IJARCCE

ISSN (Online) 2278-1021
ISSN (Print) 2319 5940

International Journal of Advanced Research in Computer and Communication Engineering
ISO 3297:2007 Certified
Vol. 5, Issue 10, October 2016

The Design and Testing of a Web Application by Using Modern Implementations Technologies

Al Rubaie Evan Madhi Hamzh

Department of Computer Science, University of Pitesti, Arges, Romania

Abstract: The paper involves creating a web architecture which will set the basis for the completion of an e-commerce site. This is part of the "design" stage, a stage which is present in almost all methodologies for the development of software systems. The technology proposed for the subsequent implementation of the site is ASP.NET MVC (model view controller). The application is designed for the submittal of offers for different products and services. Each product is described in detail: name or price, the category to which it belongs, its description and a representative image. The products are put in a basket from which they can be deleted or bought. The application allows its management through a manager's panel: adding new products, creating new categories of products, editing and deleting the already existing products. After an order has been placed, the manager is announced through an email containing the necessary data for filling the order.

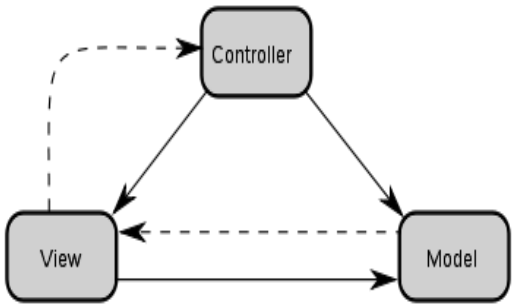
Keywords: web design, web application, web programming, and software methodologies.

1. INTRODUCTION

Generally, the purpose of many computers is to take over information from a certain location, to process it according to the user's preferences and lastly to display it to the user. After the user has modified the contents of the information and after some potential processing has been applied, the system renews the information in the place from where he took it over initially. The easiest method of performing an application that performs these operations is to put the operations together and treat them as a whole. This method is good because it's easy to implement. Nevertheless, afterwards problems occur when one of the components of the date flow is to be changed, for instance when the change of the interface is desired. Another problem involves the business logic that must be incorporated, a logic which is also subject to changes and goes beyond a simple interchange of information. Consequently, the need to modularize the application appears the need to delimit neatly the components in order for them to be able to be changed easily and after the modification the components must still be compatible with the other modules that make up the application.

A solution to this problem is the Model-View-Controller (MVC) architecture which separates data storage from data presentation and processing. So, we have three distinct classes: The model deals with the application behaviour and data; it responds to requests concerning the state of the system, requests to change the state and notifies the user when these changes have taken place so that he may react. The view transposes the model into a form allowing an easy interaction, typically a visual interface. There may be multiple views for a single model for different purposes. The controller receives input from the user and initiates an answer following the requests to model objects.

The controller is the one that controls the other two object classes, view and model, instructing them to perform operations based on the input received from the user.



The diagram of the MVC architecture presents the solid lines as direct associations and the dotted lines as indirect associations.

MVC was described for the first time in 1979 by TrygveReenskaug who worked at the time with Smalltalk within Xerox PARC. The original implementation is described in detail in the paper Applications Programming in Smalltalk-80: How to use Model-View-Controller.

An application oriented on the MVC principles may be a collection of triad model/view/controller, each one being in charge of a different element of the user interface.

MVC often occurs in web applications where the view is the HTML code generated by the application. The controller receives GET and POST variables as input and decides what to do with them and it sends them forward to the model. The model, which contains the business logic and the associated rules, may perform the necessary

IJARCCE

ISSN (Online) 2278-1021
ISSN (Print) 2319 5940

International Journal of Advanced Research in Computer and Communication Engineering
ISO 3297:2007 Certified
Vol. 5, Issue 10, October 2016

operations on the data in order to enable the application to generate the above-mentioned HTML code via template engines, XML pipelines, Ajax type requests etc.

The model is not necessarily only a database, as it's often both the database and the necessary business logic in order to manipulate the data in the application. Many applications use a persistent mechanism of data storage. MVC does not specify explicitly the level of access to the data precisely because it's obvious that this in encapsulated in the model. In some simple applications which have few logically imposed business rules, the model may be confined to the database and the functionalities provided by the database.

