Test Driven Development Guidelines

Objectives

Every software development process aims to provide some or all of the following benefits. They are listed below and they will be revisited later to see if they can be achieved by using TDD.

- Simplicity
- Clarity
- Elegance
- Modularity
- Extensibility
- Testability
- Flexibility
- Maintainability

TDD Prerequisites

TDD Prerequisites - Programming Practices

To make TDD adoption easier, we could perhaps re-examine some of the things we do currently and see if we can do them in a simpler or more elegant way. The points below are fairly basic and while we do some of them we need to ensure that we are being more consistent in adopting a programming style that we all agree on.

1. SOLID principles
2. Model binding
3. Error handling
4. Thin Controllers
5. Program against interfaces/Dependency Injection
6. Html Helpers
7. HttpContext Decoupling

SOLID principles

There are plenty of resources out there about SOLID principles so I am not going to cover all of them here.

Model Binding

```
//Form has 2 fields for id and name.
//Without binding
[HttpPost]
public ActionResult Create(int id, string name)
{
        personDao.Save(id, name);
}

//A better way
[HttpPost]
public ActionResult Create(Person p)
{
        personDao.Save(p);
}
```

It is always better to have classes interact with each other by passing valid objects where posssible instead of long, error-prone lists of arguments.

```
//if you have an accessor like this
int InsertPerson(int id, string name)
{ …. }

int UpdatePerson(int id, string name)
{ ... }

//think about providing a simpler interface to callers by creating a new method,
//overloading the exisiting methods or use a wrapper class.
int Save(Person p)
{
```

```
        if (this.GetPerson(p.id) != null)
                return this.UpdatePerson(p.Id, p.Name);
        else
                return this.InsertPerson(p.Id, p.Name);
}
```

Error Handling

```
//id and name are passed through a form to this controller method
public ActionResult Create(int id, string name)
{
        if (String.IsNullOrEmpty(name))
                ModelState.AddError("some error");
}

//A better way
public class Person
{
        [Required]
        public string Name {get; set;}
        …
}

//or by using the IValidatableObject interface that allows complex validation logic to be
//removed from the controller

public IEnumerable<ValidationResult> Validate(ValidationContext validationContext)
{
        List<ValidationResult> errors = new List<ValidationResult>();

        if (String.IsNullOrEmpty(Name))
           errors.Add(new ValidationResult("*", new string[] { " Name " }));

        return errors;
 }
```

Thin Controllers

```
//A bloated controller
//processPersonDetails amd FormatPersonDetails should not be here
public ActionResult Display(int id)
{
          var p = personDao.GetPerson(id);
          var pd = this.ProcessPersonDetails(p);
          var pf = this.FormatPersonDetails(pd);
          return View("Display", pf);
}

//A better way
public ActionResult Display(int id)
{
          var p = personService.GetFormattedPerson(id);
          Return View("Display", p);
}
```

Program against interfaces

```
//Hard coded dependency
public class PersonService
{
	private PersonDao _personDao = new PersonDao();
	…
}

//A better way
public class PersonService
{
	private readonly IPersonDao _personDao; Public PersonService(IPersonDao personDao)
	{
		_personDao = personDao;
	}
	…
}
```

Html Helpers

```
<%-- too much of that makes views hard to read and update. What happens if the logic needs to take more cases into account? --%>
<%=profile.BrandName.ToLower() == "cosmos" ? "style=\"display:none\"" : "" %>

//A better way
public static MvcHtmlString GetHeaderStyle(this HtmlHelper helper, SiteProfile profile)
{
	switch (profile.BrandName)
	{
		case “cosmos”:
		…
	}
}
```

