e-ISSN : 0976-5166
p-ISSN : 2231-3850

Communication in MVC Teams: A Test-Driven Approach

Dhaval M. Anjaria
Institute of Business Studies and Research, Navi Mumbai

Abstract MVC architecture focuses on reducing the coupling between various parts of a project especially for teams who have remote workers. The paper proposes the use of Test Driven development to communicate the design and specifications among teams in a project. It presents a structure under which teams can work independently of each other through the use of writing unit tests as a way to communicate what each team needs provide to the other teams.

Keywords*:* ASP.NET MVC, Communication, Developer Coordination, Model View Controller, Test-driven Development, Unit Testing, Web Development.

1 Introduction to MVC

MVC defines an architectural pattern that divides the application into three main, separate components: the Model which represents the Data, the View which represents how the data will be presented to a user and the Controller which provides a binding between the two.

2 Test-Driven Development

Test-driven development involves writing tests first, having them fail, and then writing code that eventually passes those tests. Using this methodology, developers are forced to think about how the application will actually function before they write the code and thus the development of the application is driven by its design, which is documented in the tests.

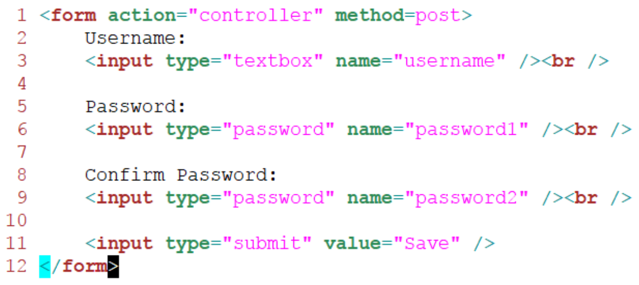
3 Tests as Communication

Communication in software development is vital and performed through a variety of tools such as Email, audio / video conferencing, messaging services, etc.[1] These tools while invaluable still leave room for uncertainty and coordination problems.[2], [3]

The paper proposes the idea of using unit tests as a means of communication between developers. It proposes that a team lay out exactly the result and behavior needed to be delivered by another team in the form of unit tests.

For example, consider a simple registration form in a web application. This registration form would require at least three textboxes: the username, the password and one to confirm the password.

Each of these is represented by a textbox in an HTML form with a specific name using the following code:



```
<form action="controller" method=post>
    Username:
    <input type="textbox" name="username" /><br />

    Password:
    <input type="password" name="password1" /><br />

    Confirm Password:
    <input type="password" name="password2" /><br />

    <input type="submit" value="Save" />
</form>
```

Fig. 1 An HTML Input Form

Assuming there is a dedicated frontend team responsible for writing this code, their primary concern would be to make sure the form looks nice, is properly formatted, etc. They would not be concerned with what happens at the backend.

When the user enters data and clicks “submit”, an HTTP POST request containing something similar to the following key-value pair is created and sent to the server:

```
{
        username: "user@mail.com",
        password1: "secret",
        password2: "secret"
}
```

Fig. 2: The HTTP POST data Generated

The backend team could then retrieve these values on the server-side in various ways depending on the language and the framework used. For example, in ASP.NET MVC, with C#, this could be implemented as:

```
var username = Request["username"];
var password = Request["password1"];
var password2 = Request["password2"];
```

Fig. 3: Retrieving values from HTTP POST

The names of the keys in the Request object have to match those used in the HTML form inputs. For the "Confirm Password" input, both "password2" and "confirmPassword" may be reasonable names. These names have to be the same in the object on the server-side (the Request object) and in the inputs on the client-side (the HTML form). So the frontend team and the backend team need to agree upon precisely what the client-side will send to the server-side.

This paper proposes that this be communicated by writing a unit test.

The tests would be the common ground, an agreement that all developers would adhere to. This may also allow them to work more independently since their requirements would be precisely defined through the unit tests.