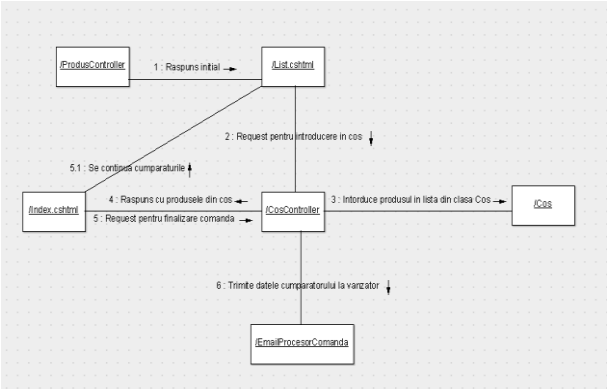
The view is also not confined to the display of the information, as it has an important role also in the interaction with the user. In the case of the above example of web applications, the interface generated via an html code is the one in charge of taking over the input and also the measures taken in order for it to be accurate.

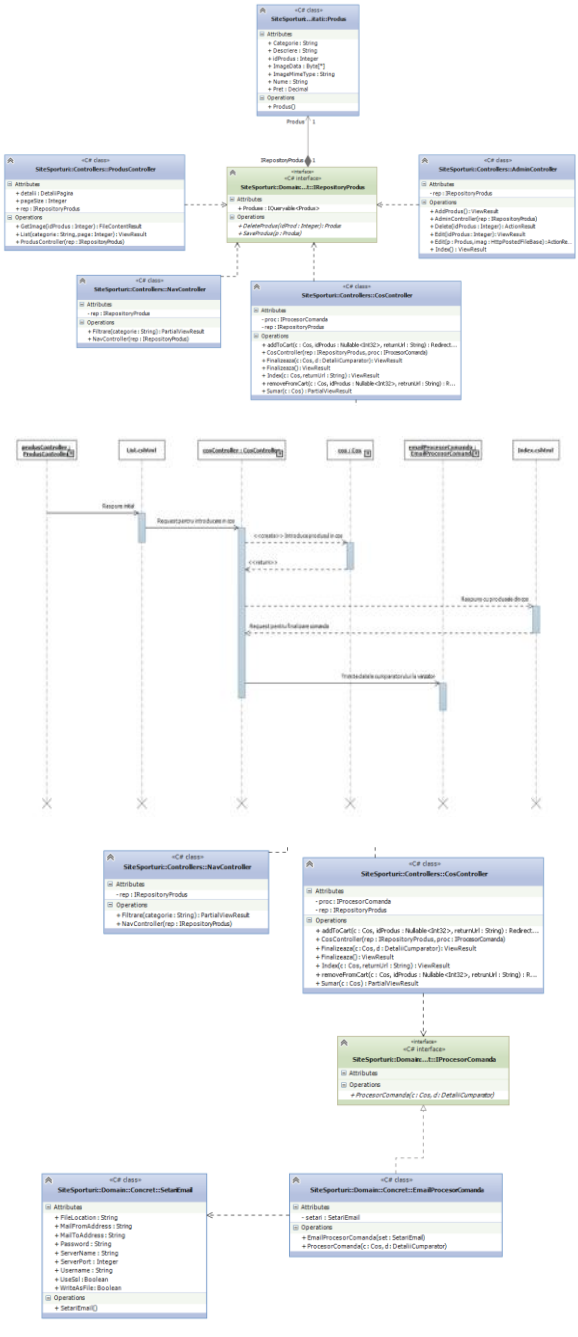
The controller is often mistaken for the application itself, whereas its role is to direct data between the other two classes of objects. Indeed, the model may perform many operations on the data, but these operations depend on data format at a given moment. The data which are displayed /collected from the user often differs significantly from the data which is stored in the database. These differences are due to the conversions the controller may apply to the data in order to facilitate information traffic between components. Each object class has certain definite expectations regarding data format, but these format transformations must be performed automatically in order to maintain a constant data flow, relieving the other classes of the concern for the conversions and assuring the application that every module gets what it expects, besides the basic function of controlling the request traffic between modules.

