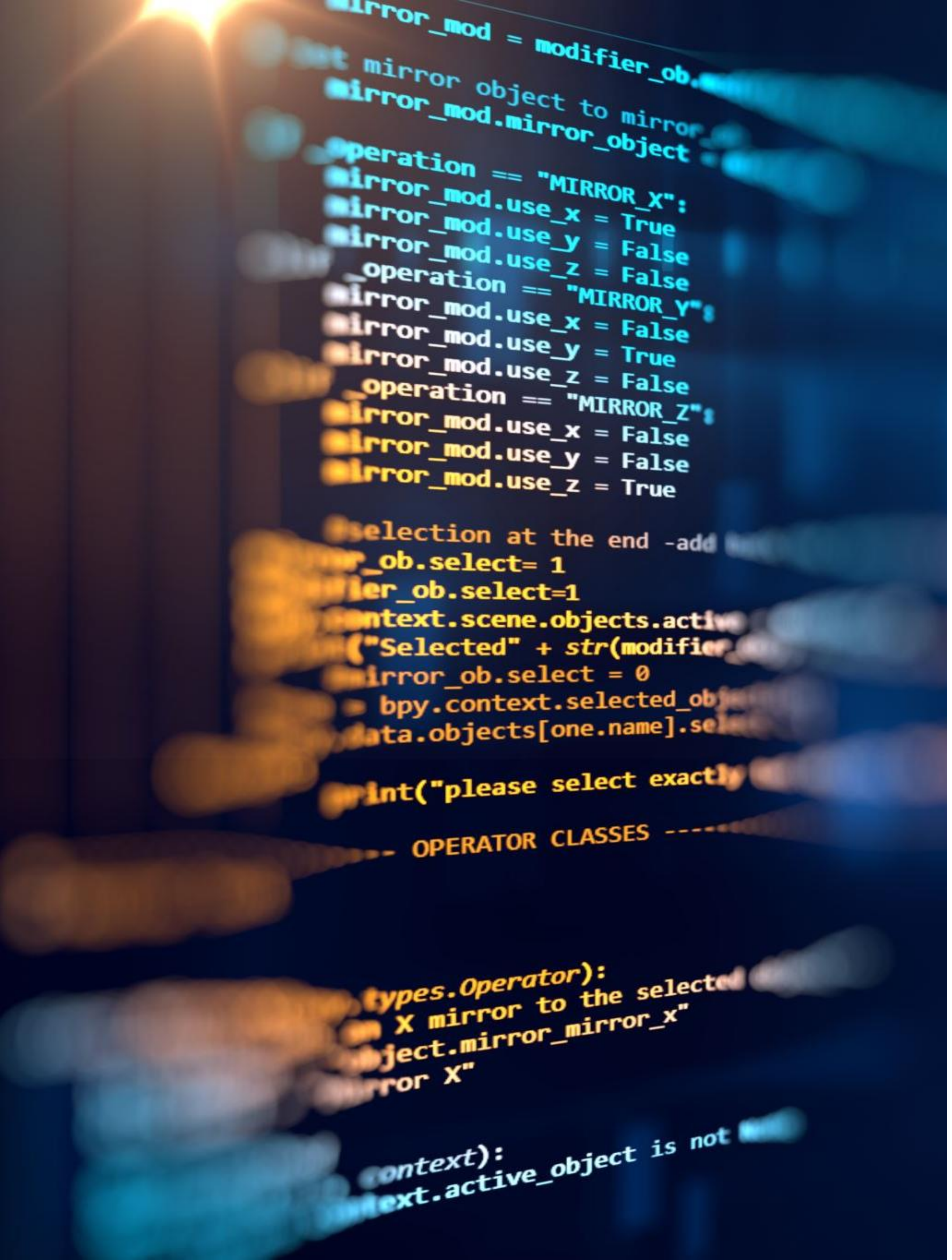


```
        else
        {
            // If no errors are present, commit changes to the database.
            SaveChanges();
        }

        // return an updated Category View.
        return GetCategory(category.CategoryID);

        #endregion
    }
    #endregion

    public CategoryView GetCategory(int categoryID)
    {
        //  we are assuming that we will always have a valid category ID (No error checking).
        return Categories
            .Where(x => x.CategoryID == categoryID)
            .Select(x => new CategoryView
            {
                CategoryID = x.CategoryID,
                CategoryName = x.CategoryName,
                Description = x.Description,
                Picture = x.Picture,
                PictureMimeType = x.PictureMimeType
            }).FirstOrDefault();
    }
```



Driver (Main)