The paper proposes the use of tests as a precise and verifiable means of communicating what needs to be done for a component. It proposes a form of Test-Driven Development, where design and specifications are communicated through unit tests that are written by one set of developers to communicate to any another set of developers about what needs to be done.

The tests would not replace any documentation but rather serve to augment it.

4 Implementation

These tests would serve as a "Single Source of Truth" for developers on all teams so that each team would know what code needs to be written.

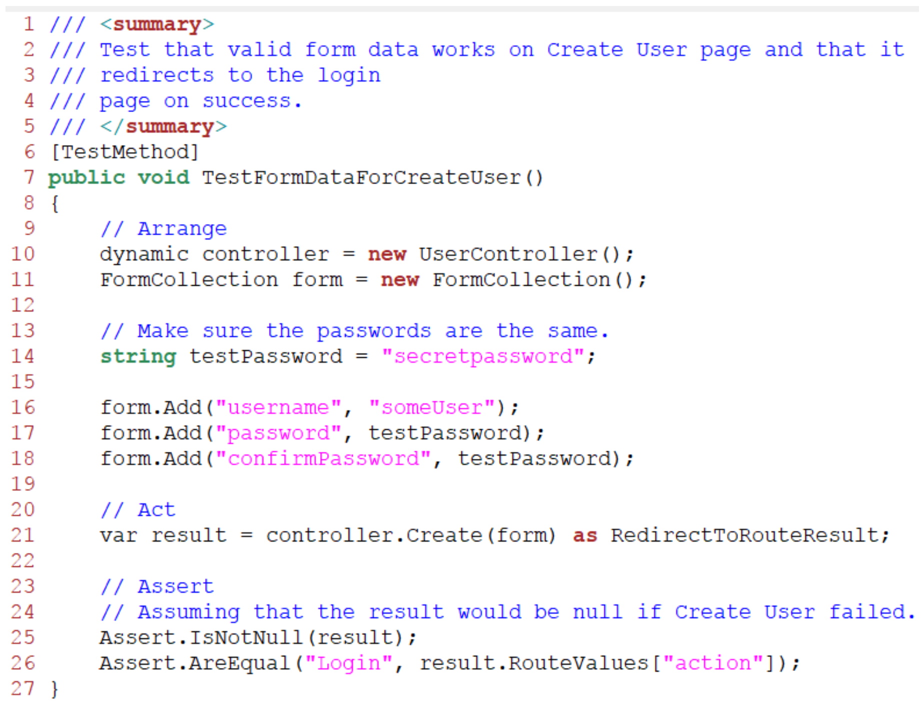
The fundamental question the paper addresses is, "How will the front-end team communicate to the back-end team what data they need for a page?" A sample implementation of this approach in a project built with the MVC pattern could be like this:

1. Dependencies between teams are identified from the design documents for a project.
2. Each team identifies what data and functionality they require from another team.
3. Team A writes tests for Component 1, which they depend on.
4. Team A verifies the tests, runs them. At this point, the tests fail.
5. Team B, responsible for implementing Component 1, reads the tests and implements Component 1 accordingly.
6. Team B also may add tests as they see fit. Eventually, all the tests pass.
7. If there are problems discovered in the tests, they are discussed between both teams and updated accordingly.
8. Once all tests are updated and passed, the code from both teams is integrated.

5 An Illustration

Consider the "Create User" form again, this time in ASP.NET MVC. Once again, we assume that a dedicated frontend team is the one building the client-side components and a backend team is working at the server-side.