Http Context Decoupling

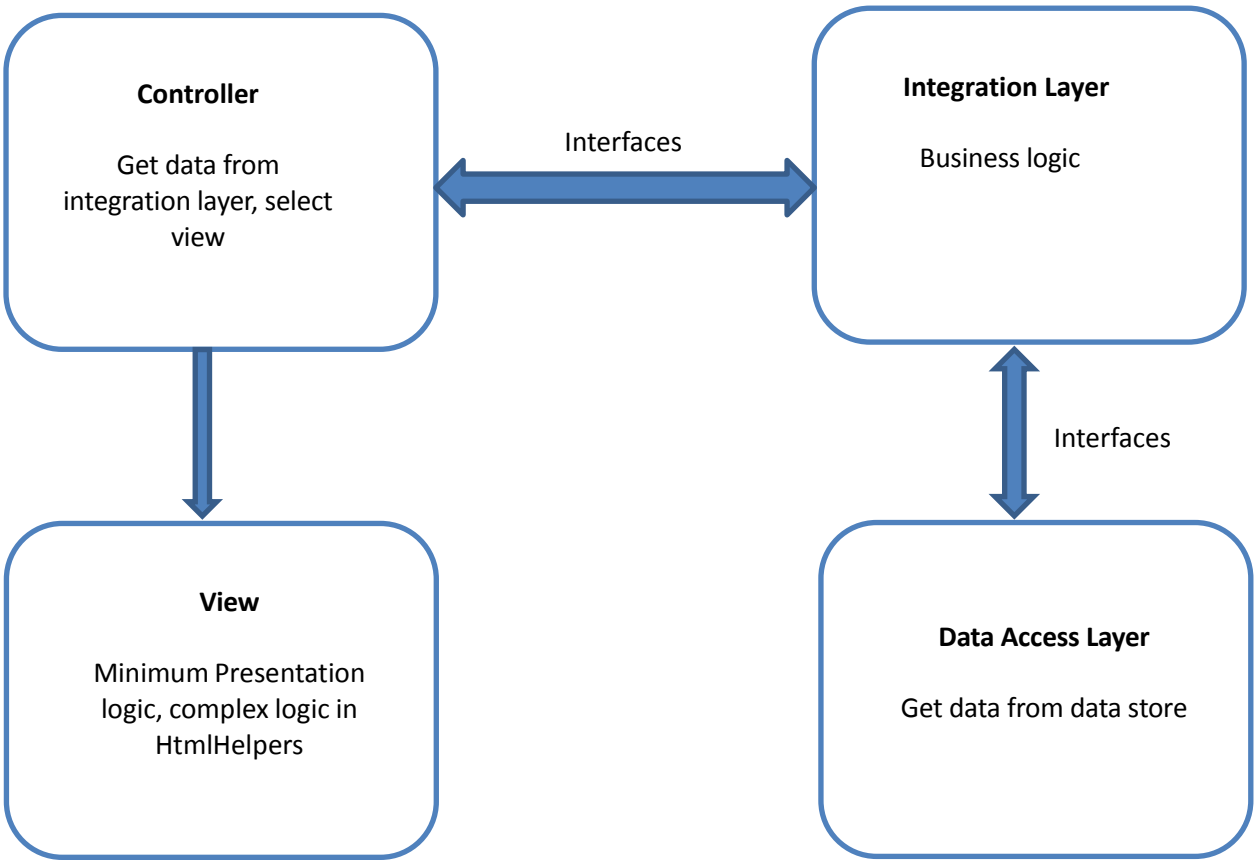
```
//This may be hard to unit test
public ActionResult Display()
{
	if (System.Web.HttpContext.Current.Session[“user”] = “john”)
		return View(“john”);
	else
		return View(“paul”);
}

//using Controller's http context property (returns HttpContextBase which can be
//mocked) or using a custom wrapper for HttpContext
public ActionResult Display()
{
	if (HttpContext.Current.Session[“user”] = “john”)
	//OR if (MyContext.GetUser() = “john”)
		return View(“john”);
```

```
        else
                return View("paul");

}
```

TDD Prerequisites – Architecture



Different layers of the application should expose functionality through interfaces. Why is this important?

```
public class OrderProcessor
{
        private order_db = new order_accessor();

        public ProcessOrder(order o)
        {
                ….. order_db.SaveOrder(o);
        }
}
```

OrderProcessor is now strongly tied to the order_accessor that is in our data access layer. If someone wants to use this class to save orders to a text file or submit it as xml to a webservice he wont be able to do it even if the operations he wants to perform are the same and defined below

```
interface IOrderAccessor
{
	void SaveOrder(Order o);
	Order GerOrder(int orderId);
}

public class OrderProcessor
{
	private readonly IOrderAccessor _orderAccessor;

	public OrderProcessor() : this(new order_accessor())
	{}

	public OrderProcessor(IOrderAccessor orderAccessor)
	{
		_orderAccessor = orderAccessor;
	}
}
```

This class is now much more flexible as it allows the OrderProcessor to be instantiated with a different implementation of the data accessor which could be a real implementation or a mock object used for our unit testing.

In general, Controllers should know as less as possible about the Model and the Model should know as less as possible about the Data Layer.

