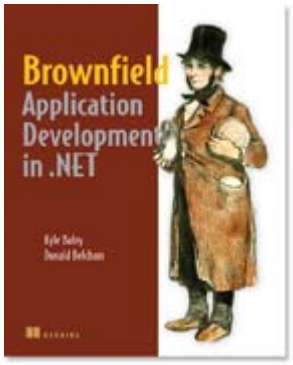
MANNING PUBLICATIONS

Taking a Domain-centric Approach

Excerpted from



Brownfield Application Development in .NET

EARLY ACCESS EDITION

Kyle Baley and Donald Belcham
MEAP Release: March 2008
Softbound print: June 2009 (est.) | 550 pages
ISBN: 1933988711

This article is taken from the book Brownfield Application Development in .NET from Manning Publications. As part of a chapter on implementing a strong layering scheme in an existing Brownfield application, this segment discusses taking a domain-centric layering approach when refactoring your code. For the table of contents, the Author Forum, and other resources, go to http://manning.com/baley/.

In many Brownfield applications, the layering is sporadic and often non-existent. Some screens may use a business logic service and others may work directly against a database. A business logic class may be a hodgepodge of methods placed there for no other reason than the developer didn't know where else to put it.

As we prepare to refactor this code, it is a good idea to think about the overall approach we would like to take in achieving layered bliss. That is, what is our overall goal in designing the seams and layers? How do we decide where to place functionality in our layers?

We will be taking a domain-centric approach to our refactoring. That is, we will refactor with an eye toward the actual domain that the application was designed to model. If the application is a family tree generator, we would expect to have concepts like Ancestor and Descendent and Relationship modeled in it.

This is all very abstract so a more concrete (though still fairly high-level) example will help illustrate.

Consider a traditional n-tier web application for a site that manages timesheets for consultants. The user navigates to a page that lists his or her timesheets to date. The code-behind instantiates a `TimesheetService` object and calls `_timeSheetService.GetTimesheets( consultantId )`, which retrieves the consultant information along with his or her timesheets.

The `TimesheetService`, in turn, may look like Listing 1.

Listing 1: Sample TimesheetService Object

```
public class TimesheetService
{
     public ConsultantData GetTimesheets( int consultantId )
     {
          var consultantData = new ConsultantData( );
          var consultantService = new ConsultantDataService( );
          var consultantDs = consultantService
              .GetConsultantDataSet( consultantId );
          var consultant = new Consultant( );
```

```
            // Insert code to translate the consultantDs into a
            // Consultant object

            consultantData.Consultant = consultant;

            var timesheetService = new TimesheetDataService( );
            var timesheetDataSet = timesheetService
                .GetTimesheets( consultantId );

            IList<Timesheet> timesheets = new List<Timesheet>( );
            // Insert code to translate the timesheetDs into a List
            // of Timesheet objects
            consultantData.Timesheets = timesheets;

            return consultantData;
        }
    }
```

The relevant data services would consist of standard ADO.NET code for returning datasets to this class. Figure 1 shows how this request might look diagrammatically.

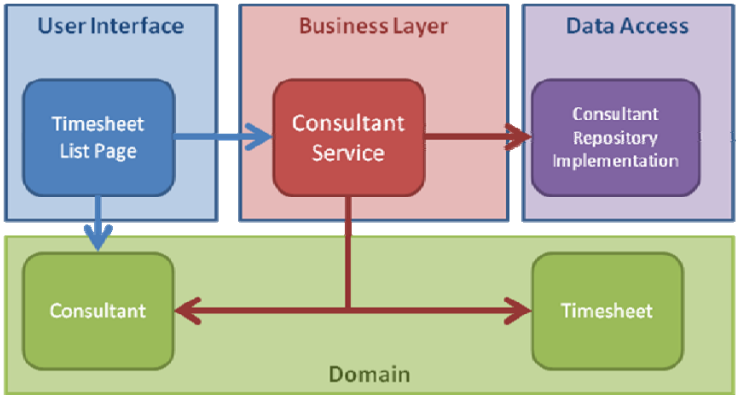


Figure 1: Traditional n-tier representation of a data request. Note that often, the Domain layer doesn't exist.

There is nothing magical about this code. Regardless of whether the use of DataSets makes you cringe, it can be used quite successfully, and has been for several years.

But to a growing number of people (including us), this type of code obfuscates the real reason for creating the application. There is talk of services and datasets and data objects which overshadow the core objects we are interested in: Consultants and Timesheets. As it is, we've encountered many applications that wouldn't even bother taking the time to convert the DataSets into objects and simply forward them on to the web page for processing. Indeed, Microsoft doctrine and training encourages this.