Let's say that the frontend team decides on the name "confirmPassword" for their textbox. Along with the relevant frontend code, the team also writes a C# Unit Test similar to what is given below:



```
/// <summary>
/// Test that valid form data works on Create User page and that it
/// redirects to the login
/// page on success.
/// </summary>
[TestMethod]
public void TestFormDataForCreateUser()
{
    // Arrange
    dynamic controller = new UserController();
    FormCollection form = new FormCollection();

    // Make sure the passwords are the same.
    string testPassword = "secretpassword";

    form.Add("username", "someUser");
    form.Add("password", testPassword);
    form.Add("confirmPassword", testPassword);

    // Act
    var result = controller.Create(form) as RedirectToRouteResult;

    // Assert
    // Assuming that the result would be null if Create User failed.
    Assert.IsNotNull(result);
    Assert.AreEqual("Login", result.RouteValues["action"]);
}
```

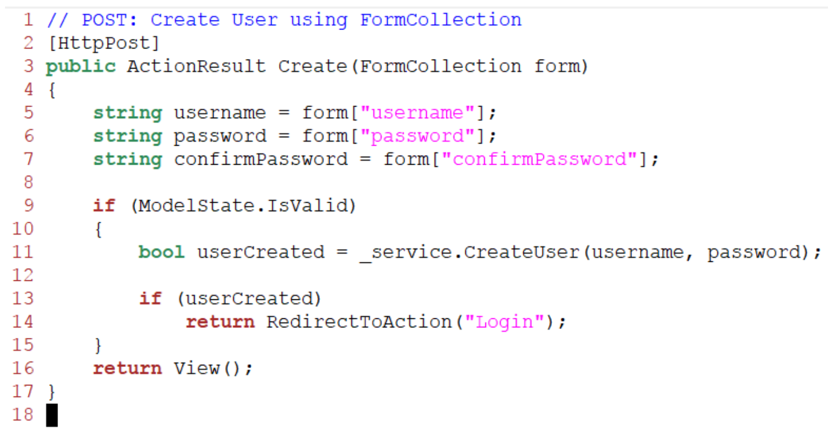
Fig. 4: Test Example 1

What is required, as outlined in the comment above the test, is that when valid form data is submitted, the controller redirects to the login page when user creation is successful.

The code in the "Arrange" section, does the necessary setup for this test to work. The code the "Act" section, simulates a user entering data into the Create User page and clicking submit.

The final section, the "Assert" section, checks the result of the operation and determines if the test passed or failed. It is recommended that when trying to understand these tests, developers read this section first before the other parts of the test code.

Sample implementation code that would satisfy this test would look like this:

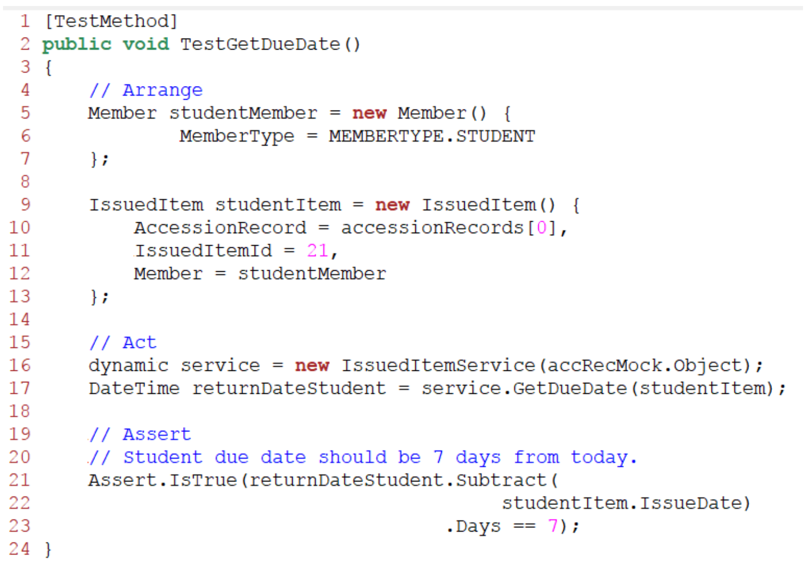


```
// POST: Create User using FormCollection
[HttpPost]
public ActionResult Create(FormCollection form)
{
    string username = form["username"];
    string password = form["password"];
    string confirmPassword = form["confirmPassword"];

    if (ModelState.IsValid)
    {
        bool userCreated = _service.CreateUser(username, password);

        if (userCreated)
            return RedirectToAction("Login");
    }
    return View();
}
```

Fig. 5: Implementation Example 1

In the code above, form data is validated and a Service object is used to create a user. If user creation is successful, the controller redirects to the Login page where the newly created user can start using using the system.