- Define the contract (Interface) of the Controller to Model Interaction
- Define the contract (Interface) of the Model to Data Layer Interaction

The Repository Pattern

This is something we are already using in the form of our accessors that encapsulate all data access logic.

```
//Nasty code without a repository
public class HomeController : Controller
{
	public ActionResult Index()
	{
		var dataContext = new MovieDataContext(); var movies = from m in dataContext.Movies
		select m;
		return View(movies);
	}
}

//A better way
public class MovieRepository : IMovieRepository
{
	private MovieDataContext _dataContext;

	public MovieRepository()
	{
		_dataContext = new MovieDataContext();
```

```
        }

        #region IMovieRepository Members

        public IList<Movie> ListAll()
        {
                var movies = from m in _dataContext.Movies select m;
                return movies.ToList();
        }

        #endregion
}

public class MoviesController : Controller
{
        private IMovieRepository _repository;

        public MoviesController() : this(new MovieRepository())
        { }

        public MoviesController(IMovieRepository repository)
        {
                _repository = repository;
        }

        public ActionResult Index()
        {
                return View(_repository.ListAll());
        }
}
```

Dependency Injection

The examples above already use Dependency Injection. Below is an example without DI. This is as inflexible as the OrderProcessor example before. The controller has a hard coded dependency on the MovieRepository class. DI aims to remove these hard coded dependencies by allowing dependencies to be “injected” at runtime, providing more flexibility and testability.

```
// Without DI
public class MoviesController : Controller
{
        private MovieRepository _movieRepository = new MovieRepository()
}

// With DI
public class MoviesController : Controller
{
        private IMovieRepository _repository;

        public MoviesController() : this(new MovieRepository())
        { }

        public MoviesController(IMovieRepository repository)
```

```
        {
                _repository = repository;
        }
        …
}
```

Notice the 2 overloaded constructors. The parameterless constructor will be used by .NET when creating your controller. The constructor with the interface parameter can be used by you when you write unit tests.

There are various ways and tools for achieving DI. The examples above use the Constructors. A setter property/method could be used as well and in more advanced scenarios the dependency can be injected by another factory class or tool.

See below for a more formal definition.

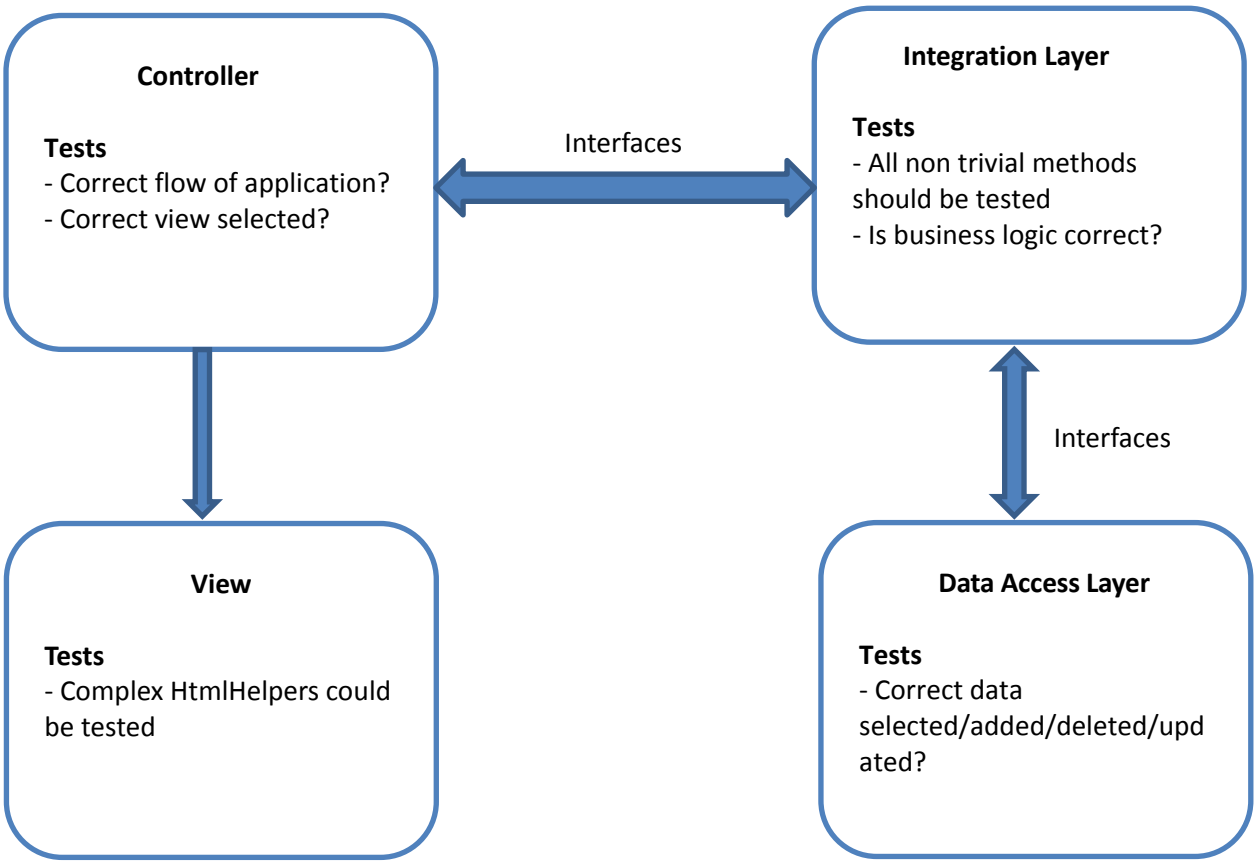
Dependency Injection involves at least three elements:

