Architecture Document

With Example Implementation

Michael Walfall

Version 1

Contents

1. Introduction

1.1 Purpose

- Describe the architecture of this application.
- Help programmers understand the design of the application and the techniques to be used to develop this application.
- Identify the application’s core components and their structure.
- Identify risks.
- Assist in the application’s project management.
- Clearly define interfaces and libraries for potential use in other applications.

1.2 Scope

1.2.1 Definitions, Acronyms and Abbreviations

- AL – Application Layer
- CRUD – Create, Read, Update and Delete
- DAL – Data Access Layer
- DDD – Domain Driven Design
- DI – Dependency Injection
- DL – Domain Layer
- DM – Domain Model
- EF – Entity Framework
- MVC – Model View Controller
- MVC2 – The second release of ASP.NET MVC
- MVC3 – The third release of ASP.NET MVC with the emphasis being Razor as the View Engine
- ORM – Object Relational Mapper
- PL – Presentation Layer
- POCO – Plain Old CLR Object
- RL – Repository Layer

- SAML – A city wide application security mechanism
- SL – Security Layer
- UI – User Interface

1.2.2 References

- *"Microsoft Application Architecture Guide, Second Edition"*
- *"Domain Driven Design: Tackling Complexity in the Heart of Software "*, Eric Evans
- *"Applying Domain-Driven Design and Patterns"*, Jimmy Nilsson
- *"Don't Create Aggregate Roots"*, Udi Dahan,
  www.udidahan.com/2009/06/29/don't-create-aggregate-roots
- *"Design of a Domain Model"*, Dino Esposito,
  msdn.microsoft.com/en-us/magazine/hh547108.aspx
- *"Programming Entity Framework"*, 2nd Edition, Julia Lerman
- *"Programming ASP.NET 3.5"*, Dino Esposito
- *"Dependency Injection in .NET"*, Mark Seeman
- *"C# in Depth, 2nd Ed."*, Jon Skeet
- *"ASP.NET MVC3 Custom Validation"* http://msdn.microsoft.com/en-us/vs2010trainingcourse_aspnetmvccustomvalidation#_Toc306346271
- *"Unobtrusive Client Validation in ASP.Net MVC3"*, Brad Wilson, bradwilson.typepad.com/blog/2010/10/mvc3-unobtrusive-validation.html
- *"The Complete Guide to Validation in ASP.NET MVC3 - Part 1"* http://www.devtrends.co.uk/blog/the-complete-guide-to-validation-in-asp.net-mvc-3-part-1
- *"The Complete Guide to Validation in ASP.NET MVC3 – Part 2"* http://www.devtrends.co.uk/blog/the-complete-guide-to-validation-in-asp.net-mvc-3-part-2
- *"Cohesion and Coupling"*, Jeremy Miller,
  msdn.microsoft.com/en-us/magazine/cc94717.aspx
- *"Performance Considerations for Entity Framework Applications"*,
  msdn.microsoft.com/en-us/library/cc853327(v=vs.90).aspx
- *"Specifications"*, Eric Evans and Martin Fowler, http://martinfowler.com/apsupp/spec.pdf

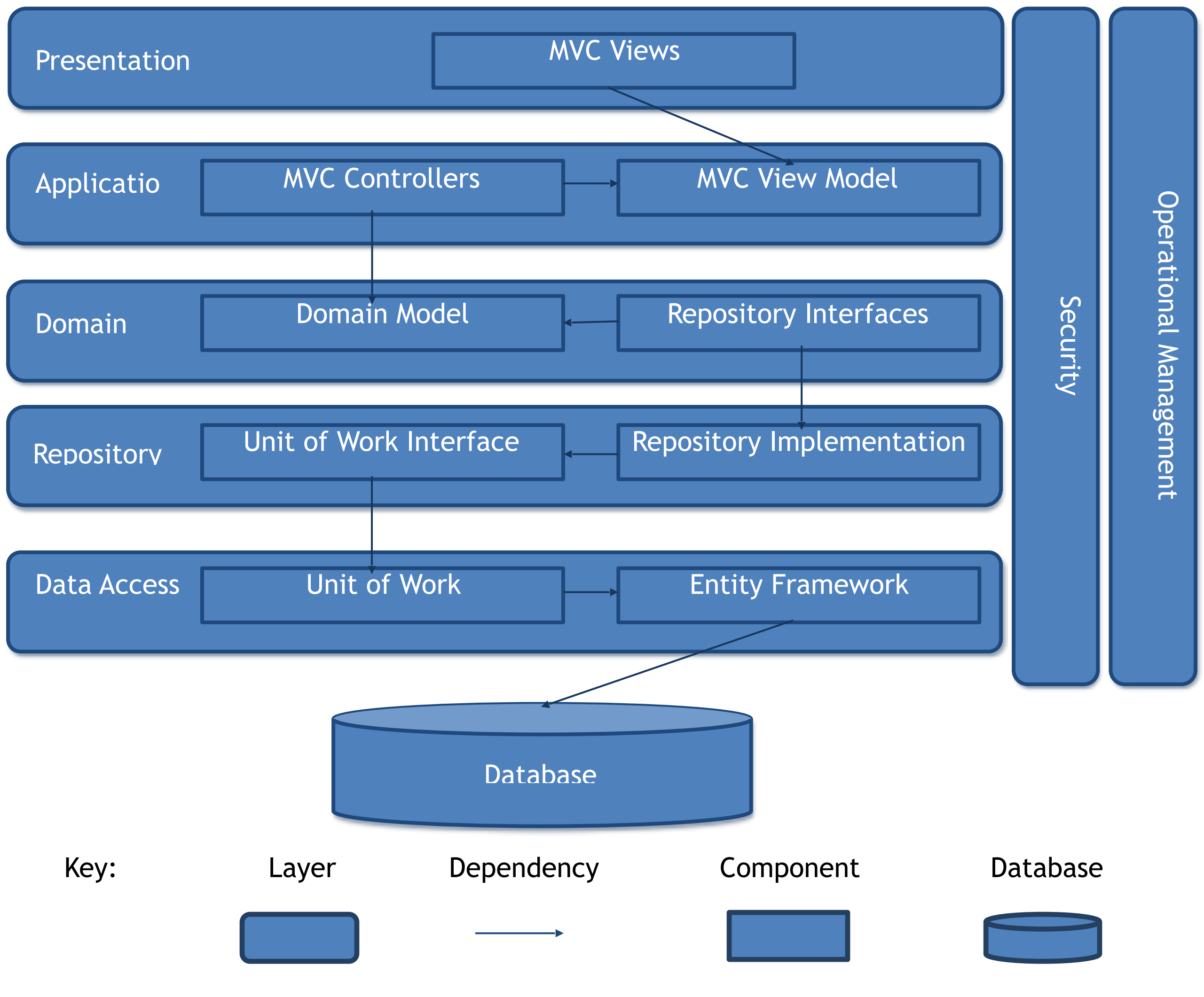
2. Architectural Solution and Representation

2.1 Overview

This section:

- Presents the application architecture.
- Discusses the rationale for the chosen architecture.
- Discusses alternatives considered and technical risks associated with implementing this architecture.

2.2 Architecture Logical Diagram



2.3 Architecture Logical Diagram Description

The **Presentation Layer (PL)** houses Views that are sent to a web browser. It is depicted as a separate layer to explicitly convey that it should not be assumed there will be only one UI platform. The application is designed to handle future UI platform requirements.

The **Application Layer (AL)** consists of ASP.NET MVC Controllers and the View Model. Views depend on View Model objects for their content. Within this architecture Views and View Models have a one-to-one relationship with each other. This design simplifies the maintenance of this layer.

Controllers are thin classes that orchestrate tasks performed during a request. They do not perform the actual work. Instead, they hand off the work to objects defined within this layer, the domain model, or a layer with cross-cutting concerns (i.e. Security).

The **Domain Layer (DL)** contains the domain model that is an abstraction of the target domain. It acts as the heart of the application and it is concerned with mimicking physical business procedures. It is designed irrespective of the interface. That is, the domain model is not concerned with how the UI interacts with it.

This application takes advantage of Domain Driven Design (DDD) techniques to implement the domain model. DDD is a software design technique and design philosophy that addresses the challenges of maintaining complex software applications.

The domain classes are implemented as POCOs. This decouples the domain classes from the Object Relational Mapper (ORM) and makes it possible to house the domain in its own layer. Consequently, the Domain Model (DM) is not dependent on any other layer or component in this application.

The DL also houses the repository interfaces that serve as a means for other layers to communicate with the DM in a decoupled manner.

The purpose of the **Repository Layer (RL)** is to implement the repository classes. This layer relieves the DL of all CRUD operations thus allowing the domain model to focus exclusively on modeling the domain.

This layer also contains the unit of work interface that serves as a contract depicting how ORMs are to communicate with this layer.

The **Data Access Layer (DAL)** houses the ORM, Entity Framework. This layer also contains the implementation the unit of work for the ORM.

Operational Management Layer (OML) and **Security Layer (SL)** house cross-cutting concerns. The OML contains functions such as caching and logging. Security is designed with the knowledge that at a future date it will be replaced by a city-wide security framework.

2.4 Layered View

2.4.1 Presentation Layer

2.4.1.1 Issues

- Identify frequently used display patterns so that client software patterns can be identified and code developed that can be reused by programmers.
- Design an intuitive, responsive, easy to use interface.
- Design with the long term maintenance of the UI in mind.
- Input validation.
- Reuse of code and partial view layouts.

2.4.1.2 Solution and Rationale

2.4.1.2.1 Custom Client Side Validation and Remote Validation

With the release of MVC3 it can be stated that the most significant scenarios for user input validation have been addressed by the Framework. This section covers custom client side validation and remote validation.

Custom Client-Side Validation

In situations where a form of validation not provided by the MVC Framework is needed, custom validation attributes can be written. A detailed discussion of using custom validation follows.

To serve as an example, a validation attribute that validates the length of a user id will be used. In the code below, the hi-lited line shows how the `UserId` property is decorated with the `LengthValidation` attribute. The attribute takes in a parameter indicating the minimum length for a `UserId`.

```
    public class Index
    {
        [DisplayName("User Id")]
        [LengthValidation(6)]
        public string UserId { get; set; }

        [Required]
        [DisplayName("User Name")]
        public string Name { get; set; }
    }
```

As the code is written, the default error message contained in the implemention of the attribute is used. However, if a different error message is desired, the following code shows how that can be done.

```
   [LengthValidation(6, ErrorMessage = "* Entry does not contain enough
characters.")]
```

Custom validation should be performed on the server and on the client when client-side validation is enabled. This is accomplished by extending the `ValidationAttribute` class and implementing the `IClientValidatable` interface.

The `LengthValidationAttribute` class, shown below, implements this setup. The first portion of the class extends `ValidationAttribute`. A default error message is defined that takes in the provided `UserId` minimum length as a parameter. `FormatErrorMessage` returns the error message string to be used when the validation criteria is not met.

`IsValid()` is the method in which the custom validation code is to be placed. In this case, the value entered by the user is checked to ensure it meets the minimum length requirement. If the requirement is not met the method returns false and the MVC `ModelState` object is populated with the error message.

