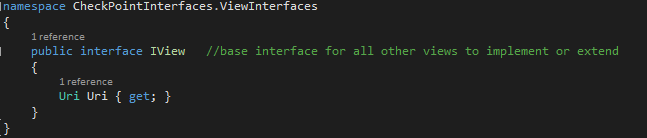
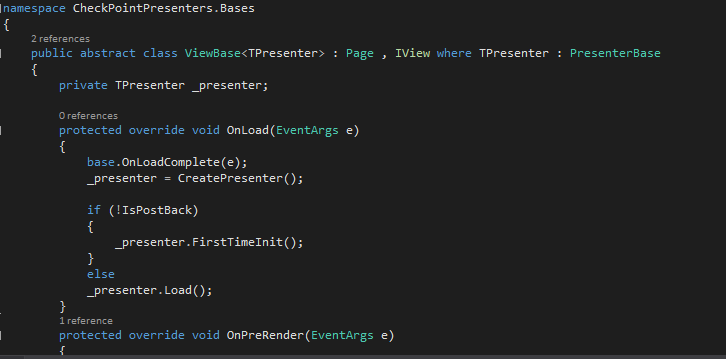
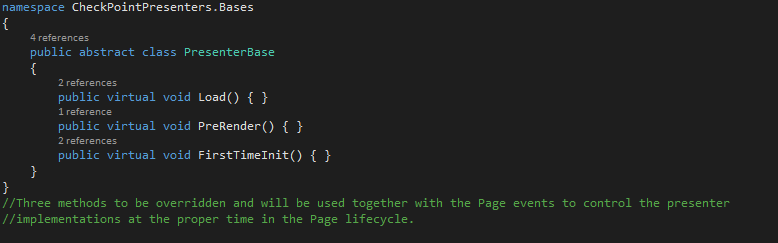
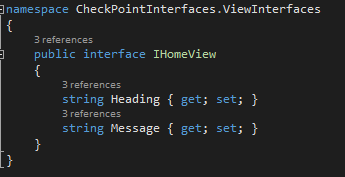
Model View Presenter