The operational scheme of an application which is modelled according to the MVC architecture is generally as follows:

1. The user interacts with the interface (example: he presses a key on the keyboard)
2. The controller receives the action of pressing the key and it converts it into an action the model can understand.
3. The controller notifies the model of the user' s action and there usually follows a change in the state of the model (for instance: the model refreshes the state of the address field)
4. A view interrogates the model in order to generate an adequate interface (for instance: the viewer displays the new address next to the old one, near a confirmation key).
5. The interface awaits additional action from the user and the cycle repeats itself.

2. THE PROJECTION OF THE APPLICATION



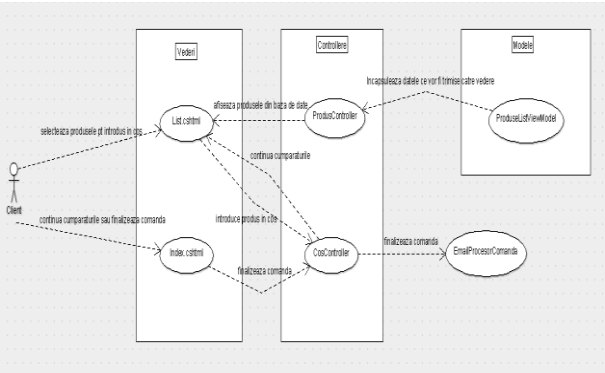


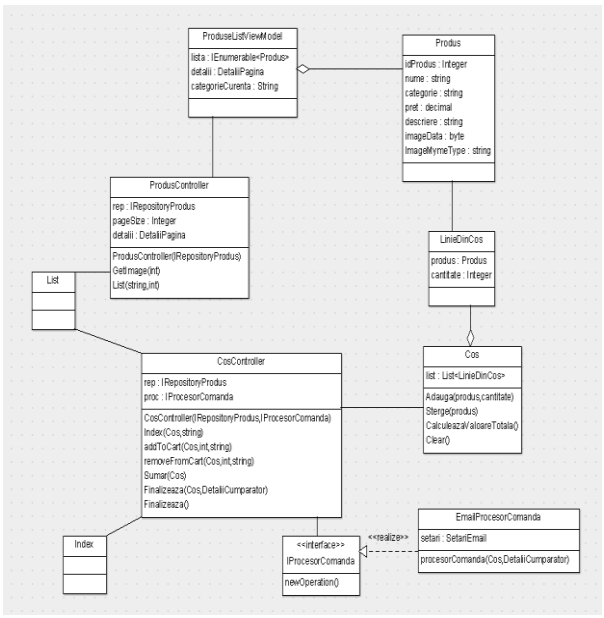
DOI 10.17148/IJARCCE.2016.51089

IJARCCE
ISSN (Online) 2278-1021
ISSN (Print) 2319 5940
International Journal of Advanced Research in Computer and Communication Engineering
ISO 3297:2007 Certified
Vol. 5, Issue 10, October 2016

3. THE IMPLEMENTATION OF THE APPLICATION AND THE DESCRIPTION OF THE FUNCTIONALITY OF THE CODE

The List method () of ProdusController is the first one to be used and it deals with listing the products which exist in the database. Actually, it creates an instance of ProduseListViewModel which it initializes with the data extracted from the database: images, name, product, price etc. This model is then sent to the List view (made up of an html code which is dynamically generated by the razor) which displays the data that is encapsulated in it. The List View contains an "Add to Cart" button which is a form that sends the ID of the desired product to the addToCart method of CartController.





The addToCart method in CartController involves the following actions: First, the database is interrogated in order to obtain an object which encapsulates the product data with the user-selected ID. Once this object has been obtained, a list shall be created and the object shall be added to the list, then the server shall return the Index view to the user. The ASP.NET technology allows this list to be deleted from the server RAM memory after a certain period of inactivity of the user thanks to whom the list was initialized. The Index view only displays the lists of the products in the cart (the prioritized list as a parameter from the addToCart method). Here, the user may manage the cart as each item of the list of the products which are in the cart is accompanied by a displayed button which accesses the removeFromCart method of CartController, which removes that product from the list. In this view, the user may access the "Continue Shopping" link, which shall send him directly to the previous List view that displays all the products. Also in the Index view, the user may finish the order by pushing the "finish" button which calls the Finish method in CartController. The methods of CartController only deal with the user's interaction with the products in the cart.