`IsValid()` takes care of server-side custom validation.

```
[AttributeUsage(AttributeTargets.Property)]
public sealed class LengthValidationAttribute : ValidationAttribute,
IClientValidatable
{
    private int MinimumLength = 0;
    private const string _defaultErrorMessage =
                                      "* User Id must be at least {0} characters
long.";

    public LengthValidationAttribute(int minimumLength) :
base(_defaultErrorMessage)
    {
        MinimumLength = minimumLength;
    }

    public override string FormatErrorMessage(string name)
    {
        return String.Format(CultureInfo.CurrentUICulture,
                             ErrorMessageString,
                             MinimumLength);
    }

    public override bool IsValid(object value)
    {
        if (value == null) return false;

        string text = (string)value;
        if (text.Length < MinimumLength) return false;

        return true;
    }

    public IEnumerable<ModelClientValidationRule> GetClientValidationRules(
                                       ModelMetadata metadata, ControllerContext
context)
    {
        return new[] { new LengthValidationRule(ErrorMessageString,
this.MinimumLength)};
    }
}
```

To provide information about the validation attribute to the client validation framework the `GetClientValidationRules()` method is implemented.

Before going any further, an overview of the MVC3 custom validation framework would be useful. The previous code shows how server validation code is written. To have this happen on the client, the same logic is written as a JavaScript function. A Framework defined JSON object is used to pass server side data (error messages and parameters) to the JavaScript function.

On the client, MVC3 extends the jQuery Validation library to perform validation. It is a two step process:

- The first step is to implement the client-side JavaScript validation function and plug that function into the jQuery Validation framework.
- The second step is to provide the error message and any validation data defined in the server-side validation attribute class to the client code.

In the previous code snippet, `GetClientValidationRules()` serves as the means for the Framework to obtain the DTOs to be sent to the client. In this example, the error message and the minimum length are sent to the Framework so thae information can be provided to the equivalent client-side validation function.

As the code snippet shows, `GetClientValidationRules()` basically returns a list of `ModelClientValidationRule` objects. The `LengthValidationRule` class extends the `ModelClientValidationRule` class.

The code below shows the implementation of the `LengthValidationRule` class. In this case, three properties in that class are populated with information utilized by the client-side validation function. They properties are:

- `ErrorMessage`: contains the error message that is to be displayed. This is the same message that would be displayed if client-side validation was not enabled.

- `ValidationType`: This is the type assigned to the client-side validation function. It is considered a type because the same validation attribute can be applied to different properties in the View Model. On the client, the same validation type can be assigned to different fields. The validation type is then associated with the validation function.

- `ValidationParameters`: The values that are to be used to perform the validation. In this case, it is the minimum length of a User Id.

```
public class LengthValidationRule : ModelClientValidationRule
{
    public LengthValidationRule(string errorMessage, decimal minimumLength)
    {
        ErrorMessage = errorMessage;
        ValidationType = "lengthvalidation";
        ValidationParameters.Add("minlength", minimumLength);
    }
}
```

At this point, the server-side validation attribute class has been written and the DTOs have been written.

The following code snippet contains the client-side code. The first method, addMethod, is used to plug the custom validation method into the jQuery validation framework. A definition of the method follows

- `addMethod( ruleName, methodImplementation, errorMessage ):`

  In this example since the error message is being provided by the server validation attribute object, `errorMessage` is set to an empty string.

```
$(function () {
    jQuery.validator.addMethod(
        'lengthvalidation',
        function (value, element, param) {
            var maxlength = param;
            if (value == null) return false;
            var text = String(value);
            if (text.length < maxlength)
                return false;
            else
                return true;
        },
        ''
    );

    jQuery.validator.unobtrusive.adapters.add(
        "lengthvalidation",
        ["minlength"],
        function (options) {
            options.rules["lengthvalidation"] = options.params.minlength;
            options.messages["lengthvalidation"] = options.message;
        });
} (jQuery));
```

- `jQuery.validator.unobtrusive.adapters.add` (adapterName, [“params”], function)

The `UserId` field has a client side validation rule attached to it. Consequently, in the HTML5 markup generated by the Razor engine,it receives the `data-val="true"` attribute.

```
<div class="editor-field">
    <input class="text-box single-line"
        data-val="true"
        data-val-lengthvalidation="* User Id must be at least {0} characters
long."
        data-val-lengthvalidation-lengthvalidation="6"
        id="UserId" name="UserId" type="text" value="" />
    <span class="field-validation-valid"
        data-valmsg-for="UserId" data-valmsg-replace="true">
    </span>
```

```
</div>
```

Remote Validation

This approach is used in situations where a user needs to know in advance of submitting a View if certain information is valid. An example would be entering a vehicle plate number to ensure it is not already in use. The code below shows the View Model containing the MVC3 RemoteAttribute.

```
    public class Index
    {
        [Required]
        [DisplayName("Plate Number")]
        [Remote("IsPlateNumberAvailable", "RemoteAjax")]
        public string PlateNumber { get; set; }
    }
```

The first parameter is the action method Remote is to use. The second attribute is the name of the Controller.

The Controller code is as follows:

```
        public JsonResult IsPlateNumberAvailable(string PlateNumber)
        {
            if (PlateNumber == "abc-123")
                return Json("* Plate Number is already being used.",
                             JsonRequestBehavior.AllowGet);
            else
                return Json("* Plate Number can be used.",
JsonRequestBehavior.AllowGet);
        }
```

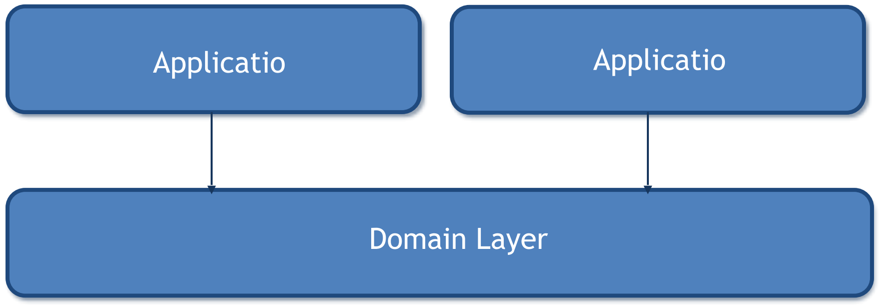
2.4.1.2.2 Client Side Error Message Dialog Box Framework

2.4.2 Application Layer

Introduction

The fundamental purpose of an application layer is to map domain objects to data objects that can be used by a presentation layer or used by another application. In the initial design the AL is producing and accepting View Model objects that are used by Views. The Views are designed to meet the needs of the business area communicating with the Domain Model.

The Domain Model is designed irrespective of the AL. The design goal for the DM is to be an abstraction of how a the entire business is run. More than one AL may seek to access the information contained within a Domain Model.



With this design, when faced with a situation where the developed interface does suit their needs a new application layer is developed. The Domain Model remains the same; tts data needs to be viewed in a different manner.

2.4.2.1 Issues

- Orderly development of the application with significant code reuse.
- Long term maintenance and enhancement. To develop a design that articulates where new code should go and where to find existing code.

2.4.2.2 Solution and Rationale

The primary purpose of the Application Layer is to:

- Orchestra the processing of requests.
- Map objects between the Domain Model and the View Model.
- Provide server-side validation.

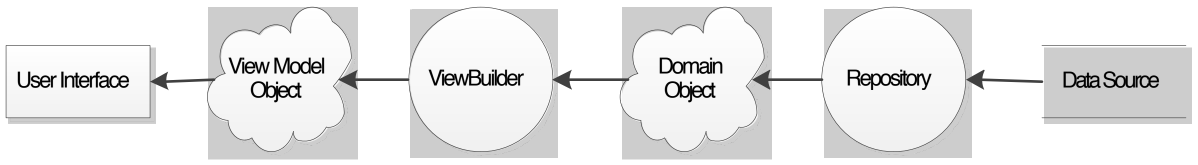
Object Interaction Diagram

An introduction to the primary object classifications in the Application Layer is warranted and useful when visualizing the use of this layer's classes. (Examples of Application Layer classes being: View Model, ViewBuilder, Factory and Repository classes.)

Definition

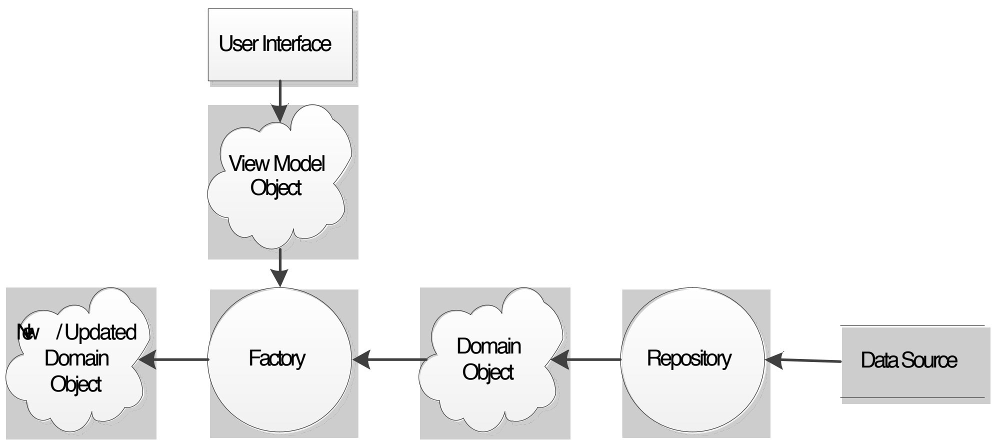
- A **ViewBuilder** takes in Domain objects and produces View Model objects that are sent to the client. It uses repository to obtain persisted Domain objects.
- A **Factory** takes in View Model objects to create Domain objects.

The *Object Interaction Diagrams* show the interaction of objects within the Application Layer. A Controller object, which is not shown in the diagrams, orchestras all the tasks performed during a request operation.



ViewBuilder Object Interaction Diagram

- To provide content to the user interface a ViewBuilder object reshapes Domain objects to create View Model (VM) objects that map cleanly to a View that is sent to a user interface (UI). The ViewBuilder relies on a Repository object to obtain Domain objects from the datasource.



Factory Object Interaction Diagram

- When responding to a UI request, the MVC framework populates a VM object with the UI's data. This VM object is then consumed by a Factory object. The Factory knows how to transform the VM object into a Domain object. It also instructs Domain objects to validate themselves before they can be made available to the rest of the application. (See Entity Validation Framework for details.)

- The Factory object also merges previously saved Domain objects with updated data contained in a corresponding VM object returned from the client. It instructs a Repository object to retreive presisted data from the data source, updates the retreived Domain object with the data contained in the VM object, and then has the updated Domain object validate itself.

Controller Design Guidelines

- Controllers should be thin and orchestrate the tasks that are to be performed during a request. The actual work should be performed by previously mentioned objects within the application.