Non-Unit Tests

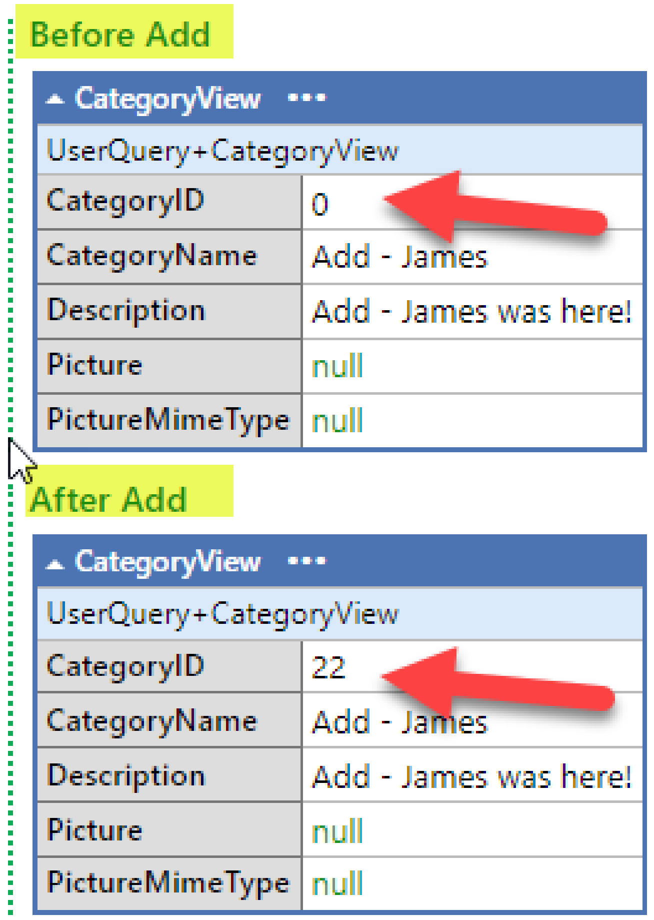


Using Main() Method

- Coding in the Main() methods is very much like coding in your Web Page. This might be things such as saving data, retrieving invoice and their invoice lines, etc.
- Like doing Unit Tests, we might ask you just to code in the Main() method to show that your method works.

Using Dump for Validation (Adding)

```
void Main()
{
    //  coding for the AddEditCategory method
    //  setup Add Category
    //  before action (Add)
    CategoryView beforeAdd = new CategoryView();
    beforeAdd.CategoryName = "Add - James";
    beforeAdd.Description = "Add - James was here!";

    //  showing results
    beforeAdd.Dump("Before Add");

    //  execute
    CategoryView afterAdd = AddEditCategory(beforeAdd);

    //  after action (Add)
    //  showing results
    afterAdd.Dump("After Add");
```



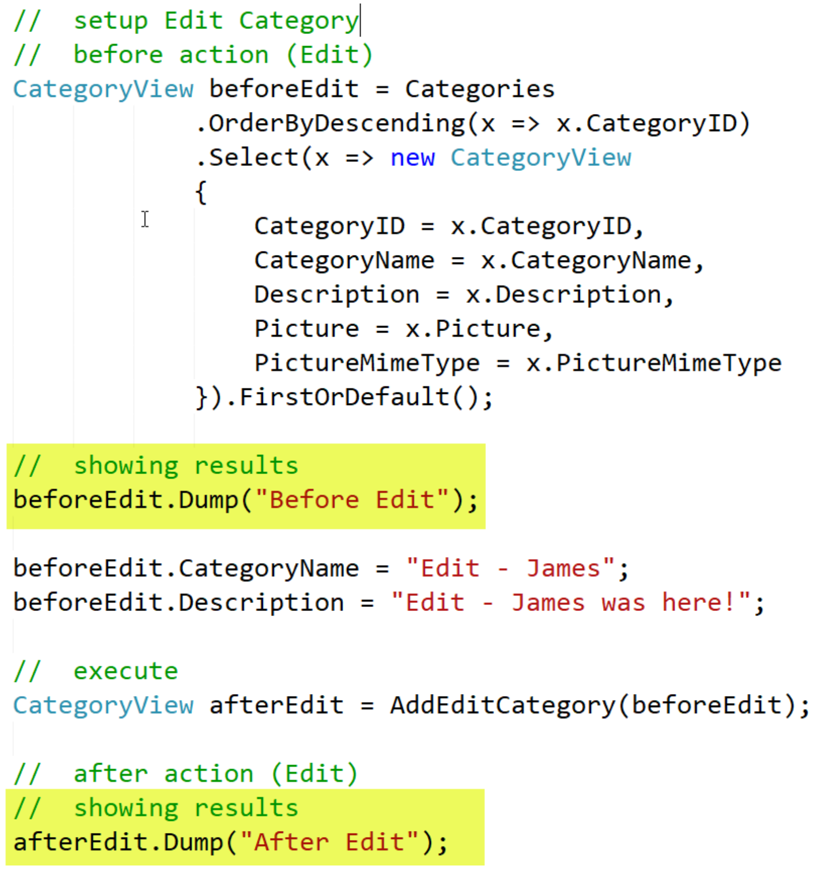
Before Add

▲ CategoryView •••	
UserQuery+CategoryView	
CategoryID	0
CategoryName	Add - James
Description	Add - James was here!
Picture	null
PictureMimeType	null