- A dependent consumer,
- A declaration of a component's dependencies, defined as interface contracts,
- An injector (sometimes referred to as a provider or container) that creates instances of classes that implement a given dependency interface on request.

The dependent object describes what software component it depends on to do its work. The injector decides what concrete classes satisfy the requirements of the dependent object, and provides them to the dependent.

So for the example above the dependent consumer is our MoviesController. Its dependency is something that implements IMovieRepository as it needs it to do its job. What we can now do if we use a Dependency Injection Tool or Container is tell the tool what needs to be injected into the MovieController when an instance of it is created. If we create a mapping between IMovieRepository and MovieRepository then the tool will inject an implementation of MovieRepository at run time.If instead we choose to map IMovieRepository to OtherMovieRepository then an instance of OtherMovieRepository will be injected.

TDD in the different layers of the application



It is questionable whether using TDD with the data access layer/repositories, controller code and view code can yield any real benefits. The focus should really be on the Integration Leayer that provides the majority of the business logic. And even there, if we want to be pragmatic about it, only the functionality that is complex enough to justify the investment in time for adopting this methodology, may be the one we need to focus on.

TDD Definition

Test-driven development (TDD) is a software development process that relies on the repetition of a very short development cycle: first the developer writes a failing automated test case that defines a desired improvement or new function, then produces code to pass that test and finally refactors the new code to acceptable standards.

Test-driven development requires developers to create automated unit tests that define code requirements (immediately) before writing the code itself. The tests contain assertions that are either true or false. Passing the tests confirms correct behavior as developers evolve and refactor the code. Developers often use testing frameworks, such as xUnit, to create and automatically run sets of test cases.

What TDD really is

What the name Test Driven Development has going against it is that it doesn't properly express the purpose of TDD; namely, that it is a process designed to help you drive and iterate the design of your implementation at the unit level. ***The result of the design process is unit tests, but their primary purpose is not one of quality assurance; rather, it is an expression of the intended usage of the component under design.*** In this way, the "tests" that you are writing become the first client of your component, and come into being just before the component's code is written. The rhythm in TDD is "write a test, watch it fail; write the production code, watch the test now pass; when prudent, refactor the code to increase clarity and remove duplication".

The unit tests written by TDD have some quality assurance value as a secondary effect of the test, but that is not their primary goal. Their primary goal is to help you design the code, and to give you a safety net with which to refactor your code. Note that when I use the word refactor, I mean it in the classical sense: to change the internal implementation of a component without changing its externally observable behavior; that is, if you find yourself needing to change a unit test, you are not doing refactoring.

What TDD really is NOT

A replacement of all other testing activities. TDD will improve the quality of the code and will reduce the amount of bugs in the application but it is by no means there as an excuse to not perform any other testing.

Integration & Regression testing are still essential.
TDD is not a replacement for the process of designing your architecture. After all using a proper architecture is necessary for getting TDD to work in the first place so it is something that should always be done at the start of the project.

TDD Implementation

So now we have (thanks to SOLID principles, mainly Dependency Injection and the Repository pattern) a clean architecture that allows us to instantiate controllers and Model classes in our test project and pass to them fake repositories or other fake objects.

We have been asked to write something that validates a credit card number and returns the credit card type.

```
bool isValidCreditCard(string cardNumber, out string type)
{
        throw new NotImplementedException();
 }
```

Before we write any code we need to think what the correct behavior of the method should be and how we can test that behavior.