- A programmer should be aware of situations where a Controller is performing a set of tasks on behalf of a set of Domain objects. For example, in another application the userId is stored in two different objects with different encryption. Which object should responsible for the updating of the other object? The Controller could orchestra this updating but this places domain logic in the Controller. Instead, a Domain Service object should oversee the operation. The Controller would call a method on the Domain Service and the service would know how to update the objects.

- Having the Controller focus on calling methods on different objects instead of actually performing the work promotes the reuse of code and simplifies the maintenance of the application.

View Model Design Guidelines

- Only View Model objects will be sent to a View. This decouples the View for the Domain. It promotes the unit testing of view content and enables the View to be developed independently of the Domain. In other words, Domain Model objects should never be used directly within a View.

- There will be a one-to-one relationship between a View and a View Model object for the following reasons:

  - It simplifies the maintenance of this application since only the data required for a single View is contained in a View Model object. Having data related to multiple views in a single View Model object complicates the maintenance of the Views and the View Model object.

  - It improves the performance of the application since only the required data is transmitted to the client.

  - It simplifies testing of a view's content. A View Model object decouples the Application Layer from the Presentation Layer. This enables the View to be tested with mock data. It also enables ViewBuilders, which generate View Model content, to be unit tested irrespective of the View.

View Model Design Format

Frequently, when working with Views certain fields are used to display data while other fields are used to capture and return data. In these situations the following format should be followed:

- Separate display data from input data.

- Define the input properties in a separate class.

The code snippet below provides an example. It shows the display data defined separately from the input content. The input properties are defined in the `VehicleInputViewModel` class.

```
public class CreateViewModel
{
    public IList<SelectListItem> StatusList { get; set; }
    public IList<SelectListItem> ConfigList { get; set; }
    public IList<SelectListItem> HseLocList { get; set; }

    public VehicleInputViewModel Input { get; set; }

    public class VehicleInputViewModel
    {
        [DisplayName("Status: ")]
        public string SelectedStatus { get; set; }
```

```
            [DisplayName("Transmission SN: ")]
            public string TransmissionSN { get; set; }

            [DisplayName("Fuel: ")]
            public string Fuel { get; set; }
        }
    }
```

Vehicle CreateViewModel

This approach makes the View Model class easy to understand. The first three properties contain content for drop down lists. The embedded `VehicleInputViewModel` class contains the input properties.

If there is another View Model class with identical input properties the `VehicleInputViewModel` class can be housed in its own file and reused.

Note that the list properties are of type `IList<SelectList>`. This is a MVC defined type that can be cleanly used by `HTMLHelper` classes. The code below shows how the list is created from a Repository object.

```
        private IEnumerable<SelectListItem> GetStatusList()
        {
            var statusList = _repository.GetStatusList();

            var list = new List<SelectListItem>();
            list.AddRange(statusList.Select(item => new SelectListItem
            {
                Value = item.Id,
                Text = item.StatusName
            }));

            return list;
        }
```

Building the DropDownList.

In the following code snippet the list is cleanly consumed by the `Html.DropDownListFor` method.

```
<tr>
  <td>@:Html.LabelFor(model => model.VehicleInputViewModel.SelectedStatus)</td>
  <td>
    @:Html.DropDownListFor(model => model.VehicleInputViewModel.SelectedStatus,
      Model.StatusList)
  </td>
</tr>
```

Displaying the List.

The final code snippet shows the benefit of housing the input properties in their own class. When the View is submitted, the input data is validated and consumed by the Controller as follows:

```
      [HttpPost]
        public ActionResult Create(CreateViewModel.VehicleInputViewModel
viewModel)
        {
            if (!ModelState.IsValid)
        …
```

Validating the returned data.

The primary benefit with this View Model design format is when validation errors occur and the View Model object has to be resent to the client. The design clearly states what attributes need to be repopulated during an update. (The section entitled "ViewBuilder Guidelines" will go into more detail on this subject matter.)

Development Tool: AutoMapper

The AutoMapper library will be used to promote code reuse and reduce code written to map Domain Model objects to View Model objects.

Previously, the code snippet entitled "Building the DropDownList" was used to build a vehicle status dropdown list. Using AutoMapper the method `GetStatusList()` can be replaced by the following code:

```
        createViewModel.StatusList = Mapper.Map<IEnumerable<StatusModel>,
                    Ienumerable<SelectListItem>>(_repository.GetStatusList())
```

AutoMapper being used to create a list.

Mapping between classes containing properties are defined in the AutoMapper `GlobalProfile` class. AutoMapper uses the convention that properties with the same name are mapped to each other. Otherwise, mappings can be explicitly defined and reused throughout the application. Here is an example of the Vehicle Status mapping being explicitly defined.

```
            CreateMap<StatusModel, SelectListItem>()
                .ForMember(d => d.Text, o => o.MapFrom(x => x.StatusName))
                .ForMember(d => d.Value, o => o.MapFrom(x => x.Id));
```

Explicitly defining the relationship between properties to be mapped.

The `SelectListItem` class has two properties that are to be populated, `Text` and `Value`.

ViewBuilder Guidelines

- A `ViewBuilder` class should be associated with a `Controller` class. Each method on the `ViewBuilder` should be associated with a View and return a View Model object for that View. For example:

- A `VehicleController` contains an `Edit()` action.
- By the MVC3 convention, the `return View()` statement will return a view named Edit.chtml.

Adhering to the MVC convention over configuration principle, all Vehicle View related View Models are placed in a Vehicle folder. For example, that folder that could contain:

1. An `EditViewModel` class for the `Edit` view.
2. A `VehicleViewBuilder` class that contains a `CreateEditViewModel` method and a `ReCreateEditViewModel` method. The first method is used to create an `EditViewModel` object on the initial request from the client. On subsequent requests, the data entered by the client must be preserved while drop down list related data must be repopulated in the `EditViewModel` object. Because of the previously mentioned View Model Design Format, the code needed to recreate a view is very clean and easy to understand.

Such an approach simplifies application maintenance. Each method has a single responsibility. Therefore, there would be only one reason to change the method.

- `ViewBuilders` are stateless objects that act as services. Any dependencies they need should be injected into them. In addition, these parameters should be defined as interfaces. This decouples the ViewBuilders from the implementation of the interfaces.

Factory Design Guidelines

- A Factory encapsulates the knowledge needed to construct new non-trivial Domain objects and a valid object graph from an aggregate root. It returns these objects in a valid state with respect to themselves and other Domain objects. (See Aggregate Roots for details.)
- Factories should be aligned with aggregate roots and return a reference to the root of that Domain object's graph.
- Factory methods (operation) must be atomic.
- They should return complete objects that are ready to be saved by the unit of work.
- In this application, the Factories reside in the Application Layer. They take View Model objects as an input parameter and return Domain objects. Placing the Factories in the Domain Layer would cause the Domain to be dependent on the Application Layer since ViewModel classes are defined in the AL.
- A Factory does not perform the actual validation of a DM object. This task is handled by the DM object's `IsValid` property when called by the Factory. (See Entity Validation Framework for details.)
- A Factory assists in ensuring that a domain object is in a valid state with respect to another domain object. For example, suppose there is a rule stating each Vehicle object must have a unique license plate number. After the Vehicle object is created, the Factory would use an associated Repository object to provide the Vehicle object

with the information it needs to perform the plate validation rule. This approach ensures the decoupling of the Domain object from the Factory. It enables unit testing of the Domain object's IsValid() method irrespective of the Factory since the data provided by the Factory could be mocked.
- A Factory ensures that all validation is performed by all objects in the aggregate.

Loosely Coupled Classes: Dependency Injection

This application will use Dependency Injection (DI). DI consists of software design principles and patterns that promote the development of loosely coupled classes. It addresses the issue of object interacting with each other such that changes to an object will have no or minimum effect on a dependent object. Under DI:

- Object dependencies are to be injected into the object that uses them. Dependencies are not to be instantiated from within that object. For example, a ViewBuilder uses repositories to handle retrieval of objects from the database. The repository objects are to be injected into the ViewBuilder either through constructor injection (parameter in the constructor) or method injection (parameter in the method).

- The type of the input parameter being injected should always be an interface. For example, the constructor for the `VehicleViewBuilder` could be `VehicleViewBuilder(IVehicleRepository repository)`. It should not be an implementation of the `IVehicleRepository` interface. By using an interface the implementation can be changed and the `VehicleViewBuilder` code remains the same.

- When an object contains some functionality that assists in an operation that does not seem to logically fit the description of that object, that code should be placed in a separate class and injected into the dependent class. For example, an employee object may contain an encrypted UserId property. There should be an object that provides encryption services and it should be injected into the employee object.

- This application will use StructureMap, a DI container that facilitates the centralized "wiring up" of an application's objects. The use of interface based parameters with a DI container makes it possible to have a centralized location were the implementation of an interface can be defined. In addition, the definition is adhered to throughout the application.

2.4.3 Domain Layer

2.4.3.1 Issues

- Status Transition Rule Engine implementation.

- Implementation of a domain object validation framework that integrates with the MVC Framework.

- Designing the domain so that overtime it is maintainable.

- Develop and articulate a design that indicates where new code should go and where to find existing code.

2.4.3.2 Solution and Rationale

A depiction of the Domain model is presented in the Class Diagram section. This section focuses on concepts related to the design that facilitates the simplification, integrity, and maintenance of the Domain Layer.

2.4.3.2.1 Domain Model

The Domain Model (DM) is to be designed with no consideration for the UI. The DM is an abstraction of the physical domain that attempts to mimic the physical domains business processes. The Model will consist of the following types of classes:

- *Entities* are objects that have an identity and have continuity throughout a life cycle. A Vehicle and an engine is an example of an entity. They are entities because it has been decided by the domain designer that there is a need to uniquely identify each object. Presented with two engines it is important to know *which* engine is *which*. All the data contained for the two engines could be the same, yet *for this particular application* it is determined that it is still important to distinguish between the two.

- *Value objects* do not need to be uniquely identified. Vehicle Status is an example of a value object. What is of importance is *what* a status is not *which* status is *which*. Given two status objects containing the same data *for this particular application* they are considered the same status object.

- A *Domain Services* are objects that most often perform operations that require coordination between more than one domain object. They are defined by the service they perform for their clients. Services tend to coordinate operations between domain objects. These operations are stateless. Consequently, Domain Services consist of only methods and no properties. The type of operations services performs relates to a domain concept that does not fit naturally as part of an entity or value object. Its interface is defined in terms of other objects in the Domain Model.

Note that in the description of entities and value objects "*for this particular application*" is emphasized. This is to stress that the same entity or value object in one application may not be considered an entity or value object in another.

2.4.3.2.2 Aggregate Roots

The aggregate root is a concept that has proven to be very useful in another application and will be implemented in this application[1].

"*An AGGREGATE is a cluster of associated objects that we treat as a unit for the purpose of data changes. Each AGGREGATE has a root and a boundary. The boundary defines what is inside the AGGREGATE. The root is a single, specified ENTITY contained in the AGGREGATE. The root is the only member of the AGGREGATE that outside objects are allowed to hold references to, although objects within the boundary may hold references to each other.*" (Evans, pg. 126-127)

[1] Using the aggregate root concept the time to display an Order Index View was reduced from approximately 11 seconds to approximate 0.5 seconds.