After Add

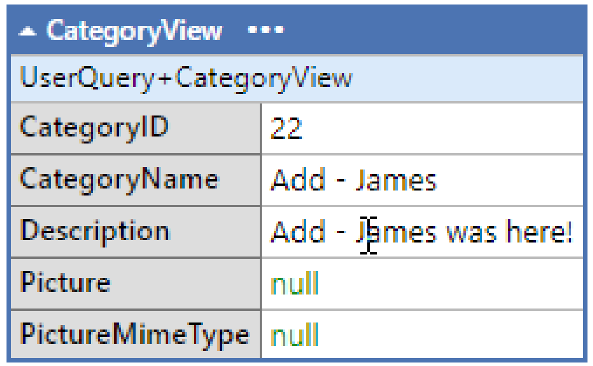
▲ CategoryView •••	
UserQuery+CategoryView	
CategoryID	22
CategoryName	Add - James
Description	Add - James was here!
Picture	null
PictureMimeType	null

Using Dump for Validation (Editing)



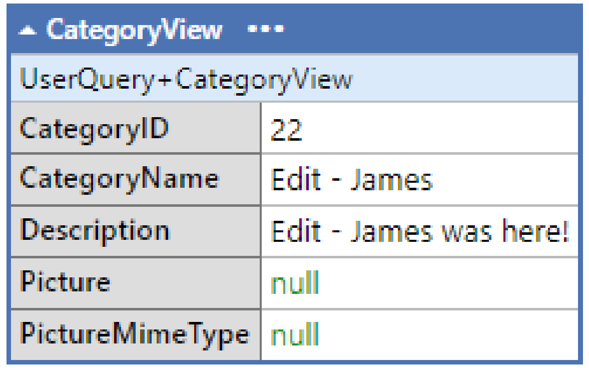
```
//  setup Edit Category
//  before action (Edit)
CategoryView beforeEdit = Categories
            .OrderByDescending(x => x.CategoryID)
            .Select(x => new CategoryView
            {
                CategoryID = x.CategoryID,
                CategoryName = x.CategoryName,
                Description = x.Description,
                Picture = x.Picture,
                PictureMimeType = x.PictureMimeType
            }).FirstOrDefault();

//  showing results
beforeEdit.Dump("Before Edit");

beforeEdit.CategoryName = "Edit - James";
beforeEdit.Description = "Edit - James was here!";

//  execute
CategoryView afterEdit = AddEditCategory(beforeEdit);

//  after action (Edit)
//  showing results
afterEdit.Dump("After Edit");
```