For simplicity let's assume that we only accept Visa and MasterCard and we check if Visa starts with digit 4 and if MasterCard starts with digit 5. If not, it is an invalid card number. To test correct behavior we need to pass a number starting with 4 and see if we get a true/visa back and also pass a number starting with 5 and see if we get a true/mastercard. Any other numbers/strings should return false. To create our test we will follow a common sense pattern called ***arrange/act/assert***

```
[TestMethod]
public void TestIsValidCreditCard()
{
        //Arrange by creating necessary objects and mocks
        CreditCardValidator ccv = new CreditCardValidator();
        string visaNumber = “408843048348048”;
        string masterCardNumber = “5345345353534”;
        string invalidCardNumber = “2234234242432342”;

        string visaType, msType, invalidType = “”;

        //Act, by calling the behavior to be tested
        bool isVisa = ccv.IsValidCreditCard(visaNumber, out visaType);
        bool isMs = ccv.IsValidCreditCard(masterCardNumber, out msType);
        bool isOther = ccv.IsValidCreditCard(invalidCardNumber, out msType);

        //Assert, by checking if you have what you expect
        Assert.IsTrue(isVisa);
        Assert.AreEqual(visaType, “visa”);
        Assert.IsTrue(isMs);
        Assert.AreEqual(msType);
        Assert.IsFalse(isOther);
        Assert.AreEqual(invalidType, “”);
}
```

Now that the Unit Test is written you can start coding the solution. The theory is that by writing the Unit test you have though a lot more carefully about the requirements, the correct behavior and all possible scenarios.

Remember that thanks to DI anything can be in a Test method now

```
[TestMethod]
public void TestPersonControllerUpdate()
{
        //arrange
        IPersonAccessor personAccessor = new FakePersonAccessor();
        PersonController controller = new PersonController( personAccessor);

        //test controller
        ……
}
```

TDD ShortComings

- Test-driven development is difficult to use in situations where full functional tests are required to determine success or failure. Examples of these are user interfaces, programs that work with databases, and some that depend on specific network configurations. TDD encourages developers to put the minimum amount of code into such modules and to maximize the logic that is in testable library code, using fakes and mocks to represent the outside world.
- Management support is essential. Without the entire organization believing that test-driven development is going to improve the product, management may feel that time spent writing tests is wasted.
- Unit tests created in a test-driven development environment are typically created by the developer who will also write the code that is being tested. The tests may therefore share the same blind spots with the code: If, for example, a developer does not realize that certain input parameters must be checked, most likely neither the test nor the code will verify these input parameters. If the developer misinterprets the requirements specification for the module being developed, both the tests and the code will be wrong.
- The high number of passing unit tests may bring a false sense of security, resulting in fewer additional software testing activities, such as integration testing and compliance testing.
- The tests themselves become part of the maintenance overhead of a project. Badly written tests, for example ones that include hard-coded error strings or which are themselves prone to failure, are expensive to maintain. This is especially the case with Fragile Tests. There is a risk that tests that regularly generate false failures will be ignored, so that when a real failure occurs it may not be detected. It is possible to write tests for low and easy maintenance, for example by the reuse of error strings, and this should be a goal during the code refactoring phase described above.
- The level of coverage and testing detail achieved during repeated TDD cycles cannot easily be re-created at a later date. Therefore these original tests become increasingly precious as time goes by. If a poor architecture, a poor design or a poor testing strategy leads to a late change that makes dozens of existing tests fail, it is important that they are individually fixed. Merely deleting, disabling or rashly altering them can lead to undetectable holes in the test coverage.

TDD Tools

There are 3 main areas for which tools may be required

- Dependency Injection Tools such as Ninject or Unity
- Unit Testing Tools such as built-in VS functionality or tools like xUnit
- Tools for creating mock objects such as Moq

Objectives Revisited

- Simplicity
- Clarity
- Elegance
- Modularity
- Extensibility
- Testability
- Flexibility
- Maintainability
- Business Value

Does TDD help us achieve some of these objectives? The answer is we will know at the end of the process! The theory is promising however as TDD is a process that encourages developers to go about their tasks properly.

- Structuring the application so that it can be properly unit tested is a first and perhaps in some cases the most important step towards improving the quality of your code. TDD encourages you to think about Dependency Injection, the Repository pattern and other important ways to improve your code.
- Encourages the developer to think about requirements before coding.
- Encourages the developer to think about the problem and all possible scenarios before coding.
- Encourages the developer to break a task into small easily testable units making it easier to adhere to the “single purpose” principle.
- Provides a verification process that will result in fewer bugs and better quality software.