The aggregate root concept:

- Simplifies the design of the application.
- Promotes the data integrity of the application. (See Data Integrity for details.)
- Enhances the performance of the application. The persisting/retrieval of an object via the aggregate root cause all the necessary data to be saved to or retrieved from the database in a single trip.
- Takes advantage of the features offered by ORMs such as Entity Framework. EF provides the means to load specific parts of an object graph in a single query. In addition, EF is capable of persisting the entire graph when the root is persisted. (See Why the Unit of Work Pattern for details.)

The guideline for aggregate roots is as follows:

- The root entity has global identity.
- When changes to any object within an aggregate are committed all invariants[2] of the whole aggregate must be validated. (This is why Factories will be aligned with aggregate roots.)
- Entities inside the boundary have local identity unique only within the aggregate.
- An object outside the boundary cannot hold a reference to an object inside the boundary. These objects are obtained via the root.
- Only aggregate roots can be obtained directly from the database. That means repositories should be aligned with aggregate roots. There should not be a repository for every object in the Domain Model.
- Objects within the aggregate can hold references to other aggregate roots.
- It's up to the developers of the Domain Model to determine where an aggregate begins and ends.
- A delete operation must remove everything within the aggregate boundary at once. This can be accomplished in two ways. The `Clear()` method can be called on the child collection or cascading deletes can be setup in the database. The objective is to preserve the integrity of the database by not having orphaned children objects or deleting objects that are referenced by other others. For example, when a vehicle is deleted the status object should not be deleted.

2.4.3.2.3 Entity Validation Framework

The MVC Framework provides client side and server side data validation through the use of attributes that can be attached to properties in View Model objects. This operation ensures the View Model object is in a valid state. After mapping the View Model object to a Domain Model object it is desirable to validate the state of the newly created domain object.

This section demonstrates a framework that entails domain objects knowing how to validate themselves and then notifying the MVC Framework of any violations so that information can

[2] Consistency rules that must be maintained whenever data changes.

be passed to the client via the MVC Framework's `ModelState` class. It is presented from the point of view where a programmer wants to develop entities that take advantage of the Validation Framework.

Validation Framework Requirements

Any solution should:

- Take advantage of the features offered by the MVC Framework. MVC provides Data Annotations and a set of classes that provide a means to validation data objects based on the Data Annotations.
- Be easy to use. Ideally, a single method code on a domain object should handle validation.
- Follow MVC conventions. In this case, it should be used similarly to the MVC `ModelState.IsValid` property.

How to Setup Validation of a Domain Class

In addition to entities having state and behavior they should also know how to validate themselves. Here is how it is done.

This application uses POCOs that are generated by a T4 template. This particular T4 template knows how to take the EF Entity Model metadata and create Domain Classes for each Entity in the model. When the EF Entity Model is updated and saved, the T4 template overwrites the existing class files with the updates. To overcome this limitation, three partial classes are used when applicable. The content of the classes are separated into:

- Properties
- Behavior
- Validation

Note, because of the way this architecture is designed Behavior and Validation partial classes need to be created only when necessary.

This example focuses on the Vehicle properties and the Vehicle validation partial classes. The following code shows a Vehicle partial class with its properties and constructor.

```
public partial class Vehicle
{
    public string EngineSN { get; private set; }
    public string TransmissionSN { get; private set; }
    public string Fuel { get; private set; }

    public Vehicle(string engineSN, string transmissionSN, string fuel)
    {
        EngineSN = engineSN;
        TransmissionSN = transmissionSN;
        Fuel = fuel;
    }
}
```

Implementation of a Vehicle class

The validation portion of the Vehicle class, shown below, is contained in a partial class file named Vehicle. The content of the file is explained.

- `Vehicle` inherits from the `ObjectValidator<T,Q>` class. The `ObjectValidator<T,Q>` knows how to have the DataAnnotations library to validate the properties in the class. The `ObjectValidator` class takes two types. The first is the domain class and the second is that domain class's metadata class. This class also contains an important property named `IsValid`. When this property is checked it causes validation to be performed. (Details to follow.)
- The second class, `VehicleMD`, has the metadata that defines how the properties are to be validated. (MD stands for metadata.) Note that all the properties in this class are of type `object`. Validation attributes are used to assign validation rules.

This approach promotes code reuse since the same validation attributes can be used for ViewModel classes.

```
public partial class Order : ObjectValidator<Order, OrderMD>
{}

public class OrderMD
{
    [Required(ErrorMessage = "An Order Date is required.")]
    public object OrderDate { get; private set; }

    [Required(ErrorMessage = "A Customer is required.")]
    public object CustomerId { get; private set; }

    [Required(ErrorMessage = "An Item is required.")]
    public object Item { get; private set; }
}
```

The Validation portion of the Oder class

That ends the description of how to setup a class validation. The next step is to describe how this setup can be used within the MVC Framework.

How to use the Validation Framework

The following code shows how the Validation Framework (VF) plugs into the MVC Framework. After the `VehicleViewModel` is validated a `Vehicle` object is created. The Vehicle's `IsValid` property is checked to determine its validity. If violations are found, the `UpdateModelStateWithRuleViolations` extension method is used to populate the MVC `ModelStateDictionary` with the violations.

`ModelState` is a property on the controller class that returns a `ModelStateDictionary` object. This object represents the state of an attempt to bind the data returned from a View to a ViewModel object that contains validation rules. If errors are found, the object's IsValid property is set to false.

Note that `UpdateModelStateWithRuleViolations` is not a method of the ModelState class. It is an extension method that is basically a special type of static method that acts as an instance method of the ModelStateDictionary class . The C# compiler attaches the method to the ModelStateDictionary at compile time.

```
        [HttpPost]
        public ActionResult Create(OrderViewModel viewModel)
        {
            if (!ModelState.IsValid)
                return View(viewModel);

            var order = new Order(viewModel.OrderDate,
                                  viewModel.CustomerId,
                                  viewModel.Item);
            if (!order.IsValid)
            {
ModelState.UpdateModelStateWithRuleViolations(order.RuleViolations);
                return View(viewModel);
            }
            else
                return RedirectToAction("Index");
        }
```

VF integration with MVC Framework

To summarize, validation occurs when the `IsValid` property is checked.

Interface Descriptions

For completeness the code for the `ModelStateDictionary` extension method is shown below followed by the code for the `ObjectValidator<T,Q>` class.

```
    public static class ModelStateUpdateService
    {

        public static void UpdateModelStateWithRuleViolations(
                                         this ModelStateDictionary modelState,
                                         IEnumerable<ValidationResult> violations)
        {
            foreach (var issue in violations)
            {
                modelState.AddModelError(issue.MemberNames.First(),
issue.ErrorMessage);
            }
        }
    }
```

The UpdateModelStateWithRuleViolations extension method

```
static void UpdateModelStateWithRuleViolations(
                   this ModelStateDictionary modelState,
                   IEnumerable<ValidationResult> violations)
```

- The first parameter identifies the class that the extension method is to attach to. The second parameter is the list of rule violations that the domain object returned. The `ValidationResult` class is defined in the `DataAnnotations` namespace.

For more on extension methods see, "*C# in Depth*", Jon Skeet, as listed in reference.

The code for the `ObjectValidator` class follows.

```
    public class ObjectValidator<T,Q>
    {
        private IEnumerable<ValidationResult> _result = null;

        public IEnumerable<ValidationResult> RuleViolations
        {
            get { return _result; }
        }

        public bool IsValid
        {
            get
            {
                var result = Validate();
                if (result == null)
                    return true;
                else
                {
                    _result = result;
                    return false;
                }
            }
        }

        private IEnumerable<ValidationResult> Validate()
        {
            TypeDescriptor.AddProviderTransparent(new
                      AssociatedMetadataTypeTypeDescriptionProvider(
                                                 typeof(T), typeof(Q)), typeof(T));
            var context = new ValidationContext(this,
                                                 serviceProvider: null,
items:null);
            var results = new List<ValidationResult>();
            var isValid = Validator.TryValidateObject(this, context, results,
true);
            return (isValid) ? null : results;
        }
    }
```

The ObjectValidator class

`ObjectValidator<T,Q>`

Is a generic class that takes two parameters:

- T: the domain class.
- Q: the metadata for that class. This class contains the validation attributes that are to be used for validation.

`ObjectValidator` used by all domain classes that require validation.

`IEnumerable<ValidationResult> RuleViolations`

- A property that returns a list of rule violations. It is used by the extension method `UpdateModelStateWithRuleViolations` to obtain the list of rule violations.

`bool IsValid`

- A property that is set to false if rule violations are found. Otherwise, it is set to true.

`IEnumerable<ValidationResult> Validate()`

Note: All the mentioned classes in this section, with the exception of `TypeDescriptor`, are located in the `System.ComponentModel.DataAnnotations` namespace.

A detailed description of the `Validate()` method follows:

1. The ASP.NET TypeDescriptor architecture enhances the capabilities of .NET reflection by allowing metadata for an instance of a class to be added at runtime. In this particular case, the added metadata consists of attribute validation rules. This design allows the same code to be used to validate different types of objects.
2. The `AssociatedMetadataTypeTypeDescriptionProvider` class provides a means to extend the metadata definition. The first parameter is the class type whose metadata is to be extended and the second parameter is the class containing the metadata to be added.
3. The `TypeDescriptor.AddProviderTransparent()` method adds the type description to the dll component. (`TypeDescriptor` is located in the `System.ComponentModel` namespace.)
4. The `ValidationContext` class is used to create a context within which the object will be validated. For this discussion only the first parameter is of importance, which takes an instance of the object to be validated . In this example that object is identified by the **`this`** property of the class since every entity extends the ObjectValidator class.
5. An object of type `List<ValidationResult>()` is created. This object will be populated with rule violations.
6. Finally, the static method `Validator.TryValidateObject()` is called to perform the validation. The parameters are: an instance of the object to validate, the validation context, returned violations (if any) and a flag indicating if all the attributes are to be validated.

Extensibility: Adding Custom Rules

Custom validation attributes can be created to perform validation that is not provided by the DataAnnotations namespace. This is accomplished by extending the `ValidationAttribute` class as shown below. The aspects of the extended class are discussed.

```
[IsNumeric(ErrorMessage="* Reference Number must be numeric.")]
public virtual string ReferenceNumber { get; set; }
```

Customer Validation Attribute Usage

The `IsNumeric` ValidationAttribute checks to ensure the entered data is numeric. If it is not valid, the defined error message is sent to the client.

```
[AttributeUsage(AttributeTargets.Field | AttributeTargets.Property,
                AllowMultiple = false,
                Inherited = true)]
public sealed class IsNumericAttribute : ValidationAttribute
{
    private const string _defaultErrorMessage = "'{0}' must be numeric.";

    public IsNumericAttribute() : base(_defaultErrorMessage) {}

    public override string FormatErrorMessage(string name)
    {
        return String.Format(CultureInfo.CurrentUICulture,
                             ErrorMessageString,
                             name);
    }

    public override bool IsValid(object value)
    {
        Regex isNumber = new Regex("[^0-9]");
        string valueAsString = value as string;
        if (String.IsNullOrEmpty(valueAsString))
            return true;
        else
            return !(isNumber.IsMatch(valueAsString));
    }
}
```

Custom attribute example.