Before Edit

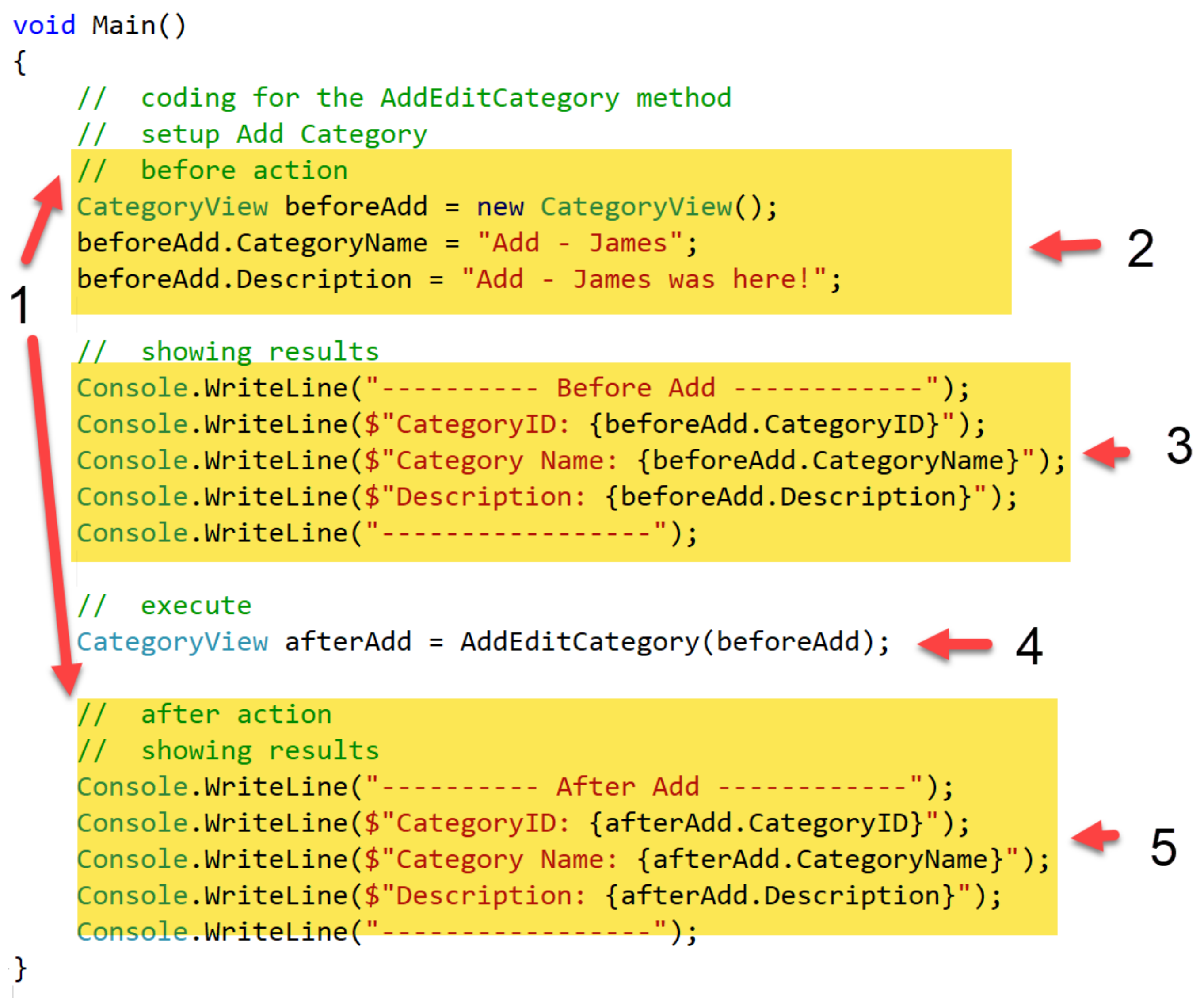


```
▲ CategoryView •••
UserQuery+CategoryView
CategoryID       22
CategoryName     Add - James
Description      Add - James was here!
Picture          null
PictureMimeType  null
```

After Edit

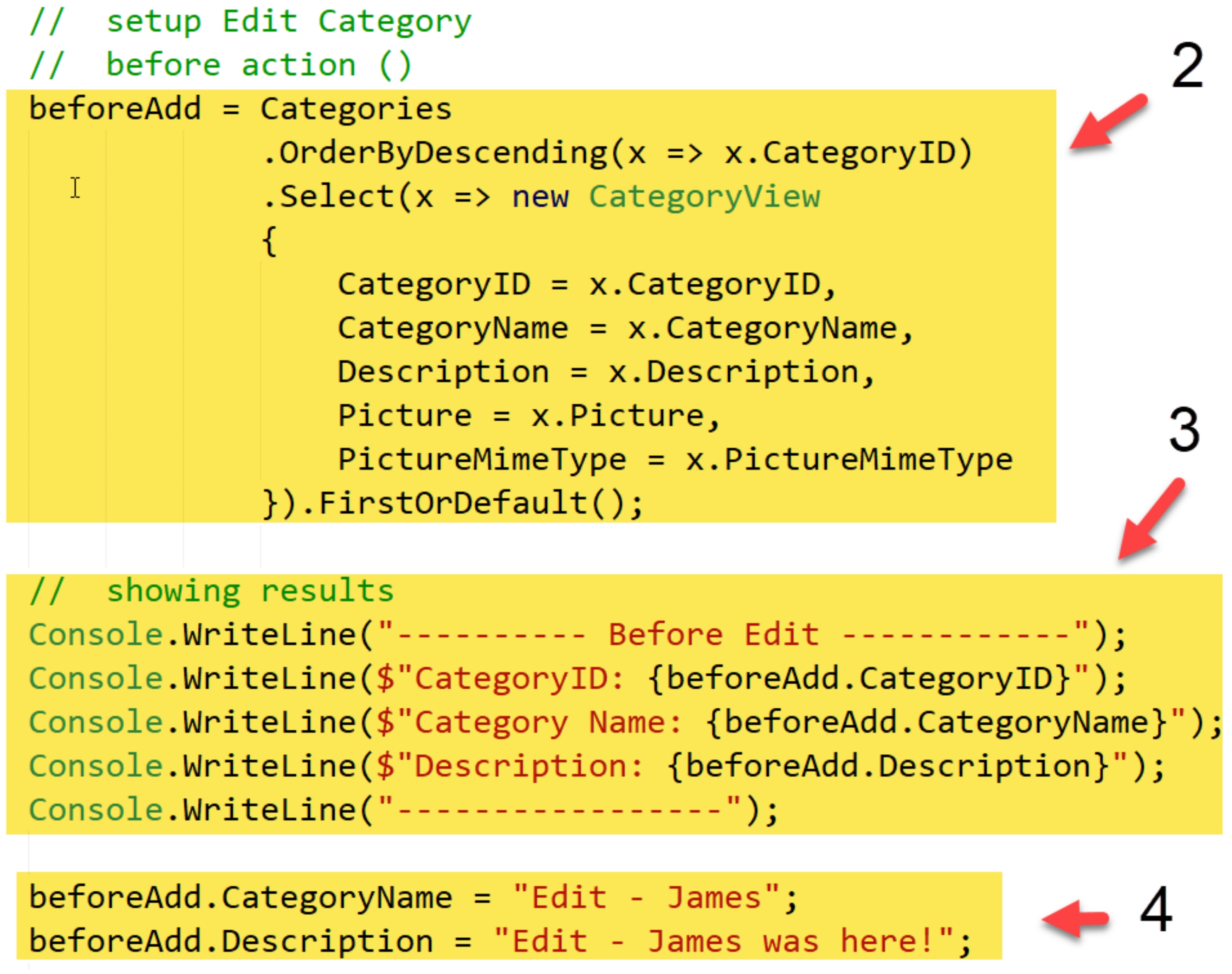


```
▲ CategoryView •••
UserQuery+CategoryView
CategoryID       22
CategoryName     Edit - James
Description      Edit - James was here!
Picture          null
PictureMimeType  null
```

