Presentation Layer Class Diagram:

Model View Presenter



The above class diagram is an overview of the basic structure of the CheckPoint presentation layer. This layer has been designed according to the Model View Presenter (MVP) architectural pattern. The primary goal of the MVP pattern is loose coupling between the user interface, the presentation logic, and the business logic. This is achieved by using methods of abstraction across a three-layer structure that resides mostly within the presentation layer of the application.

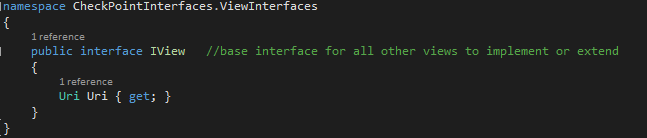
* The View layer contains classes that concern the user interface (UI).
* The Presenter layer contains the classes that act as a mediator between the View and the Model layers.
* The Model layer contains the classes that hold the business logic along with the data that is to be displayed in the View.

Loose coupling between the layers, especially the View and the Model, leads to more flexible, maintainable code and a more easily testable application. The use of interfaces is the foundation of the pattern providing a way for the Presenter layer to communicate with the other layers through abstraction instead of using concrete implementations of the classes and methods it needs to perform its tasks. The View remains completely unaware of the Model and the business logic. The View only knows about an abstract version of the Presenter. Similarly, the Presenter is only aware of an abstract version of the Model and the Model is completely unaware of both the Presenter and the View. This separation of concerns is of great benefit when changes are made to the code. When a part of the code in one layer is changed, there will be minimum impact in the other layers. This reduces the amount of changes that will need to be made in the rest of the code. If a program is very simple, this might not make such a big difference. However, for any application that has some degree of complexity, failing to adequately separate concerns and handle dependencies can quickly result in code that is very difficult or sometimes even impossible to maintain.

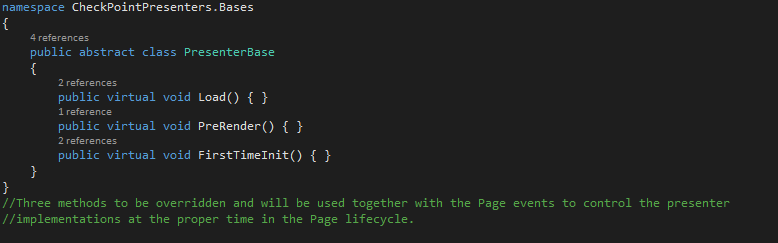
There are several variants of the MVP pattern, the two most popular being known as “Supervisory Presenter” and “Passive View”. CheckPoint is structured around the Passive View type whereby the View is reduced to being as inactive as possible with all display related logic moved into the Presenter class. The Presenter is responsible for controlling what the View should display and how it should be displayed. The View is prohibited from direct communication with the Presenter. Any interaction is performed indirectly using events published by the View. To ensure maximum flexibility it is important to make sure that the Presenter only sends primitive data types to the View rather than platform specific types.

The following code snippets will attempt to explain the class relationships of the CheckPoint application found in the presentation layer. The focus will be how interfaces have been used to achieve the desired de-coupling.

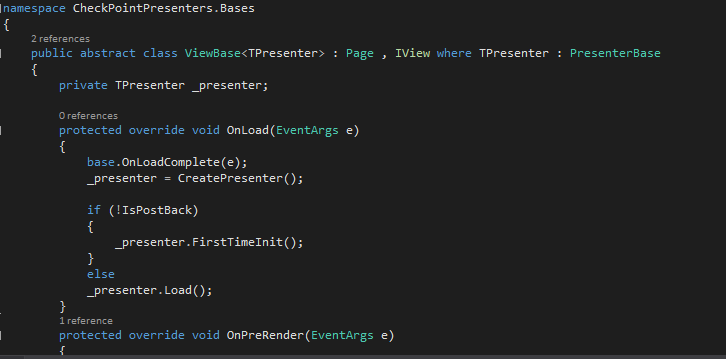
We begin with a very basic interface that all of our views will implement and extend. Any properties or methods in this interface will be available globally to all Views of the application. In the code snippet below a single property is described which simply returns the current HTTP context URL of the ASP.NET Webform.



In a similar way, we define an abstract base class for all of our Presenters to inherit. Inside the abstract class, we define three virtual methods that will be overridden by the concrete Presenter classes. These three methods are used to control the creation and implementation of a Presenter at the appropriate time during a Webform lifecycle. This is necessary due to the way that ASP.NET controls the loading of WebForm pages. These base classes assist us in reusing code and avoiding unnecessary duplication of common methods and attributes.