- Custom attributes extend the `ValidationAttribute` class.
- The usage of the attribute is defined. In this case, the attribute can be used with fields and properties.
- As a best practice, the format for a default error message is defined and sent to the base class during construction of the object.

The base class method that must be override follows:

`string FormatErrorMessage(string name)`

- Used to format the error message. `ErrorMessageString` is defined in the base class. Referring to the code showing how the IsNumericAttribute class is used, if `ErrorMessage` is defined the content of that variable is used instead of `_defaultErrorMessage`. (See the code in “Custom Validation Attribute Usage” for example of how ErrorMessage is set.)

`bool IsValid(object value)`

- This method contains the custom validation code and must return a Boolean value.

2.4.4 Repository Layer

2.4.4.1 Issues

- Data integrity. Repositories should be designed in accordance with the defined relationships between DM objects in the Domain Model.
- Reuse of code. Repository classes perform the same CRUD operations against different tables. However, there are situations where queries unique to a domain object are required. The design should consolidate the code needed for basic CRUD operations while being able to accommodate unique situations.
- Decoupling. It should be possible to change the implementation of the repositories without affecting the Domain Layer.
- Performance. Minimize database retrieval time. Lazy Loading is designed to improve the response time for stateless type applications by retrieving the minimum amount of information during a trip to the database. However, there are situations where it is known in advance what related data will be needed. In these situations, all relevant data should be retrieved during the initial trip to the database. The repositories should be designed to provide a standard way of doing this.
- Performance. The processing of queries should be performed as close to the data source as possible. In addition, only data that meets the query criteria should be returned from the data source.

2.4.4.2 Solution and Rationale

- Repositories reduce the complexity of an application by encapsulating the code needed to perform CRUD operations.
- To further reduce complexity repositories should reflect domain model concepts. For example, an acceptance letter has no meaning unless assigned to a vehicle. This relationship can be thought of as a logical unit within the domain. An aggregate root depicts a set of objects that form a logical unit. A repository should return logical units. Therefore, repositories should be aligned with aggregate roots.

The following patterns are implemented in this layer:

- Repository
- Unit of work
- Dependency Injection

The use of these patterns along other design techniques that will be presented, address the issues raised for this layer.

2.4.4.2.1 Data Integrity

Repositories will be designed and used to take advantage of the DDD concept of an aggregate root. For example, an Acceptance Letter has no meaning unless it is associated with a vehicle. Therefore, when creating an acceptance letter a valid vehicle object is a prerequisite and when persisting an acceptance letter it is the vehicle that is persisted. An outcome of this logic is that an acceptance letter repository is not needed since the root is the vehicle. CRUD operations should be done via a vehicle repository.

This sort of design promotes data integrity since an aggregate root defines what constitutes a complete domain object, thus reducing the likelihood of orphaned children in the database. In addition, validation begins at the root so not only is an object's property data validated but parent-child validation rules can also be validated in factory objects.

As per data integrity the outcome of using the aggregate root concept is a multi-layered integrated scheme to ensure the correctness of the application's data. The design promotes higher cohesion and looser coupling.

2.4.4.2.2 Decoupling: Unit of Work Interfaces

The unit of work interface consists of two interfaces. The base interface is `IUnitOfWork` that contains one method, `Save()`. The implementation of this method calls the save method associated with the ORM being used.

```
public interface IUnitOfWork : IDisposable
{
        void Save();
}
```

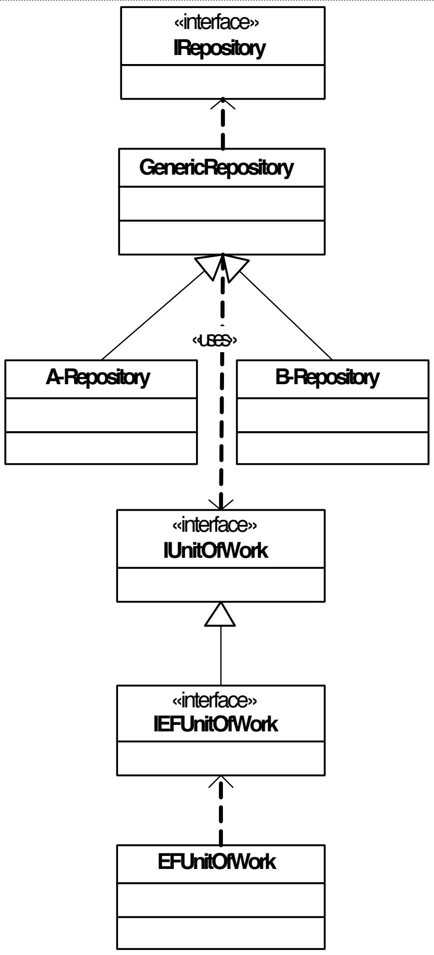
Unit of Work Base Interface

One level above `IUnitOfWork` is an interface that is used to house the context object implemented by the particular ORM. A description of the interface for Entity Framework is shown below. "T" represents the type of the particular unit of work; for Entity Framework that would be the `ObjectContext` object. These interfaces consist of one method named `GetContext()` that is used by repositories to extract the context to perform CRUD operations against it.

```
public interface IEFUnitOfWork<T> : IUnitOfWork
{
        T GetContext { get; }
}
```

EF Interfaces

Repository – Unit of Work Class Diagram



2.4.4.2.3 Code Reuse: Generic Repository Implementation

The design of the repository layer consists of a generic repository and a set of specialized repositories. The generic repository performs operations that are common for all repositories, such as: update, add and delete. The following code shows the implementation of the generic repository.

```
    public class GenericRepository<T> : IRepository<T> where T : class
    {
        private readonly ObjectContext _context;
        private IObjectSet<T> _objectSet;
        private readonly IEFUnitOfWork<ObjectContext> _unitOfWork;

        public GenericRepository(IUnitOfWork uoW)
        {
            if (uoW == null)
                throw new ArgumentNullException("GenericRepository: UoW not 
Provided");
            _unitOfWork = (IEFUnitOfWork<ObjectContext>)uoW;
            _context = _unitOfWork.GetContext;
        }
        public IQueryable<T> GetQuery()
        {
            return this.ObjectSet;
        }

        public IList<T> GetAll(Func<IQueryable<T>, IOrderedQueryable<T>> orderBy =
null)
        {
            if (orderBy == null)
                return this.ObjectSet.ToList();
            else
                return orderBy(this.ObjectSet).ToList();
        }

        public IList<T> Find(Expression<Func<T, bool>> whereFilter, string[]
include)
        {
            return include != null
                ? this.ObjectSet.Include<T>(include).Where(whereFilter).ToList()
                : this.ObjectSet.Where(whereFilter).ToList();
        }

        public T GetSingle(Expression<Func<T, bool>> expressionFilter,
                           string[] includeList)
        {
            return (includeList != null)
                ? this.ObjectSet
                      .Include<T>(includeList)
                      .SingleOrDefault(expressionFilter)
                : this.ObjectSet.SingleOrDefault(expressionFilter);
        }

        public T GetFirst(Expression<Func<T, bool>> expressionFilter,
                          string[] includeList)
        {
            return (includeList != null)
                ?
this.ObjectSet.Include<T>(includeList).FirstOrDefault(expressionFilter)
                : this.ObjectSet.FirstOrDefault(expressionFilter);
        }

        public void Add(T entity)
        {
            if (entity == null)
              throw new
                  ArgumentNullException("Entity not provided for addition to
Context.");
            this.ObjectSet.AddObject(entity);
        }

        public void Delete(T entity)
        {
            if (entity == null)
              throw new
                 ArgumentNullException("Entity not provided for deletion from
Context.");
            this.ObjectSet.DeleteObject(entity);
        }

        public void Save()
        {
            this._context.SaveChanges();
        }

        private IObjectSet<T> ObjectSet
        {
            get
            {
                if (_objectSet == null)
                {
                    _objectSet = this._context.CreateObjectSet<T>();
                }
                return _objectSet;
            }
        }
    }
```
The GenericRepository class

Relevant Interface Description

`IQueryable<T> GetQuery():`

- Returns an `IQueryable<T>` object containing all the objects in the specified entity type. It is used to build customized queries against the entity.

`IList<T> GetAll(Func<IQueryable<T>, IOrderedQueryable<T>> orderBy = null):`

- This function returns a list of entities in the specified sort order. An example of how it might be used is: `GetAll(q => q.OrderBy(x => x.FirstName)).`

`IList<T> Find(Expression<Func<T, bool>> whereFilter, string[] includeList ):`

- This method executes a Where method for the lambda expression methods provided by Entity Framework. It returns a list of objects that meet the search criteria provided in the `whereFilter` parameter which is a lambda expression. If objects are not found, an empty list is returned.

`T GetSingle(Expression<Func<T, bool>> expressionFilter, string[] includeList):`

- This method is used in situations where one and only one object should satisfy the query condition. If an object is not found, null is returned. If more than one object is found the meets the query condition an exception is thrown.

`T GetFirst(Expression<Func<T, bool>> expressionFilter, string[] includeList):`

- This method returns the first element in a set of elements that meet the search criteria. If objects are not found, null is returned.

Relevant Parameter Description

Some methods contain the **`string[] includeList`** parameter which is used to identify which objects in the object graph are to be explicitly loaded with the query. (In EF default behavior is LazyLoading.)

Here are examples of the methods being used. The list of vehicle objects will contain the purchase order objects. (See Minimize Data Retrieval Trips: The Extensions Class for details pertaining to the Include method.)

```
        string[] includeList = { "tblModItem.tblPurchaseOrderMod.tblPurchaseOrder"
};
        var vehicles = _vehicleRepository.GetQuery()
                                  .Include(includeList)
                                  .OrderBy(v => v.tblSery.Series)
                                  .ThenBy(v => v.VID);
```

Examples of GenericRepository method usage

2.4.4.2.4 Extensibility: Specialized Repository Example

The listing below shows how the `GetQuery()` method from the `GenericRepository` base class can be used. In this case, `Where()` clauses are attached to the `IQueryable` object that is returned by the `GenericRepository.GetQuery()` method. The query is not executed until the `ToList()`method is called.