Let us take another example, this time for a Library Management System, assuming that our requirements are that Student members can borrow books for 7 days. This could be written down as a test as follows:



```
[TestMethod]
public void TestGetDueDate()
{
    // Arrange
    Member studentMember = new Member() {
            MemberType = MEMBERTYPE.STUDENT
    };

    IssuedItem studentItem = new IssuedItem() {
        AccessionRecord = accessionRecords[0],
        IssuedItemId = 21,
        Member = studentMember
    };

    // Act
    dynamic service = new IssuedItemService(accRecMock.Object);
    DateTime returnDateStudent = service.GetDueDate(studentItem);

    // Assert
    // Student due date should be 7 days from today.
    Assert.IsTrue(returnDateStudent.Subtract(
                                    studentItem.IssueDate)
                               .Days == 7);
}
```

Fig. 6: Test Example 2.

In the above test, we assume that a Service (IssuedItemService) object exists that performs database operations and other functions, such as getting the due date.

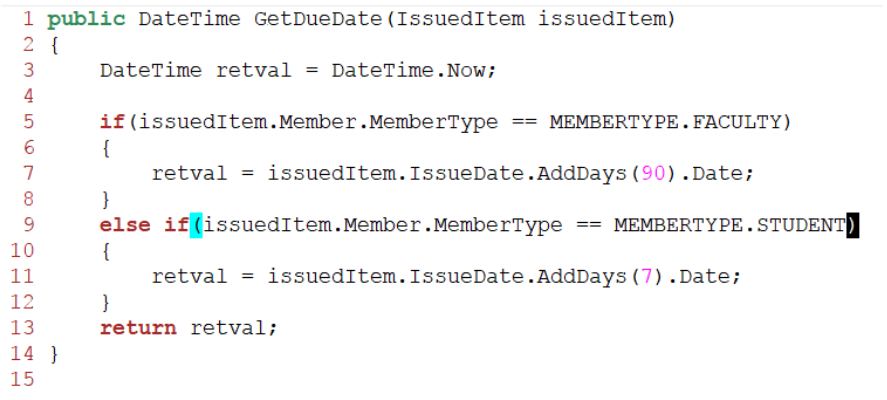
The Arrange section creates a Student user. It then creates IssuedItem records which means the student has borrowed a book from the library.

Once that is done, the Act section simulates getting the due date from the database. Finally, the assert section simply checks that the proper due date is returned.

This test may be complicated to get right. This is why the tests need to be properly commented. It is recommended that developers on both teams communicate if the test is unclear or faulty.

One of the advantages of this particular test is that it ensures that the developers implementing the code use Service objects in their implementation. There are several ways in ASP.NET MVC to get data from the database. This ensures that developers use the a specific method to do so.

Implementation code that might satisfy this test is given below:



```
public DateTime GetDueDate(IssuedItem issuedItem)
{
    DateTime retval = DateTime.Now;

    if(issuedItem.Member.MemberType == MEMBERTYPE.FACULTY)
    {
        retval = issuedItem.IssueDate.AddDays(90).Date;
    }
    else if(issuedItem.Member.MemberType == MEMBERTYPE.STUDENT)
    {
        retval = issuedItem.IssueDate.AddDays(7).Date;
    }
    return retval;
}

```

Fig. 7: Implementation Example 2.

This implementation is simple. However the test ensures that it is implemented and that this important function has no defects.

6 Main Advantages of the approach

6.1 *Changes made to the requirements will be reflected in the tests*

As the requirements of a project change, so will the tests. Hence the tests would serve as living documentation. Whatever changes need to occur, would be written in the form of tests.

For example, if the due date for Faculty members in the examples above (Fig. 6, 7) changes from 90 days to 60 days, that change should be reflected as a change in the tests which will fail if it is not implemented.