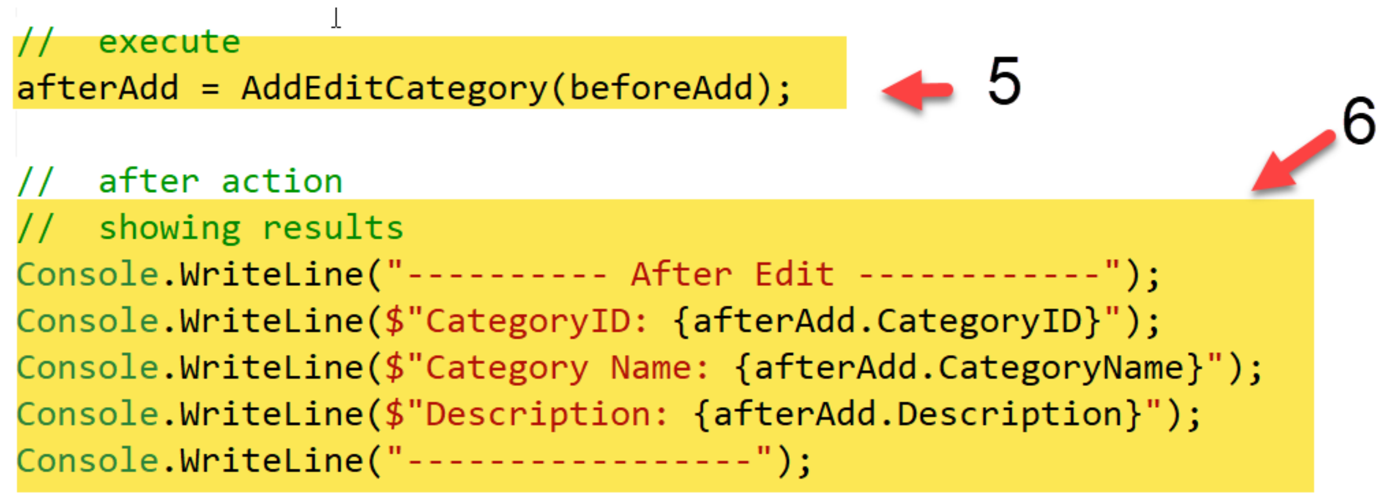


```
void Main()
{
    //  coding for the AddEditCategory method
    //  setup Add Category
    //  before action
    CategoryView beforeAdd = new CategoryView();
    beforeAdd.CategoryName = "Add - James";
    beforeAdd.Description = "Add - James was here!";

    //  showing results
    Console.WriteLine("---------- Before Add ------------");
    Console.WriteLine($"CategoryID: {beforeAdd.CategoryID}");
    Console.WriteLine($"Category Name: {beforeAdd.CategoryName}");
    Console.WriteLine($"Description: {beforeAdd.Description}");
    Console.WriteLine("-----------------");

    //  execute
    CategoryView afterAdd = AddEditCategory(beforeAdd);

    //  after action
    //  showing results
    Console.WriteLine("---------- After Add ------------");
    Console.WriteLine($"CategoryID: {afterAdd.CategoryID}");
    Console.WriteLine($"Category Name: {afterAdd.CategoryName}");
    Console.WriteLine($"Description: {afterAdd.Description}");
    Console.WriteLine("-----------------");
}
```