The second method, `GetSpecIdBySeriesId()`, uses the `GenericRepository.GetSingle()` method. This method takes two parameters. The first parameter accepts a lambda expression that is used to filter the data being requested. The second parameter accepts a lambda expression indicating how the returned objects are to be sorted.

```
   public class OrderRepository : GenericRepository<OrderModel>, IOrderRepository
   {
      public OrderRepository(IUnitOfWork UoW) : base(UoW) {}

      public IList<OrderModel> GetOrders(string order, int customerNo, int lineId)
      {
         var query = GetQuery();

         if (!String.IsNullOrEmpty(order))
            query = query.Where(x => x.Orders.OrderType == order);

         if (!String.IsNullOrEmpty(customerno))
            query = query.Where(x => x.Customers.ContractNo == customerNo);

         if (equiptype.HasValue && equiptype != 0)
            query = query.Where(x => x.Order.LineItem.LineId == lineId);

         return query.ToList();
      }

      public long GetOrderIdByLineId(long id)
      {
         var obj = GetSingle(x => x.LineItem.LineId == id, null);
```

```
            return (obj != null) ? obj.OrderId : 0;
        }
    }
```

Specialized Repository Implementation

The Benefits of Generics and Lambda Expressions

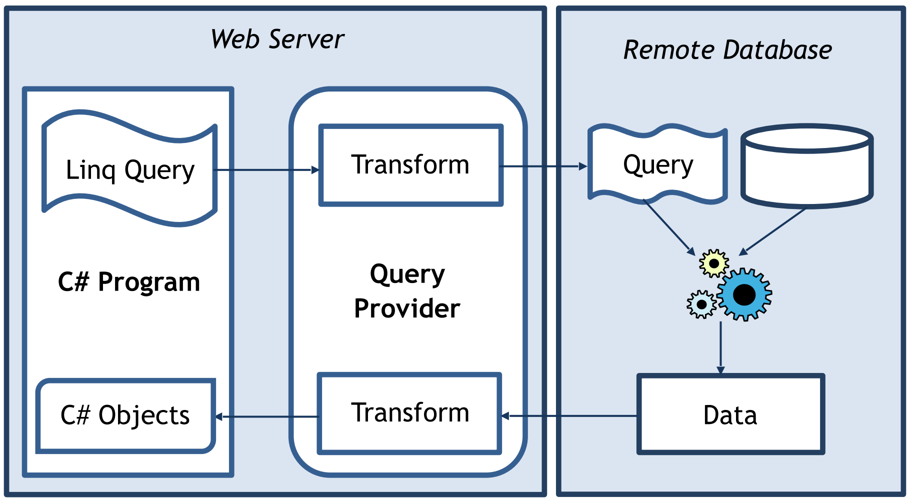
The use of generics and lambda expressions promotes reuse and significantly reduces written code. Generics allow the same code to be used against different types of Domain Model objects. Lambda expressions allow the same method to be used to select different objects sorted in different ways.

2.4.4.2.5 Performance

Why IQueryable<T>?

The method `GenericRepository.GetQuery()` returns an object of type `IQueryable<T>`. It could have returned an object of type `IEnumerable<T>` since `IQueryable<T>` is derived from `IEnumerable<T>`. The following provides the reasoning for this decision.

Using the diagram below as a guide, `IEumerable<T>` processes data on the WebServer. This means it first retrieves all the data from the remote database and then applies the conditions outlined in a lambda expression. In contrast, `IQueryable<T>` provides a lambda expression that is transformed by a query provider into the logic format of the appropriate provider. In this case, a sql query that is sent to the database. That query is then executed by the database engine and only the data that satisfies the query is returned. As a consequence, in many situations `IQueryable<T>` will process data faster than `IEnumerable<T>`.



In the diagram above a query provider translates an IQueryable object into a sql format that can be used by the database. The query is executed by the database engine. The resultset is sent back to the Query Provider where it is transformed into C# objects.

Minimize Data Retrieval Trips: The Extensions Class

The `Include()` method used in the `GenericRepository` class methods is actually an extension method. Its implementation is shown below. These methods are used to "eager load" the requested data along with specified data in its graph . They are used in situations where it is known that the related data will be needed and enhance performance since a single join query is sent to the database to be processed.

NOTE: The default behavior of EF is lazy loading. In this case, the root element is returned and graph objects are retrieved when required. For example, when a `return` or `foreach` statement is encountered. This results in multiple trips to the database.

```
   public static class Extensions
   {
       public static IQueryable<T> Include<T>(this IQueryable<T> source, string 
path)
       {
           var objectQuery = source as ObjectQuery<T>;

           if (objectQuery != null && !string.IsNullOrEmpty(path))
               return objectQuery.Include(path);

           return source;
       }

       public static IQueryable<T> Include<T>(this IQueryable<T> source, string[] 
paths)
       {
           var objectQuery = source as ObjectQuery<T>;

           if (objectQuery != null && paths.Length > 0)
           {
               return paths.Aggregate(objectQuery,
                                      (current, path) => current.Include(path));
           }
           return source;
       }
   }
```

Include() extension method implementation

`Include()` is an overloaded method in the `Extensions` class.

`IQueryable<T> Include<T>(this IQueryable<T> source, string path)`

- An extension method that takes a string containing the name of the child objects that are to be returned with the query. Notice that Linq provides an Include method. However, the Include methods cannot be chained to return different object types in the graph when using POCOs. For example, `_repository.GetQuery().Include("tblSery.Series").Include("POItem")` could not be used.

`IQueryable<T> Include<T>(this IQueryable<T> source, string[] paths)`

- This method is used when multiple object types from the graph is required. The second parameter is an array that contains the object types to be returned with the root object. The Aggregate method calls the Include method for each element in the array.

2.4.5 Data Access Layer

2.4.5.1 Issues

- How to design this application so that it can take advantage of the features offered by an object relational mapper (ORM) and be able to change the ORM implementation with minimum impact to the rest of the application.
- How to fulfill the decoupling requirement outlined in the Architecture Logical Diagram Description.
- Security: securing connection strings and putting up safe guards again hacker attacks.
- Performance.
- Exception management and logging: identify exceptions that should be handled by this layer with all others bubbling up through the application. Design a component that logs exceptions and notifies the appropriate individuals when critical errors occur.

2.4.5.2 Solution and Rationale

- The Data Access Layer will be implemented as a separate component that implements a single interface defined in the Repository Layer.
- Performance gains will be realized through the use of the IUnitOfWork Save() method. ORMs track the state of domain objects and persist the state of those objects as a single transaction when the Save() method is called. Repositories will not contain a Save() method.
- The use of a unit of work also promotes data integrity and simplifies the coding process.

Performance

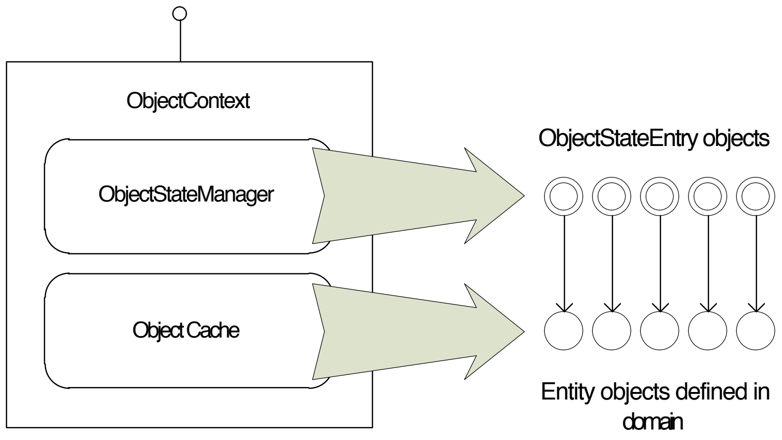
- ORMs provide the means to execute stored procedures that historically have better performance than queries generated by a data source client. However, today's ORMs perform significantly faster. In situations where performance gains are desired, tests should be run to compare query processing speeds.

2.4.5.3 DataAccess Layer Interface

Previously it was stated that using unit of work interfaces reduces the effect changing the ORM would have on the Repository Layer. Since this application will use EF, the implementation of IEFUnitOfWork is presented. In addition, the rationale for the implementation of a unit of work is provided.

2.4.5.3.1 Why the Unit of Work Pattern?

EF is designed to emulate a unit of work. EF simplifies programming and promotes the integrity of the data it handles. The diagram below serves to explain how EF works. Programmers communicate with EF through the `ObjectContext` class. This class contains methods for CRUD operations and a `SaveChanges()` method. Internally, the ObjectContext contains a cache and ObjectStateManager object. All entity objects that have gone through CRUD operations are contained in the cache. The ObjectStateManager tracks the state of entities by creating ObjectStateEntry objects for each entity. Those states can be: Added, Unchanged, Deleted or Modified.



When deemed appropriate by the programmer the objects in the cache are explicitly persisted to the data source when the `ObjectContext.SaveChanges()` method is called.

The benefit of this design is that CRUD operations can be performed without concern for the state of the database due to these operations. If a sequence of steps is to be performed during an http request and one of these steps fails, the entire operation can be negated by disregarding the unit of work. Since the SaveChanges() method was not explicitly called, programmers do not have to be concerned with what was or was not saved to the database at the point of failure.

To take advantage of this capability offered by the ObjectContext the application must implement the unit of work pattern.

2.4.5.3.2 Unit of Work Implementation

The DataAccess Layer contains a single interface that implements the IUnitOfWork interface defined in the Repository Layer. The code in Figure 4 shows the Entity Framework implementation of the interface. It contains a:

- Constructor that takes in the `ObjectContext`.
- `GetContext` property that can be used by repositories to obtain a reference to the `ObjectContext`.
- `Save()` method that causes the `ObjectContext` to persist all CRUD operations to the database.

```
       public class EFUnitOfWork : IEFUnitOfWork<ObjectContext>
       {

              private readonly ObjectContext _context = null;
              private bool disposed = false;

              public EFUnitOfWork(ObjectContext context)
              {
                     if (context == null)
                            throw new ArgumentNullException("Unable to initialize
");
                     _context = context;
              }

              public ObjectContext GetContext
              {
                     get { return _context; }
              }

              public void Save()
              {
                     _context.SaveChanges();
              }
       }
```

Entity Framework implementation of unit of work

2.4.5.3.3 Using the Unit of Work

The instantiation and use of the Unit of Work follows:

```
       IUnitOfWork UoW = new EFUnitOfWork(new ObjectEntities());
        GenericRepository<Rule> target = new GenericRepository<Rule>(UoW);
```

Unit of Work instantiation.

In the preceding code the ObjectContext is housed in the ObjectEntities object and it is passed into the constructor of the unit of work during its creation. The unit of work is then passed into the constructor of a repository.

With this design the same instance of the unit of work is passed into different repositories. This connects all the repositories to the same `ObjectContext`. Consequently, any combination of operations across different repositories can be persisted as a single transaction.

Here is how the Save method would be used:

```
UoW.Save();
```

2.4.6 Security Layer

Security means different things to different parts of the application. This section covers:

- Authentication and authorization: It must be easy to use and extensible.
- Securing access to the application's database.
- Securing the application against hacker attacks.

2.4.6.1 Securing Data Access

- Any connection string contained in the app.config file will be encrypted.
- This application uses SQL Server and will use Windows authentication to implement a trusted database subsystem.
- When using stored procedures, only parameter based queries are to be used. This ensures that a hacker does not have a means of writing a query that makes its way to the database.