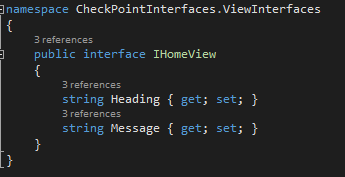


Next, we proceed to define an abstract base class for all of our views to inherit. Firstly, this base class implements our basic View interface. Secondly, this base class requires and accepts any generic Presenter class so long as that Presenter inherits form the base Presenter class that we defined. This base view class itself inherits from the “Page” class of an ASP.NET WebForm and in doing so lets us hook in to the important lifecycle events that take place when a Page is loaded. We override these events and use them to trigger a call to our base Presenters virtual methods. Whenever we redirect to a View and it fires the “OnLoad ()” event, a Presenter will be created and the appropriate method will be called on the Presenter according to whether it is the first time that the View is loaded or a Postback. The specific details of the “CreatePresenter()” method we see below will be explained in a later chapter “SturctureMap Inversion of Control” and not shown here for the sake of simplicity. The important point is that we are able to call a method that can instantiate a Presenter at the correct time during a Page load.

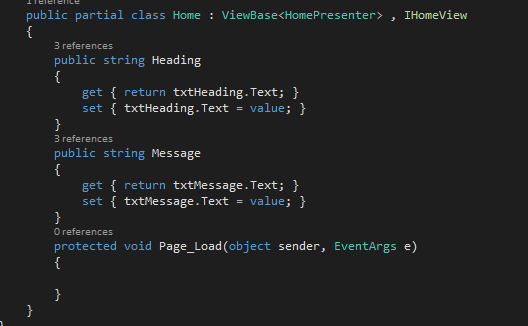


Now the basic infrastructure is in place that we can begin to create our more specific View and Presenter classes.

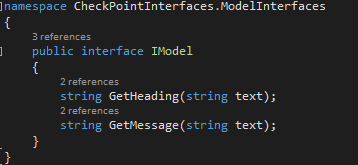
An interface for a homepage is defined with two get and set string properties. This is kept as simple as possible to provide an example of how data will be passed to the display.

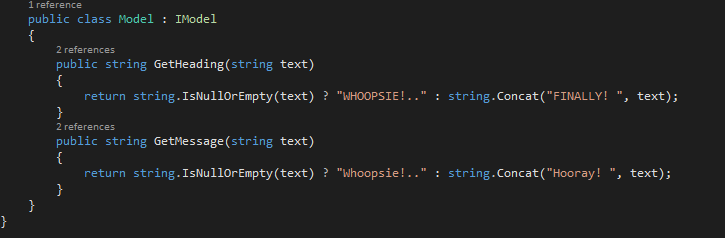


Next a concrete home View is defined. This View inherits from both the abstract base View and the home View interface. The abstract base View automatically endows the home View with all of the Page class attributes as well as the basic View interface attributes. The abstract class also obligates the home View to require a Presenter type and therefore exposes the important methods needed for a View to create its associated Presenter when it loads. The properties of the home View interface are now defined and assigned to two textboxes. These properties will be exposed to and controlled by the Presenter.

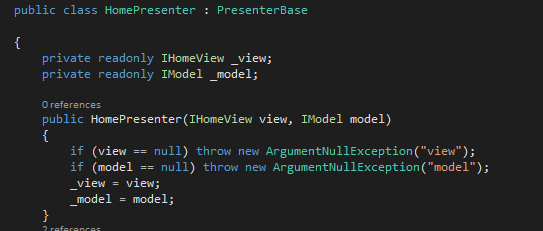
.

A Model interface and its respective class is defined in the same way as the View. Once more in the interest of simplicity, the Model has just two properties that reflect the data that the View will display. In reality, the Model will contain the objects and machinery that collects the required data from a persistent source and processes it before passing it to the Presenter. Here again the interface defines the function and the concrete class provides the detail of how the function will be accomplished.

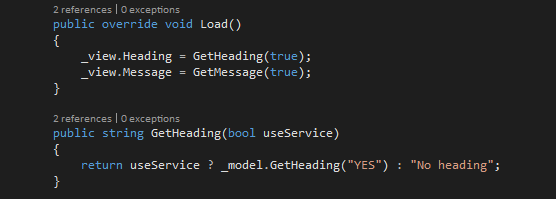




Finally, a home page Presenter class is defined with a constructor that accepts a View interface and a Model interface as arguments. This allows us to compose a Presenter that can access all the methods and properties of the View and Model without needing to instantiate the concrete View or Model. The Presenter does not depend on the details of the objects it will use. It depends only on the abstractions.