6.2 *The quality of the product may improve*

Research at IBM[4] and Microsoft[5] suggests that writing tests results in a decrease in defects and an increase in perceived quality of the product at the cost of a little productivity. This approach, being a form of test-driven development, would result in more tests being written overall.

6.3 *Reduced ambiguity in even the smallest requirements*

Tests are at the end of the day, code. Therefore they offer a certain amount of precision and granularity. For example, if a function requires the parameters to be a Linked List, the tests will require a Linked List. A dynamic array will not satisfy the test. Neither will any other similar object.

This introduces an understanding between all developers about what dependencies they satisfy and what dependencies they need satisfied. For example if the security team decides that certain fields in a form must not be included in an HTTP POST form, they can cement that requirement as a test.

6.4 *Tests fail if code is not written*

If functionality is missing or is neglected for whatever reason, the tests will fail until it is implemented.

7 Existing Literature and Approaches

The problem of communication in software development has been approached before and this section includes a review of existing solutions and how the approach described here differs from them.

The first approach is standard TDD, where a developer writes their own tests first, writes code to pass the tests and then refactors the code to make it better. This approach was outlined in Kent Beck's book, "Test-Driven Development by example". This kind of approach is sometimes used in Extreme Programming.[10]

This paper takes a somewhat different approach to TDD. For this approach, any functionality that involves more than one team, tests will be written as the single source of truth for what each team expects from one another. These tests may or may not be separate from tests developers write for themselves.

A solution such as StarUML[6] helps accomplish the goal of updated documentation. Unit tests can contain behaviors such as the redirection function explained in Fig. 7 which may be difficult to express as an automatically generated UML diagram.

The tests that have been written can also serve as a metric for how much work is done and how much remains. IDEs such as Visual Studio and Eclipse provide support for unit testing provide red-green colour-coded visual feedback of exactly how many tests have passed and how many have not. Hence, the approach described can be a tracker / monitor of how much work is planned, how much work is complete and how much needs to be done.

"Specification By Example", popularized by Martin Fowler describes a similar approach[7]. The approach outlined here can be thought of as a specific implementation of that idea. However, this approach is limited to developer-to-developer communication. Those unfamiliar with the technology used are not expected to understand the tests. To achieve that, using more abstract specifications is recommended.

8 Limitations of the approach

An informal trial project was conducted at the Institute of Business Studies and Research, Navi Mumbai using this approach. The limitations and advantages of approach that were discovered were similar to the ones described for TDD in general, in the IBM and Microsoft research.

The project included teams of students that were divided into frontend and backend teams. The frontend team required a fixed structure of data to build their Views against and the backend team needed to conform to those structures. Since the teams could not always be present in the same room, a formal means of communication was required to make sure the teams had a common record of what is required from each other when using ICTs. This approach was then applied.

Some limitations of this approach that were observed are given below:

8.1 *The tests are not perfect*

Tests could sometimes be inaccurate and therefore the developer in charge of writing the implementation code was given the liberty to change the tests if they felt they needed to. Most of the time, the developer did so after discussion with the rest of the team. Version control software was used to track exactly what changes they made.

8.2 *The tests require additional documentation*

A descriptive comment about the test is recommended to accompany it.

8.3 *The tests become very granular*

If the tests are used as the exclusive mode of communicating requirements, they may need to be extremely granular covering all possibilities of outputs that a function may or may not produce. This may sometimes need more time than is required to get the job done.

Secondly, the tests by themselves did not give developers a clear idea of the bigger picture of the project. As such, the tests serve to augment existing documentation rather than replace it.

8.4 *Developers need to be familiar with how tests are written*

Developers who were not familiar with unit testing was done in C# had to be trained to understand the tests. Only then could they use them properly, otherwise the tests served no purpose.

9 Conclusion

The basic idea of this paper is that developers on two teams that are working on features dependent on one another, for example the frontend and the backend teams, use tests as a common document to communicate exactly what output is needed and what input will be given by each team. This way, tests are used as commuication between developers.