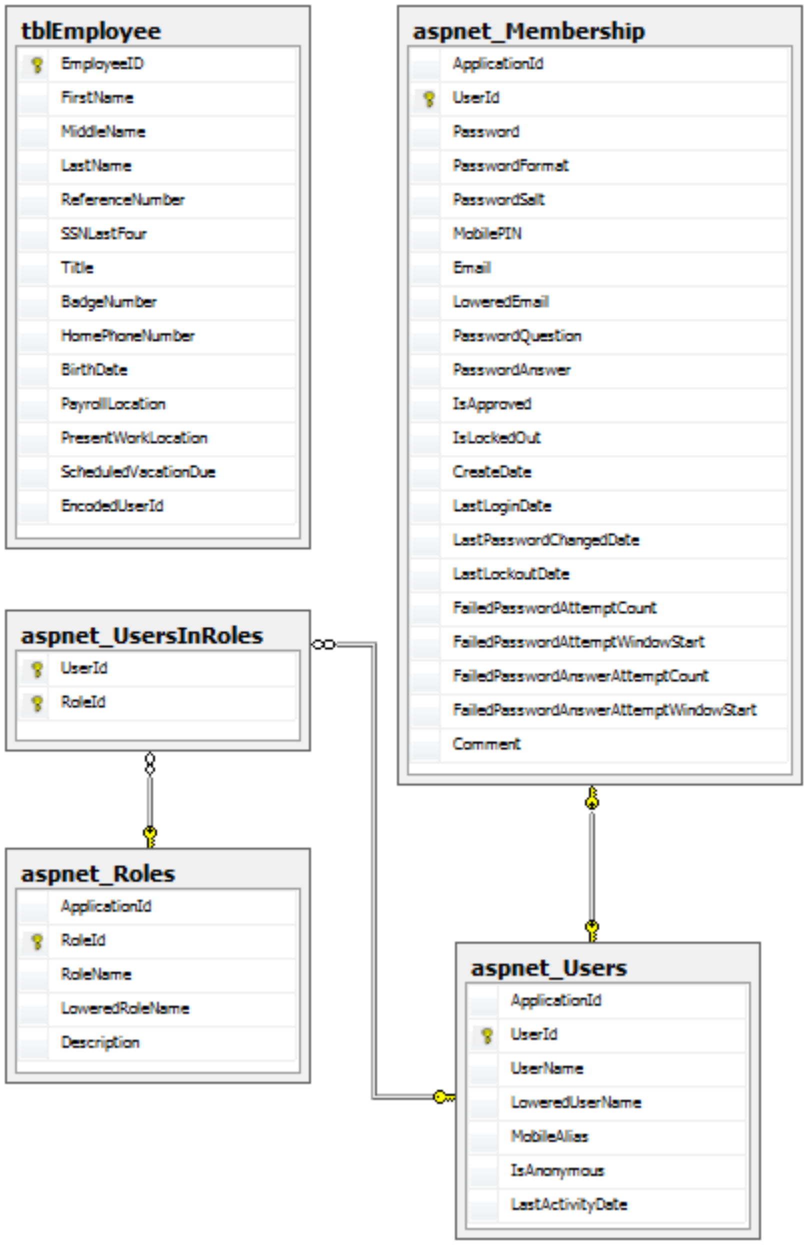
2.4.6.2 Securing Application.

2.4.6.1 Authorization and Authentication Implementation

This section describes how authentication and authorization is implemented in this application. It is a temporary solution that will be replaced by a city-wide SAML system.

NOTE: In this section security means authorization and authentication.

2.4.6.1.1 Security Data Model



The diagram shows the tables that are of importance for this document. Application Security (AS) uses the preexisting tblEmployee table to identify users. An employee provides a userId and password that is stored in the aspnet_Membership table. Note: UserId and Password are encrypted fields within the aspnet_Membership table.

Notice that there is not a link between the tblEmployee and aspnet_Membership table. The related fields are the EncodedUserId and the UserId. Both of these fields are encrypted by different methods. Consequently, although they relate to the same data, their contents will be different. The scheme is as follows:

1. A user provides a userid and password that is encrypted by FAF and authenticated.
2. FS encrypts the userid via its encryption method and looks up the employee by the EncodedUserId.

This scheme ensures that if someone was able to gain access to the tblEmployee table it could not be determined the userid and password for an employee. This applies to situation where someone gains access to the aspnet_Membership table.

2.4.6.1.2 Security Entity Model

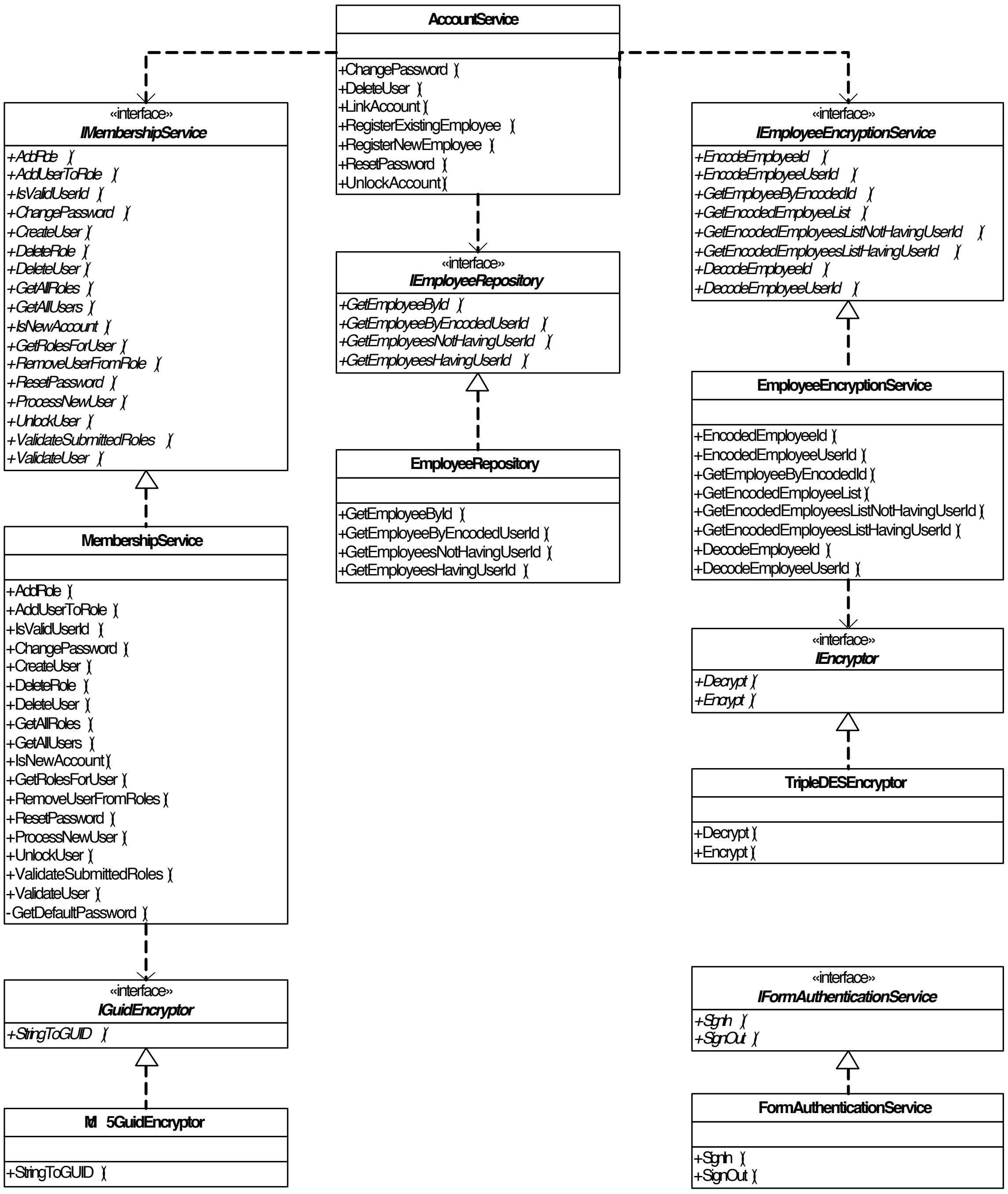
The Security Entity Model is shown below.

Application security is based on the ASP.NET Forms Authentication Framework (FAF). Two classes in AS act as implementation to interfaces to the services provided by FAF:

- `FormAuthenticationService` class that provides login and logout functionality. If a user is authenticated, an authentication cookie is created that is tracked by the FAF. The existence of the cookie means the user is authenticated. Logging out destroys the cookie. This can be accomplished in two ways. A user can be explicitly signed out by calling the `SignOut()` method or implicitly signed out when all browser windows are closed. This class utilizes the methods contained in the FAF class `System.Web.Security.FormsAuthentication`.
- `MembershipService` class that provides all the services related to managing user accounts and roles. This class utilizes the methods in the FAF classes: `Roles` and `MembershipProvider`.

Within the domain model, security functions are housed in the `AccountManager` class. This service provides all the functionality needed by the various views used to manage user accounts.

Two methods in `AccountManager` provide examples of using this class and warrant explanation. Forms Authentication is setup against an existing Employee table. In situations where an employee is in the employee table the `RegisterExistingEmployee()` method is used to setup a user account. In situations where an employee is not in the employee table `RegisterNewEmployee()` method is used.


AccountService
+ChangePassword()
+DeleteUser()
+LinkAccount()
+RegisterExistingEmployee()
+RegisterNewEmployee()
+ResetPassword()
+UnlockAccount()
«interface» IMembershipService
+AddRole() +AddUserToRole() +IsValidUserId() +ChangePassword() +CreateUser() +DeleteRole() +DeleteUser() +GetAllRoles() +GetAllUsers() +IsNewAccount() +GetRolesForUser() +RemoveUserFromRole() +ResetPassword() +ProcessNewUser() +UnlockUser() +ValidateSubmittedRoles() +ValidateUser()
MembershipService
+AddRole() +AddUserToRole() +IsValidUserId() +ChangePassword() +CreateUser() +DeleteRole() +DeleteUser() +GetAllRoles() +GetAllUsers() +IsNewAccount() +GetRolesForUser() +RemoveUserFromRoles() +ResetPassword() +ProcessNewUser() +UnlockUser() +ValidateSubmittedRoles() +ValidateUser() -GetDefaultPassword()
«interface» IGuidEncryptor
+StringToGUID()
Md5GuidEncryptor
+StringToGUID()
«interface» IEmployeeRepository
+GetEmployeeById() +GetEmployeeByEncodedUserId() +GetEmployeesNotHavingUserId() +GetEmployeesHavingUserId()
EmployeeRepository
+GetEmployeeById() +GetEmployeeByEncodedUserId() +GetEmployeesNotHavingUserId() +GetEmployeesHavingUserId()
«interface» IEmployeeEncryptionService
+EncodeEmployeeId() +EncodeEmployeeUserId() +GetEmployeeByEncodedId() +GetEncodedEmployeeList() +GetEncodedEmployeesListNotHavingUserId() +GetEncodedEmployeesListHavingUserId() +DecodeEmployeeId() +DecodeEmployeeUserId()
EmployeeEncryptionService
+EncodedEmployeeId() +EncodedEmployeeUserId() +GetEmployeeByEncodedId() +GetEncodedEmployeeList() +GetEncodedEmployeesListNotHavingUserId() +GetEncodedEmployeesListHavingUserId() +DecodeEmployeeId() +DecodeEmployeeUserId()
«interface» IEncryptor
+Decrypt() +Encrypt()
TripleDESEncryptor
+Decrypt() +Encrypt()
«interface» IFormAuthenticationService
+SignIn() +SignOut()
FormAuthenticationService
+SignIn() +SignOut()

Security Object Model

2.4.6.1.3 How Authorization and Authentication is Enforced

Security is enforced by using attributes attached to a controller class or a controller action method. More precisely:

- Authentication attributes are attached to controller classes. This ensures that an action method will not be executed unless the user is authenticated.