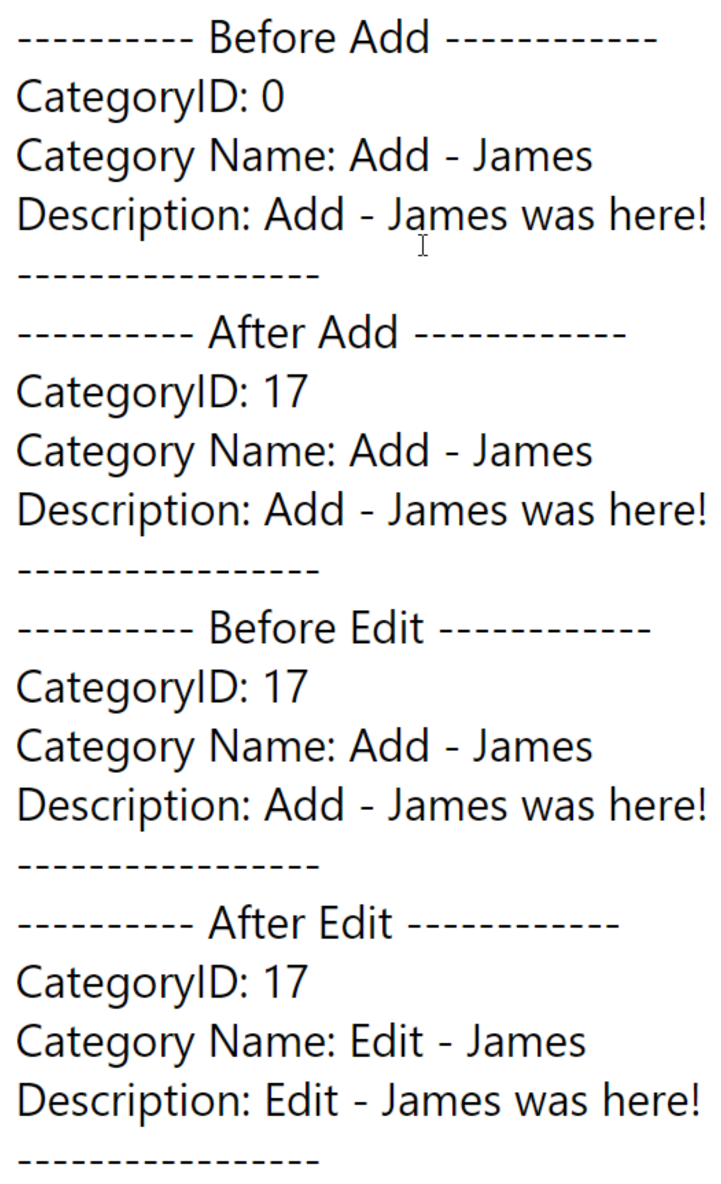
```
---------- Before Add ------------
CategoryID: 0
Category Name: Add - James
Description: Add - James was here!
------------------
---------- After Add ------------
CategoryID: 15
Category Name: Add - James
Description: Add - James was here!
------------------
```



```
//  setup Edit Category
//  before action ()
beforeAdd = Categories
            .OrderByDescending(x => x.CategoryID)
            .Select(x => new CategoryView
            {
                CategoryID = x.CategoryID,
                CategoryName = x.CategoryName,
                Description = x.Description,
                Picture = x.Picture,
                PictureMimeType = x.PictureMimeType
            }).FirstOrDefault();

//  showing results
Console.WriteLine("---------- Before Edit ------------");
Console.WriteLine($"CategoryID: {beforeAdd.CategoryID}");
Console.WriteLine($"Category Name: {beforeAdd.CategoryName}");
Console.WriteLine($"Description: {beforeAdd.Description}");
Console.WriteLine("----------------");

beforeAdd.CategoryName = "Edit - James";
beforeAdd.Description = "Edit - James was here!";
```



```
//  execute
afterAdd = AddEditCategory(beforeAdd);

//  after action
//  showing results
Console.WriteLine("---------- After Edit ------------");
Console.WriteLine($"CategoryID: {afterAdd.CategoryID}");
Console.WriteLine($"Category Name: {afterAdd.CategoryName}");
Console.WriteLine($"Description: {afterAdd.Description}");
Console.WriteLine("-----------------");
```