The idea of using tests as a mode of communication between developer teams has advantages and disadvantages, similar to the ones found with the TDD approach. There are existing solutions to developer communication, however the main advantage with this approach is that code would have more tests.

This paper provides a hypothetical development model on how this may be implemented in a project along with an informal case study of a project where it was applied. More research needs to be conducted on the viability of this approach in various scenarios. It is hoped that this approach will lead to easier distributed development.

10 Acknowledgments

The author wishes to thank Prof. Murali Krishnamurthy, Prof. Deepa Manoj, Dr. Sumita Sarkar and Dr. M. L. Monga.

The author also thanks the contribution of Akshat Sharma, Siddhant Kumbhar, Soumyajit Maiti and Farzaad Gorporia for their work in the trial project [17].

11 References

[1] Thissen, M. Rita, Jean M. Page, Madhavi C. Bharathi, and Toyia L. Austin. "Communication tools for distributed software development teams." In Proceedings of the 2007 ACM SIGMIS CPR conference on Computer personnel research: The global information technology workforce, pp. 28-35. ACM, 2007.

[2] Lai, Su-Ying, Richard Heeks, and Brian Nicholson. "Uncertainty and Coordination in Global Software Projects: An UK/India-centred Case Study", University of Manchester. Institute for development policy and management (IDPM), 2003..

[3] Poile, Christopher, Andrew Begel, Nachiappan Nagappan, and Lucas Layman. "Coordination in large-scale software development: Helpful and unhelpful behaviors." Dept. of Management Sciences University of Waterloo Waterloo, ON, Canada cpoile@ uwaterloo. ca and Microsoft Research Redmond, WA, fandrew. begel, naching@ microsoft. com. and Dept. of Computer Science, North Carolina State University, Raleigh, NC, lucas. layman@ ncsu. edu (2009).

[4] Sanchez, Julio Cesar, Laurie Williams, and E. Michael Maximilien. "On the sustained use of a test-driven development practice at ibm." In Agile Conference (AGILE), 2007, pp. 5-14. IEEE, 2007.

[5] Williams, Laurie, Gunnar Kudrjavets, and Nachiappan Nagappan. "On the effectiveness of unit test automation at microsoft." In Software Reliability Engineering, 2009. ISSRE'09. 20th International Symposium on, pp. 81-89. IEEE, 2009.

[6] StarUML (https://en.wikipedia.org/wiki/StarUML)

[7] Alister Scott, "Specification By Example", (https://www.thoughtworks.com/insights/blog/specification-example), 2011

[8] Heidke, Nick, Joline Morrison, and Mike Morrison. "Assessing the effectiveness of the model view controller architecture for creating web applications." In Midwest instruction and computing symposium, Rapid City, SD. 2008.

[9] Hameed, Madiha, Muhammad Abrar, Ahmer Siddiq, and Tahir Javeed. "MVC Software Design Pattern in Web Application Development."

[10] Code the Unit Tests First (http://www.extremeprogramming.org/rules/testfirst.html)

[11] Wikipedia.Communication in Distributed Software Development (https://en.wikipedia.org/wiki/Communication_in_Distributed_Software_Development)

[12] Burbeck, Steve. "Applications programming in smalltalk-80 (tm): How to use model-view-controller (mvc)." Smalltalk-80 v25 (1992): 1-11.

[13] Allister Scott. Specification By Example. https://www.thoughtworks.com/insights/blog/specification-example

[14] Avi Silberschatz, Peter Baer Galvin, Greg Gagne. "Remote Procedure Calls", pp. 136-140. Operating System Concepts, 9th Edition, Wiley, 2016.

[15] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. "Design Patterns in Smalltaclk MVC" Design Patterns: Elements of Reusable Object-Oriented Software, pp. 4, Pearson, 1994.

[16] Kent Beck, Test-Driven Design by Example, Wiley, 2002.

[17] MVC Library management System (https://github.com/ibsarbca/MVCLibraryManagementSystem)