The presentation layer pattern is complete and the Presenter can now set the View properties with data from the Model when the View loads.



CheckPoint Business Layer & Data Access Layer Class Diagram:

Repository Pattern

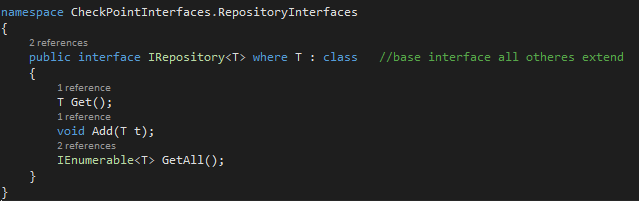


The class diagram above represents an overview of the data access layer along with some specific classes from the business logic layer. Similar to the way that the CheckPoint presentation layer has been organized, the data access layer employs two patterns in order to achieve loose coupling between the core business logic and the data access layer. The patterns used are known as the “Repository” pattern and the “Unit of Work” pattern. These two patterns work in combination to separate the business layer from the data access layer and invert the direction of dependencies such that the data access layer depends on the business layer and not vice versa.

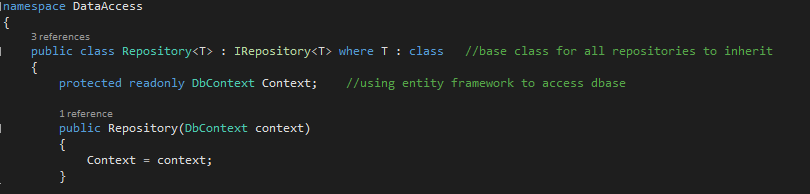
Often applications are written in a way that ties them to a specific vendor’s database, storage system or third party object relational mapping tools. Even if later it is desirable to change to another provider this can prove to be too difficult to do without rewriting major parts of the source code. The goal of these patterns is to make it possible to change the persistence layer if desired without a major rewrite. Once more, this is achieved by abstracting the functionality out to interfaces.

The following code snippets will attempt to show how the Repository pattern and Unit of Work pattern are implemented to communicate with a database.

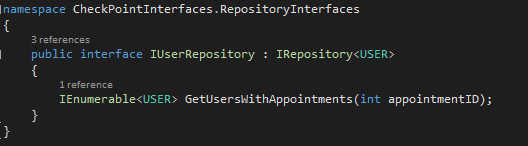
We begin with a completely generic interface that provides the most general queries common to all table queries. Every Repository class will inherit from this interface allowing us to avoid duplicating these basic methods throughout the code.



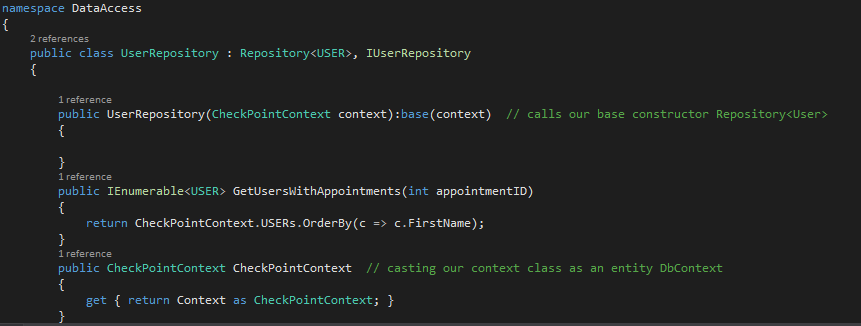
Next, a generic Repository class is defined that inherits form the basic interface and accepts a “DbContext” object in its constructor. The Entity framework DbContext object provides a way to access a database with a predefined connection string and make queries using convenient “LINQ” expressions. All Repository classes can extend this class through inheritance to provide specific functions for each table as desired.



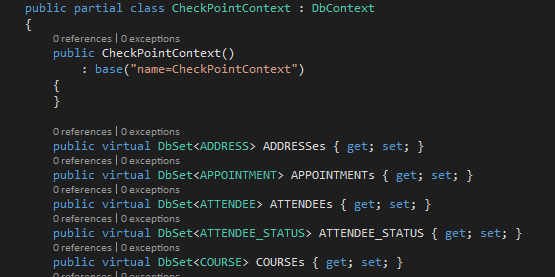
An interface is now defined for a specific type of table that will return “User” type object data from its queries. The “User” class is a model of the user table that exists in the database. This interface provides access to the Repository functions and methods through abstraction rather than needing a specific implementation to call upon or create inside a client class.