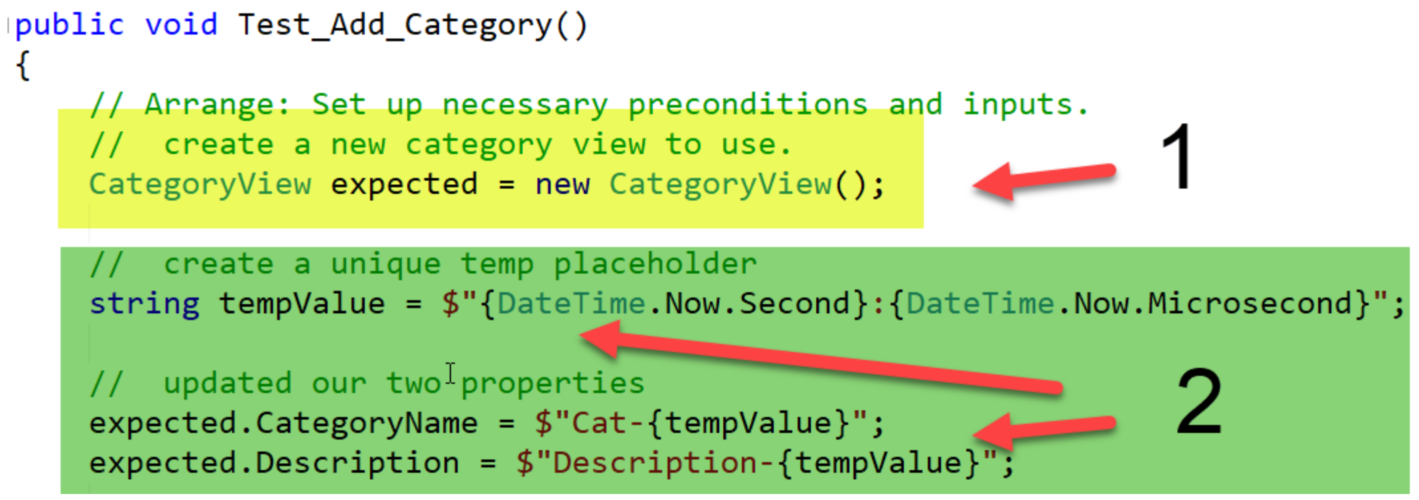


```
---------- Before Add ------------
CategoryID: 0
Category Name: Add - James
Description: Add - James was here!
-----------------
---------- After Add ------------
CategoryID: 17
Category Name: Add - James
Description: Add - James was here!
-----------------
---------- Before Edit ------------
CategoryID: 17
Category Name: Add - James
Description: Add - James was here!
-----------------
---------- After Edit ------------
CategoryID: 17
Category Name: Edit - James
Description: Edit - James was here!
-----------------
```

Unit Tests

Arrange

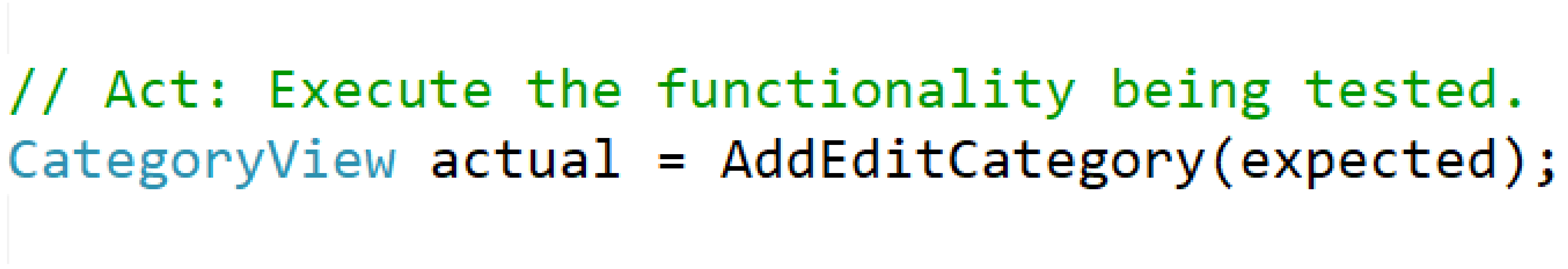
1. Create a new Category View
2. Create a unique temporary variable for testing and update any properties that we wish to test.



```
public void Test_Add_Category()
{
    // Arrange: Set up necessary preconditions and inputs.
    //  create a new category view to use.
    CategoryView expected = new CategoryView();

    //  create a unique temp placeholder
    string tempValue = $"{DateTime.Now.Second}:{DateTime.Now.Microsecond}";

    //  updated our two properties
    expected.CategoryName = $"Cat-{tempValue}";
    expected.Description = $"Description-{tempValue}";
```

Act

- Pass the expected "Category View" to the Add/Edit Category method and store the returning result.