The Finish Method in CartController returns to the user a view that requests him to introduce his personal data in order to finish the order: email, surname, first name, address etc. Once the form has been filled in, it shall be sent to the server, namely to the second Finish method, the one with parameters. This method checks the cart, and if the cart is empty, it returns an error “We’re sorry, we cannot finish the order, as the cart is empty". If the cart is not empty, then the OrderProcessor method belonging to the EmailOrderProcessor class is used through the IOrderProcessor interface. This is possible due to the dependency injection of the container, which allows the methods belonging to the EmailOrderProcessor class to be accessed through the IOrderProcessor interface. This is one of the SOLID principles, which enable the programmer to create programs which are easy to maintain and extend in time. In our case, if the name of the EmailOrderProcessor is modified, we won't have to make changes in all the places where we initialized it because we always used its interface and the name of the implemented class shall only be modified in the dependency injection container.

The OrderProcessor method of the EmailOrderProcessor class uses an email model which it fills in with the buyer's data and the products ordered by him. This email is sent to the email of the owner of the online shop. Now the entire process is finished.

4. TESTING THE APPLICATION

The purpose of the unit testing it to have a suite of tests where the system behaves in a certain manner for each test. Each unit test is used in order to check a certain action regarding the system, but the side effect of each one is the supply of examples regarding the manner of interacting with different objects. These examples could specify the types of arguments, the types which expect to be returned or the scenarios when the exceptions are thrown away. All these together achieve an excellent form of system documentation. The unit tests are generally created with the support of a testing framework. There are several frameworks available for .NET but we are going to use the one which is integrated into visual studio. These

IJARCCE
ISSN (Online) 2278-1021
ISSN (Print) 2319 5940
International Journal of Advanced Research in Computer and Communication Engineering
ISO 3297:2007 Certified
Vol. 5, Issue 10, October 2016

frameworks provide support in order to know which methods are performed as tests and they provide additional functions in order to support the developer when he writes the tests.

Basically, there are a few key points when the unit tests are created. First, the test methods must belong to a public class, which has the attribute TestClass. Secondly, the test method per se must be public, with the test attribute TestMethod (different test frameworks have a different syntax in these cases).

After the tests have been created, there must be a way to perform them in order to find out if they're valid or not. Most frameworks come with different ways of performing unit tests, but in our case this step is achieved through the visual studio platform.

When creating unit tests, there is a series of best practices which should be followed. The most important practice is to treat the test code with the same consideration and importance as the application code. This means checking the source code, building the tests together with the application, and making sure that they are easy to read without having any duplicate code or an identical logic broken down into separate methods.

One of the reasons for which some tests are easy to perform is the fact that they do not depend any other class in order to work. Nevertheless, there are objects in the real projects which cannot work independently. In these situations, we must find a way to concentrate on the class or methods in which we are interested, in order to prevent the classes on which it depended from being tested implicitly. A useful approach is that of using false objects, which simulate the functionality of the real objects within the project we are working on, but in a very specific and controlled manner. The false objects allow us to restrict the tests in order to examine only the functionality in which we are interested. The paid versions for Visual Studio 2012 include support for creating false objects, but we'll use a library named Rhino Mocks, which is easy to use and may be used together with all Visual Studio editions.

Adding a false object to a unit test is telling the Rhino Mocks library the kind of object we want to work with, the configuration of its behaviour and then its application to the test code. We can see how we added and used a false object in our unit tests belonging to the ValidatorTests class, a class which we specified above.

An issue which is often overlooked is the fact that every test must be performed only in order to test one single thing. There are two reasons for this: first, if the test fails, the identification of the reason for its failure becomes a lot more difficult, because there may be several causes. Ideally, we want the reason for the failure of the test to be as clear and obvious as possible, because the troubleshooting of the unit test code in order to identify the reason for its failure is a big waste of time, energy and productivity. The second reason is maintainability: if we have large tests, then their maintenance cost will be all the higher as their code is more difficult to understand.