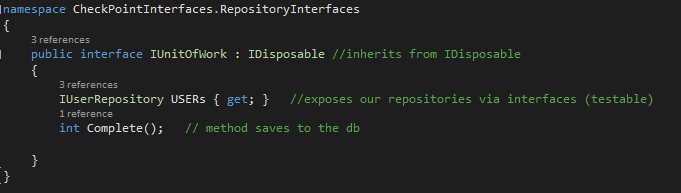
A concrete specific “User” Repository is defined that describes the specific details of the functions that the Repository interface provides. This is how the methods of the interface will actually be carried out yet the client class that calls the methods will not need to have any knowledge of this class itself – it will merely reference the “User” Repository interface instead. Using composition, we define the class constructor to accept a Context object.



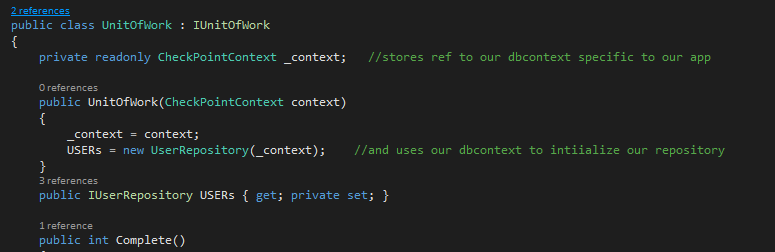
The “CheckPointContext” object is a class we define ourselves so that we have the ability to swap Entity framework with another if that became preferable. CheckPointContext inherits form the Entity Framework DbContext class but it could easily be modified to inherit from another context object from another provider. The class properties are the object type representation of our database tables that we use to manipulate the data without directly accessing the database tables. Should we use another OR/M framework this, and the basic Repository class, is where the modifications would need to be made rather than throughout the source code. This isolates the framework dependency to the data access layer only.



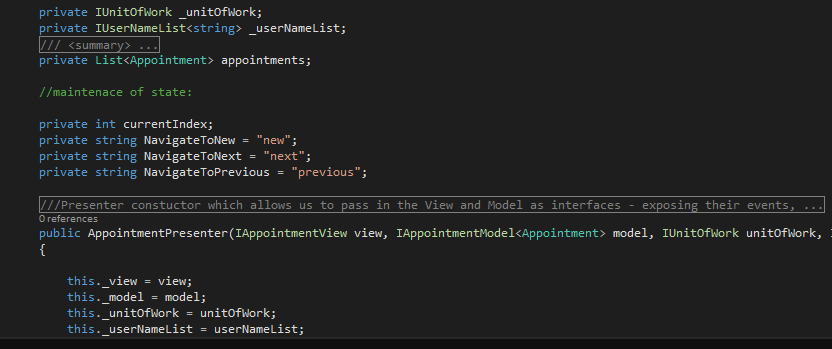
Now two things are necessary to complete the pattern. A way to ensure that we do not break any rules of concurrency when saving changes to the database and a way to access the functionality provided by all of the different types of Repositories that we will create in one centralized place. For this, we will create a “Unit of Work” interface where we will add each type of specific Repository interface that we want to use as properties of the Unit of Work class. A complete method will need to be called at the end of any type of query that will alter the data in the database. Since the Unit of Work will use a single DbContext object to carry out its work we are sure that any changes are performed as a single simple transaction. The Unit of Work keeps track of data that we manipulate as objects in the application and then handles the database update as a single transaction at the end during which the database will lock so that two users cannot make changes at the same time.



A concrete Unit of Work class provides the details for the implementation and holds properties that allow access to any Repository methods. The constructor accepts our own Context object that inherits from DbContext. We are now able to call any Repositories and their methods through the Unit of Work interface rather than needing this concrete class to be instantiated. This de-couples the client from its service and lets us maintain separation between the layers of our architecture.



It is now possible to inject our Unit of Work interface via the constructor to our Presenter and gain access to any Repository queries we wish to use using composition.





If the event that the OR/M or the type of database were swapped out, we would need to make a new concrete Repository classes to reflect the new Context objects or query syntax but we could avoid changing the code that exists in the core Business Logic layer of our application.

SturctureMap Inversion of Control: Inversion of Control Container



The above class diagram represents the Inversion of Control mechanism used by CheckPoint to perform the necessary dependency injections.