- Authorization is implemented by assigning *Roles* to each user. A user can be assigned multiple roles. In addition to roles, attributes assigned to controller action methods indicate which roles are permitted to execute the method.

For information on how to use and extend this framework refer to the document "*Application Security: The Authentication and Authorization Framework*", as noted in this documents References section.

2.4.7 Operational Management

2.4.7.1 Exception Management and Logging

Exceptions will be handled by ELMAH (Error Logging Modules and Handlers), an open source library that is widely accepted by the developer community due to its ease of use, ability to log information to various data sources and its ability to send exception notification by different means.

This library provides the following features that are of importance to this application:

- Logging of unhandled exceptions.
- Provides webpage to remotely view log of recorded exceptions.
- In some cases, view the original yellow screen.
- E-mail notification of each exception at the time it occurs.
- Ability to send error messages/tweets to iPhone, iPad or a custom application.

In this application:

- All unhandled exceptions will be logged.

- Concurrency conflict will be propagated to the user for resolution.

- ORM exceptions will be searched to determine if the actual SQL sent to the database can be obtained. **Being researched.**

2.5 Selection Criteria for Chosen Architecture

The criteria used to arrive at this architecture are as follows:

- Facilitate the isolated design and development of each layer.

- Promote reuse of code.

- Simplify the maintenance of the code base.

- Reduce the amount of time required to implement future enhancements.

- Ensure the integrity of the data processed by the application.

- Ensure a satisfactory response time by the application.

- Ensure the security of the application.
- Ensure exception handling and logging.
- Promote the availability of the application.

The MVC framework is chosen over WebForms because it better suits the needs of modern web applications. Whereas early web applications essentially returned static documents, present day browser requests actually want a set of tasks performed with every request. The result of the execution of these tasks determines the View that will be returned to the client. The MVC pattern is best suited for this scenario.

The PL and AL comprise the MVC framework. Normally these layers are not separated. However, it cannot be assumed that the platform for presenting the UI will always be the same. It could be a laptop, tablet or cell phone which uses a browser, SilverLight, Flash or native presentation components. The architecture aims to explicitly convey this message by having the UI in its own layer.

The AL consists of the ASP.NET MVC Controllers and the View Model. The View Model serves as a means to decouple the PL from the AL. With such a setup, work on the interface can progress independently of the AL. In addition, View Models enable UI independent testing of the content being provided to the UI. They also enable automated tests to be run on UI content.

The DL is designed to be independent of any other layer. DDD is used because such a design leads to reduction in code duplication and less time required to maintaining/enhance the application. In addition, domain objects can be unit tested.

Over time the Domain Layer will experience the most change. This layers lack of dependency on other layers significantly benefits the maintenance of the application. Enhancements to business rules may change the way objects interact with each other and yet have no effect on the types of data presented to a user or obtained from the database. Put another way, a change in a business procedure does not mean there will be a change in other layers of the application.

The decoupling of layers via interfaces enables the implementation of layers to change with minimum impact on other layers.

Libraries and layers are designed with the intent of being used by other applications.

2.6 Technical Risks & Mitigation Strategy

ID	Risk Description & Impact	Mitigation Strategy and/or Contingency Plan
1	Domain Driven design is often considered as confusing. Developers seem to have a hard time grasping it.	DDD must be understood thoroughly before applying. A Domain model should not be mixed with technical side of project. It should be based on the analysis of the physical domain.
2	Entity framework can lead to performance issues in application.	Entity framework is new and is being improved with every new release.
3	Certain features in this architecture have been developed and tested; others are in the prototype stage.	This document mitigates many of the problems because it gives a birds view of the applications parts and thus can be used to figure out how to integrate the components. It serves as a guide for the logical development of components.
4	The significant task in developing this application will be the implementation of the vehicle transition engine. If designed correctly, it will significantly reduce long term maintenance.	Detailed documentation of the business rules has been written. With a thorough understanding of the rules a clean elegant solution can be devised.
5	Telerik UI components are not mature and have gone through changes.	Where possible features offered by HTML5 will be utilized.

3. Enterprise Architecture Models

3.1 As Found State

4. Architectural Goals and Constraints

- Security – The application is designed to use ASP.NET Forms Authentication. At a later date SAML will replace this layer. At the time the Security layer was written a final interface to SAML was not available. Consequently, to prepare for the use of SAML the footprint of the present security implementation is minimal. It is constrained to class and method attributes contained in the application's controllers.
- The ability to change the status of a vehicle is the central piece to this application. This functionality must be implemented such that vehicle status transition rules can be changed without the need for coding. Coming up with a way of implementing this feature will be crucial to the long term success of this application.

4.1 Technology

- Visual Studio 2010
- SQL Server 2008
- SQL Server Reporting Services
- .NET Framework 4
- ASP.NET MVC 3
- Entity Framework 4.1
- C#
- StructureMap
- AutoMapper
- jQuery/JavaScript
- Telerik extensions for ASP.NET MVC (Now a part of Kendo UI)
- HTML5/CSS3
- ELMAH
- IIS 7

4.2 Production Platform

- .NET Framework 4
- ASP.NET MVC 3
- IIS 7
- SQL Server 2008 R2

4.3 Training

- .Net framework 4
- ASP.NET MVC 3
- IIS 7
- SQL Server 2008 R2

4.4 QA Platform

- .Net framework 4
- MVC framework 3
- IIS 7
- SQL Server 2008 R2

4.5 Development Platform

- Visual Studio 2010
- SQL Server 2008
- SQL Server Reporting Services
- .NET Framework 4
- ASP.NET MVC 3
- Entity Framework 4.1
- C#
- StructureMap
- AutoMapper
- jQuery/JavaScript
- HTML5/CSS3
- ELMAH
- Telerik extensions for ASP.NET MVC (Now a part of Kendo UI)

5. Non Functional Requirements

5.1 Reliability

Reliability ensures the integrity and consistency of the application and all its transactions.

Reliability as applied to this application relates to the following issues:

- Ensure that database transactions are saved correctly. (See DataAccess Layer Interface for how this is addressed.)
- Ensuring the data entered by users is valid. (See Entity Validation Framework for how this is addressed.)
- Managing database concurrency conflicts.

This section addresses the last issue, concurrency conflict resolution when using the Entity Framework.

5.1.1 Concurrency Conflict Resolution

Concurrency issues:

- A number of data access technologies do not support pessimistic concurrency with EF being one of them. Therefore, optimistic concurrency techniques must be used.
- Consequently, in conjunction with the optimistic concurrency features provided by EF, code must be written to handle potential update conflicts.
- Any concurrency conflict solution must take the business rules of the application into consideration.

5.2 Availability

Availability ensures that a service/resource is always accessible.

Issues:

- Setting up an environment of redundant components and failover.
- Documented failover strategy for database server and webserver.
- Discuss how downtime is managed for each tier.
- Coordination of changing configuration setting when required, if necessary.

All components should be available 24/7.

5.3 Scalability

This application is a three tier application. Any gains due to scalability will be realized through the use of additional memory on the webserver. Memory, in addition to workload, processor cycles, and bandwidth on the server, will be addressed by Operations.

5.4 Performance

In order to improve performance following techniques will be implemented.

- Apply techniques used by high performance websites.
- Utilize MVC3 feature that enables views to be pre-compile.
- Caching of data used in dropdown list boxes.
- Application will use database connection pooling.
- Hardware tuning will be addressed by Operations Department
- Long running code will be performed asynchronously.
- Database will be tuned by identifying potential indexes, use of SQL profiler's index tuning wizard, query plans will be analyzed using query analyzer to optimize queries.
- Stored procedures will be used for fetching Report data.

5.4.1 Caching Strategy

A design goal of this application is to provide the most current information to a user. Consequently, the caching of data will be administered on an "as needed" basis and will not be applied unless all other options do not provide a satisfactory response time.

5.5 Extensibility & Maintainability

The application uses the following design principle and patterns in the development of this application:

- Code to an interface not an implementation: Classes should be designed to meet the contract of an interface. These interfaces are then used as the types of method parameters.

- Loose Coupling/Dependency Injection (DI): Using interfaces as parameter types enables different implementations of the interface to used by a class without needing to modify that class. It also facilitates isolated unit testing since a mock implementation can can be provided to the class being tested. When interfaces are used as parameter types and dependencies are injected into a class, a DI Container can be used to "wire up" the application from a centralized location known as the Composite Root (CR). The CR has the effect of having any change in the implementation of an interface to propogate through the entire application.

- Open-Close Principle: A class should be designed so that it is open for extension and closed for modification. Portions of code that is prone to change should be placed in separate classes that meet the contract of an interface. These objects are then injected (DI) into the objects that use them.

- Single Responsibility Principle: Classes that have many responsibilities should be reviewed to determine which responsibilities are prone to change. Those responsibilities should be removed from the main class and placed in their own class that meet the contract of an interface.

- Aggregate Roots. (See Aggregate Roots for details.)

5.6 Persistence

This application will use Entity Framework and SQL Server database to implement its data persistence layer.

5.7 Manageability

To promote an organized means for the long term enhancement of this application certain features of the *Teams Foundation Server* will be used. In addition, certain developer best practices related to *Teams* will be identified, documented and implemented. These practices will enhance the integrity of the code base and reduce confusion between developers working on this application.

6. Non-Functional Check List

7. UML Diagrams

7.1 Class Diagram

7.2 Entity Relationship Diagram

7.3 Sequence Diagram

8. Implementation View

8.1 Design Patterns

Pattern Name	**Where Used**
Repository	For data access.
Unit of Work	Persisting of data and providing repositories with a single context.
Dependency Injection (DI)	Managed by StructureMap Container and implemented throughout the application.
MVC	Used in the Presentation and Application Layers
Factory	Used to create valid domain objects.
Specification	Used to determine if certain defined sets of attributes in a vehicle meet the requirements to change the status of the vehicle. It is also used to determine is vehicle status change requirements are met.

9. Deployment View

The application will consist of dynamic linked libraries that will be published to the Operations Web Server. It is depicted below as the *Solution Package* that will contain Presentation, Domain, Repository and DataAccess components (dlls) along with other application support dlls.

9.1 Environment Details

9.1.1 Server Side

ID	Hardware/software	Details (e.g. Ports, Setup, Certificated)
1	External firewall	Operations firewall
2	Internal firewall	None
3	Load balancer	None
4	Web Server	IIS 7 (Dev: 100.10.10.11, QA: 100.10.10.10, PROD: 100.10.10.12)
5	Application Server	None
6	Portal Server	None
7	Database Server	Production: SQLProduction Development: SQLDevelopment

9.1.2 Client Side

ID	Hardware/Software	Details
1	Browser	Internet Explorer

9.2 Production Roll out Strategy