We can test the validity of the models returned by the action methods of the controller by creating an instance of the controller and using the appropriate method for a view request. Then, by using the Model property, we'll obtain the model returned by the view. Then we can compare this model to a reference model, which will enable us to find out if the controller method returns accurate data to the view.

```
[TestClass]
public class UnitTest1
{
    [TestMethod]
    public void Produse_returnate()
    {
        //arrange in care pregatesc tot ce ii este necesar controllerului
        Mock<IRepositoryProdus> mock = new Mock<IRepositoryProdus>();
        mock.Setup(m => m.Produse).Returns(new Produs[]{
            new Produs{Nume="tenis",Pret=10},
            new Produs{Nume="inot",Pret=15},
            new Produs{Nume="fotbal",Pret=20}
        }.AsQueryable());

        ProdusController pc = new ProdusController(mock.Object);
        pc.pageSize = 2;
        //act
        ProduseListViewModel lista = (ProduseListViewModel)pc.List(null, 1).Model;
        Produs[] vector = lista.lista.ToArray();

        //extragem datele primite in vedere (prima vedere adica pagina 1 caci paginei 2 ii ramane 1 produs de afisat) prin metoda Model

        //assert
        Assert.AreEqual(vector.Length, pc.pageSize);
        Assert.AreEqual(vector[0].Nume, "tenis");
        Assert.AreEqual(vector[1].Nume, "inot");
        //trebuie ca lista extrasa sa aiba lungimea 2 ca pc.pageSize
    }

    [TestMethod]
    public void Paginatie()
    {
        //arrange
        HtmlHelper helper = null;

        DetaliiPagina detalii = new DetaliiPagina
        {
            itemeTotale = 3,
            itemePePagina = 2,
            paginaCurenta = 1
        };

        Func<int, string> delegat = i => "Page" + i;
        //act
        MvcHtmlString mvc = helper.Paginatie(detalii, delegat);
        //assert
        Assert.AreEqual(mvc.ToString(), @"<a class=""selected"" href=""Page1"">1</a><a href=""Page2"">2</a>");
    }
```

```
[TestMethod]
public void poateTrimite_ProduseListViewModel()
{
    //arrange in care pregatesc tot ce ii trebuie controllerului
    Mock<IRepositoryProdus> mock = new Mock<IRepositoryProdus>();
    mock.Setup(m => m.Produse).Returns(new Produs[]{
        new Produs{Nume="tenis",Pret=10,Categorie="sport"},
        new Produs{Nume="inot",Pret=20,Categorie="lol"},
        new Produs{Nume="schi",Pret=30,Categorie="sport"},
        new Produs{Nume="baschet",Pret=40,Categorie="lol"},
    }.AsQueryable());

    ProdusController pc = new ProdusController(mock.Object);
    pc.pageSize = 2;
    string categ = "sport";

    //act
    ProduseListViewModel lista = (ProduseListViewModel)pc.List(categ, 1).Model;
    //assert
    Produs[] produse = lista.lista.ToArray();

    //Assert.AreEqual(produse[0].Nume, "tenis");
    //Assert.AreEqual(produse[1].Nume, "schi");
    //sau

    Assert.IsTrue(produse[0].Nume == "tenis" || produse[0].Categorie == "sport");
    Assert.IsTrue(produse[1].Nume == "schi" || produse[0].Categorie == "sport");

    Assert.AreEqual(lista.detalii.TotalPagini, new DetaliiPagina { itemeTotale = 2, itemePePagina = 2 }.TotalPagini);
}
```

```
[TestMethod]
public void reporteaza_categorie_selectata()
{
    Mock<IRepositoryProdus> mock = new Mock<IRepositoryProdus>();
    mock.Setup(m => m.Produse).Returns(new Produs[]{
        new Produs{Nume="tenis",Categorie="c1"},
        new Produs{Nume="inot",Categorie="c2"}
    }.AsQueryable());
    string categ = "c2";

    NavController nc = new NavController(mock.Object);

    string selectat = nc.Filtrare(categ).ViewBag.CategorieSelectata;

    Assert.AreEqual(selectat, categ);
}
```