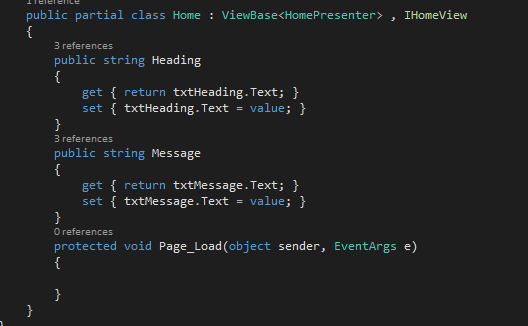


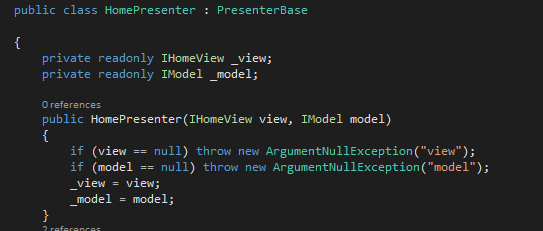


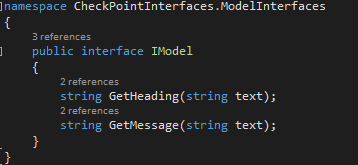


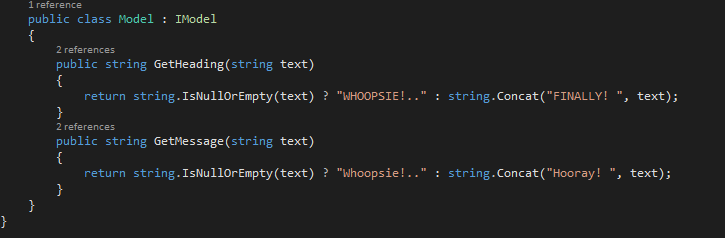






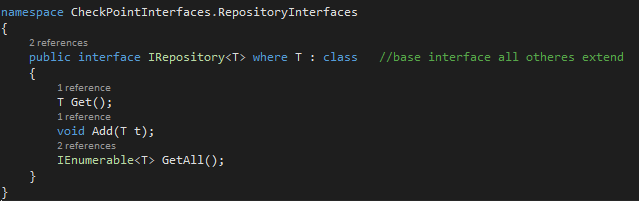




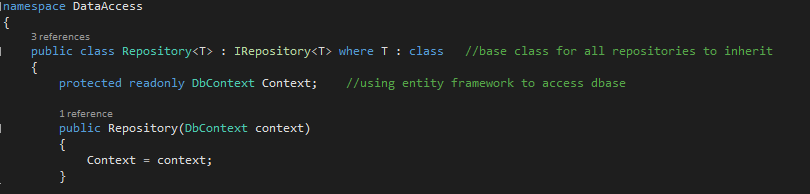


Repository Pattern

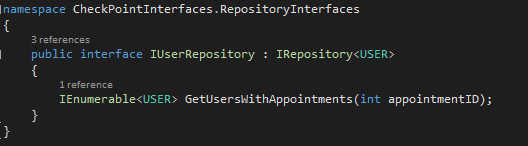




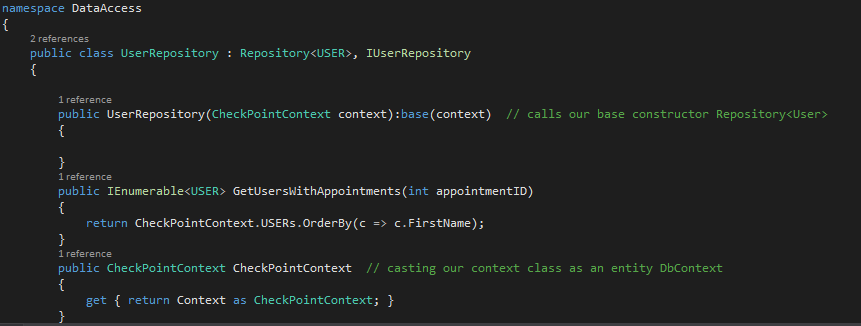
A completely generic interface that provides the most general queries common to all table queries. Every Repository class will inherit from this interface allowing us to avoid duplicating these basic methods throughout the code.



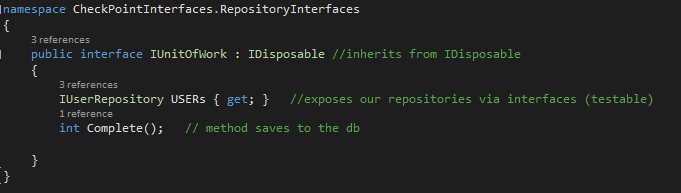
A generic class that inherits form the basic interface and accepts a “DbContext” object in the constructor providing a way to access the database through a connection string and make queries using more convenient “LINQ” expressions. All Repository classes can extend this class through inheritance to provide specific functions for each table as desired.



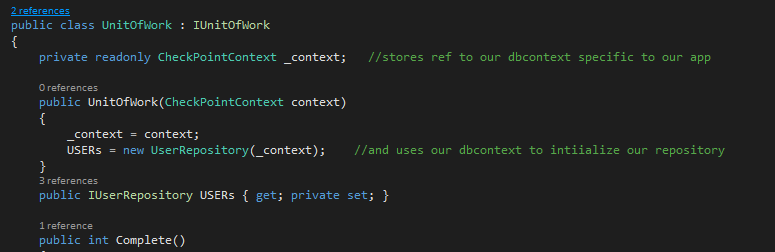
An interface is now defined for a specific type of table that will return “User” object data from its queries. This provides access to the repository functions and methods through abstraction rather than needing a specific implementation to call upon or create inside a client class.



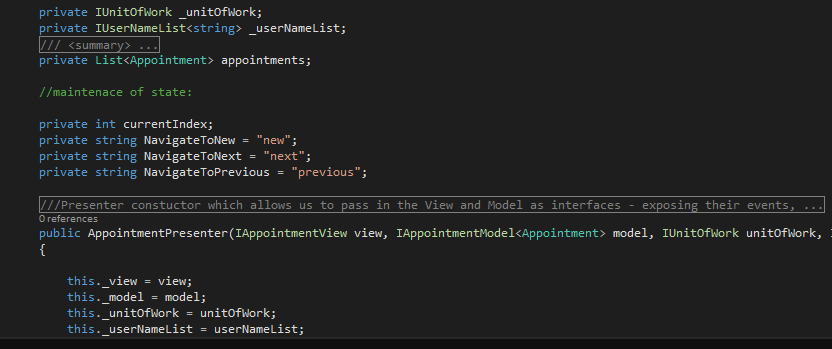
A concrete specific “User” Repository that describes the details of the functions that the Repository interface provides. This is how the methods of the interface will actually be carried out yet the client class that calls the methods will not need to have any knowledge of this class itself – it will merely reference the “User” Repository interface instead.



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 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 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 that provides the details for the implementation holds properties that allow access to the 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.



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.