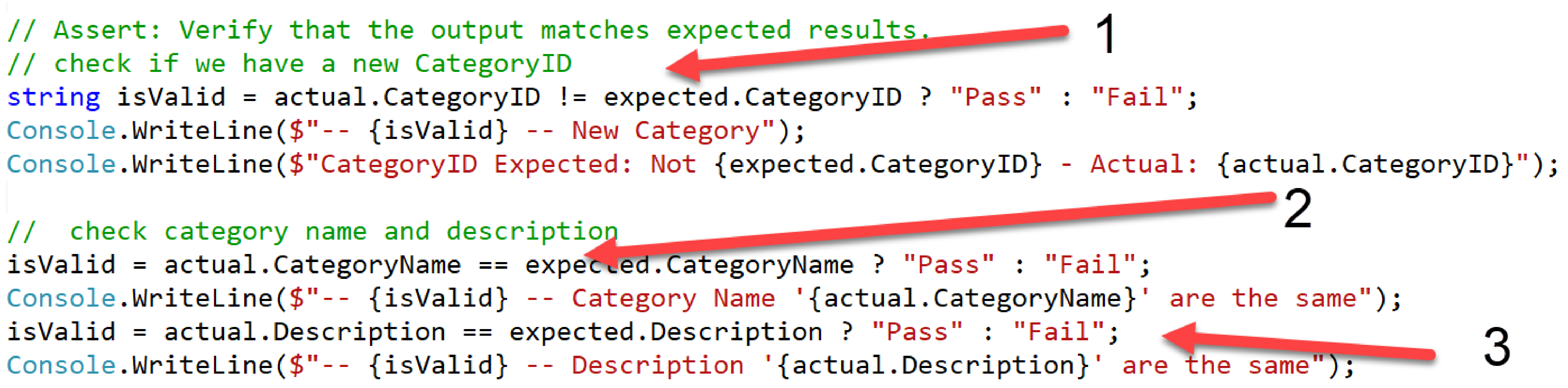


```
// Act: Execute the functionality being tested.
CategoryView actual = AddEditCategory(expected);
```

Assert

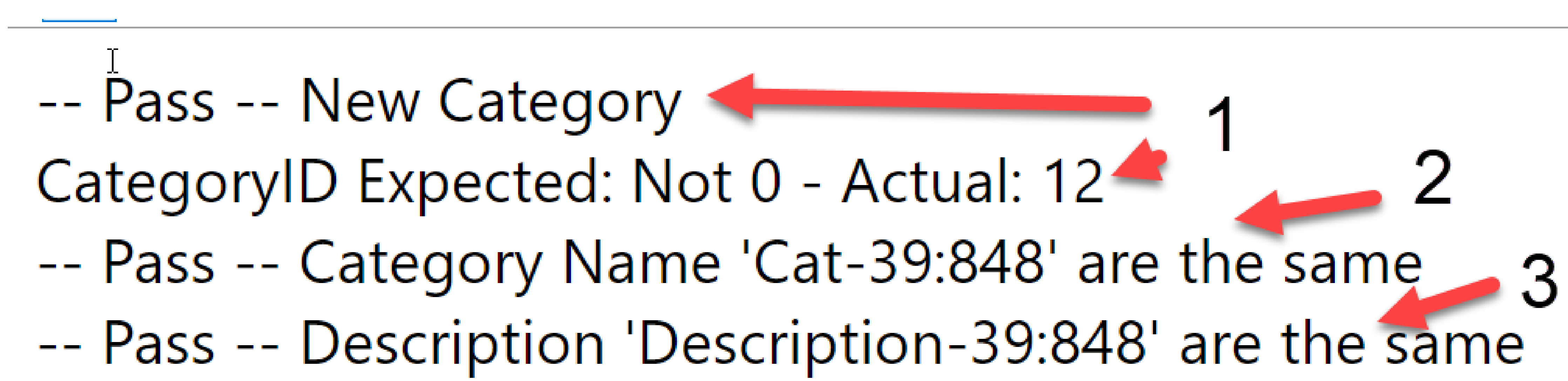
- Test three properties:

1. A new Category has been created (CategoryID greater than “0”).
   - NOTE: Our check is to ensure that the original CategoryID does not match the new CategoryID.
2. The new category name matches the original category name.
3. The new category description matches the original category description.



```
// Assert: Verify that the output matches expected results.
// check if we have a new CategoryID
string isValid = actual.CategoryID != expected.CategoryID ? "Pass" : "Fail";
Console.WriteLine($"-- {isValid} -- New Category");
Console.WriteLine($"CategoryID Expected: Not {expected.CategoryID} - Actual: {actual.CategoryID}");

//  check category name and description
isValid = actual.CategoryName == expected.CategoryName ? "Pass" : "Fail";
Console.WriteLine($"-- {isValid} -- Category Name '{actual.CategoryName}' are the same");
isValid = actual.Description == expected.Description ? "Pass" : "Fail";
Console.WriteLine($"-- {isValid} -- Description '{actual.Description}' are the same");
```

Test Results



```
-- Pass -- New Category
CategoryID Expected: Not 0 - Actual: 12
-- Pass -- Category Name 'Cat-39:848' are the same
-- Pass -- Description 'Description-39:848' are the same
```