IJARCCE

ISSN (Online) 2278-1021
ISSN (Print) 2319 5940

International Journal of Advanced Research in Computer and Communication Engineering
ISO 3297:2007 Certified
Vol. 5, Issue 10, October 2016

```
[TestMethod]
public void Merge_adaugarea_inCos()
{
    Produs p1 = new Produs { Nume = "nume1", Categorie = "c1", Pret = 10, idProdus = 1 };
    Produs p2 = new Produs { Nume = "nume2", Categorie = "c2", Pret = 15, idProdus = 2 };

    Cos c = new Cos();

    //act
    c.Adauga(p1, 3);
    c.Adauga(p2, 4);
    LinieDinCos[] lista = c.ReturnCos.ToArray();
    decimal sumaTotala = c.CalculeazaValoareTotala();

    //assert
    Assert.IsTrue(lista.Length == 2);
    Assert.AreEqual(p1, lista[0].Produs);
    Assert.AreEqual(p2, lista[1].Produs);

    Assert.AreEqual(sumaTotala, 90);

    //de obicei nu verific doua prop in aceeasi metoda test
}

[TestMethod]
public void NuPoateSaFinalizezeCosGol()
{
    Mock<IProcesorComanda> mock = new Mock<IProcesorComanda>();
    Cos c = new Cos();
    DetaliiCumparator d = new DetaliiCumparator();
    CosController controller = new CosController(null, mock.Object);

    ViewResult view = controller.Finalizeaza(c, d);

    //verifica daca comanda nu a fost trimisa la ProcesorComanda (nu trebuie sa fie apelat)
    mock.Verify(m => m.ProcesorComanda(It.IsAny<Cos>(), It.IsAny<DetaliiCumparator>()), Times.Never());
    //verifica daca metoda a trimis vederea default
    Assert.AreEqual("", view.ViewName);
    //verifica daca trimitem un model invalid la vedere
    Assert.AreEqual(view.ViewData.ModelState.IsValid, false);
}
```

UnitTest1

5. CONCLUSION

Apart from exposing the method for the design and implementation of a web application, the paper also dealt with the issue of unit testing, which involves having a suite of tests where the system behaves in a certain manner for each test. Each unit test is used in order to check a certain action regarding the system, but each one has the side effect of providing examples regarding the manner of interaction with different objects. These objects could specify the types of arguments, the types which expect to be returned or the scenarios when the exceptions are thrown away. These together achieve an excellent form of system documentation. The unit tests are generally created with the support of a test framework. There are different available frameworks for .NET, but we'll use the one which is integrated into visual studio. These frameworks enable us to know which methods are performed as tests and they provide additional functions in order to support the developer when he writes the tests.

REFERENCES

1. Patterns of Enterprise Application Architecture, Martin Fowler, David Rice, Addison Wesley 2002.
2. Adam Freeman and Steven Sanderson - “Pro ASP.NET MVC 4”, Fourth Edition.
3. Professional ASP.NET MVC4- Jon Galloway, Phil Haack, Brad Wilson, K. Scott Allen.
4. Doru Constantin, Emilia Clipici, "Backpropagation neural scheme for estimating the risk of bankruptcy of the Romanian insurance and reinsurance companies", Proceedings of the 15th International Conference on INFORMATICS in ECONOMY (IE 2016), Education, Research & Business Technologies, ISSN 2284-7472, ISSN-L 2247-1480, pg 415-421, Bucharest University of Economic Studies Press, June 02 – 05, 2016, Cluj-Napoca, Romania.
5. http://www.enode.com/x/markup/tutorial/mvc.html.
6. http://msdn.microsoft.com/en-us/library/ff649643.aspx.
7. Doru Constantin, "Principal Directions for Local Independent Components Analysis", ADVANCED TOPICS ON NEURAL NETWORKS, Proceedings of the 9th WSEAS International Conference on NEURAL NETWORKS (NN'08), Artificial Intelligence Series, WSEAS Press, ISBN: 978-960-6766-56-5, ISSN: 1790-5109, pg. 127-130, Sofia, Bulgaria, 2-4 Mai, 2008.
8. David Lane, Hugh E. Williams - Web Database Application with PHP and MySQL, 2nd Edition, O'Reilly, 2004.

BIOGRAPHY

Al Rubaie Evan Madhi Hamzh
Date of birth: 25 October 1976, Iraq, Babylon.
Qualification: Secondary School "Al Taleeaa", Iraq, Babylon University of Babylon, College of Science Doctor - University of Pitesti - Department of Computer Science. Hobby: Traveling, Reading.