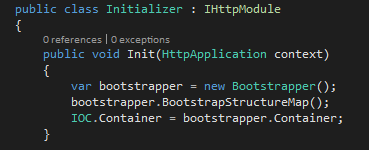
A vital part of the Checkpoint application architecture is the use of dependency injection. It is one of the main features of the Repository pattern, Model View Presenter pattern, and Unit of Work pattern. Indeed, anywhere where loose coupling is the goal it is highly likely that dependency injection and inversion of control will be employed to make it happen.

Dependency injection is nothing more than a technique for providing a class with its instance variables. If one class, the client, depends upon another class, the service, to accomplish some objective we say that the client depends on the service and the two classes are tightly coupled. If we seek to de-couple these classes then we use inversion of control to abstract the service’s functionality into an interface that the service will implement. This frees the client class from depending on the specific class and the details of how the function is performed. Instead, the client depends only on the interface abstraction. Using dependency injection, we can provide the client with any service which implements that interface. This is normally done by declaring the interface in the client class constructor but can be set as a property or a method too.

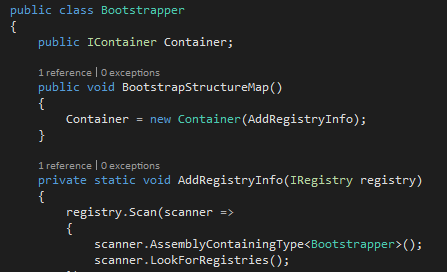
The injection of these interfaces is typically performed when the application first executes or during run-time by code outside of the main application. This is easily achieved in a Winforms solution by simply creating a new project from which we create all the objects and their dependencies and pass them into the main project when the project loads. Due to the nature of the ASP.NET Webforms Page Lifecycle, this is not so easy to do. A Webforms Page is renewed every time an event is fired or a redirection occurs. The developer has little control over this process. What is needed is a way to hook in to the Page loading events and a way to create the dependencies at the right time when a Page loads. This is where an Inversion of Control container like Structure Map is useful. StructureMap is a framework that can automatically perform the dependency injection. We begin by adding an entry to the Web.config file as shown below.



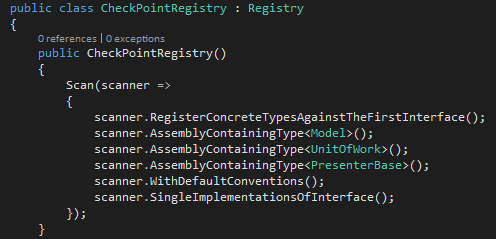
This command points to the directory of the class called “Bootstrap.Initializer” which is located in the CheckPoint.Bootstrap folder.



The Initializer class inherits from the IHttpModule. This causes the Initializer class to run “Init()” on application start up. A new Bootstrapper object is instantiated and the method “BootstrapStructureMap” is called.

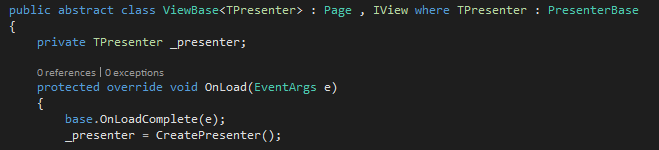


This method creates a new container that will hold a registry of all the classes and objects that we need to satisfy our dependencies. StructureMap will scan through our projects designated directories looking for any classes that inherit from the StructureMap “Registry” class.

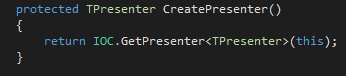


In this case, it will discover the CheckPointRegistry class and begin scanning according to the criteria found inside the “Scan” method. This means that the container will search any directory in the solution where the “Model”, “UnitOfWork”, and “PresenterBase” classes reside and register the interfaces that are implemented by those classes. StructureMap will now know which implementations to provide for a class that is dependent on the interfaces held in the container.

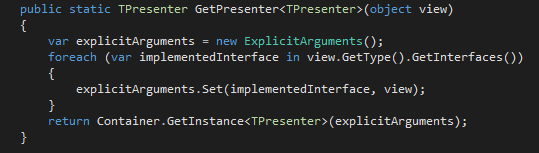
When our View loads it calls a method to “CreatePresenter()”.



This method in turn calls the “GetPresenter<TPresenter>(this)” method in the IOC class and the view inserts itself as an argument to the method.



Finally, when the “GetPresenter<TPresenter>(object view)” method inside the IOC class is called it returns a Presenter object with all of its dependencies satisfied, one of them being the View that initiated the method call in the first place.



Each time a View is loaded this procedure will run to ensure that its appropriate Presenter will be supplied and that the Presenter will have been “injected” with all of the objects it depends on.