Instead, some argue that we should start from the core objects that the application is abstracting. That is, we should focus on the domain, rather than the database and the servers and the programming language.

This approach, made popular by Eric Evans in his book *Domain Driven Design* (as well as Jimmy Nilsson's *Applying Domain-Driven Design and Patterns*), has led to the domain-centric approach we use when we move a brownfield application into layers (see Figure 2).

To return to our list of timesheets, we'll sketch out a possible way of doing this with a focus on the domain.

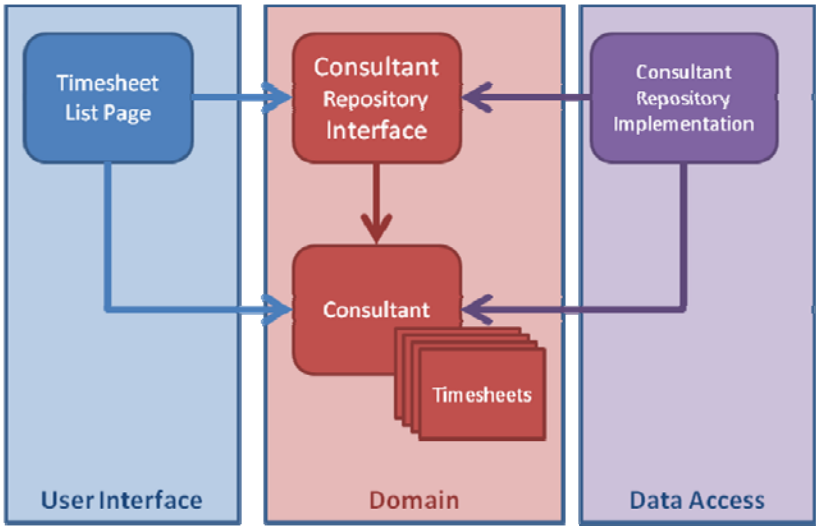


Figure 2: A domain-centric approach

This doesn't look too different from the traditional n-tier approach. It's almost as if we've combined the business logic layer and the domain.

The major difference is in the direction of the arrows. They all flow into the Domain layer. There is no sense of hierarchy, where a request is passed from one layer to the next.

In this version, the web page holds a reference to a repository. In this case, the `IConsultantRepository`. This interface lives in our domain along with the actual domain objects (Consultant and Timesheet). The implementation, on the other hand, is stored in our data access layer.

This is how we are able to use repositories as domain concepts while still divorcing them from the database-specific code needed to implement them. Without getting into too many details of Domain Driven Design, repositories are responsible for retrieving objects from a data store and saving them back again. It provides the client code with a well-defined interface (aka. contract) while still decoupling it from the underlying database technology.

Where do you get your repositories?

If you are new to the idea of repositories, you may be wondering, "If I create a `ConsultantRepository` in my client code, doesn't that mean I need a reference to the data access from my UI?" which should raise some alarm bells.

There are different ways to avoid linking your UI layer to your data access layer. You could use a Factory to create it. Or use a Service Locator. Their names are intuitive enough that you should be able to tell what they do. This is good because we don't have the space to go into lengthy discussions of either.

Another popular way is through the use of an Inversion of Control container.

The salient point is that you can have the best of both worlds. Use a domain interface in the client while the implementation is neatly tucked away in the data access layer.

Figure 2 is a much simplified version of the architecture we favor, as it ignores concepts such as logging and data translation services and unit tests. To bring it back to the abstract, Figure 3 is a depiction of what we are working toward.

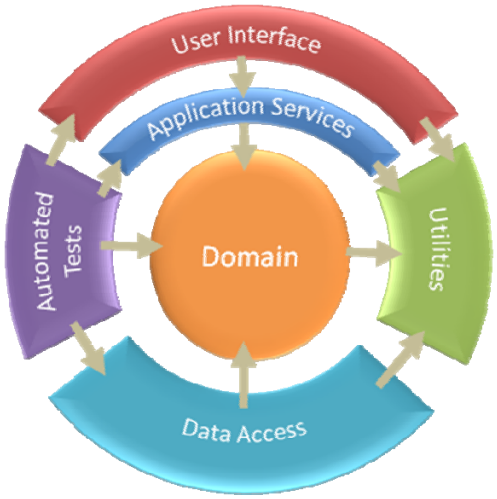


Figure 3: Domain-centric architecture we are working toward

A couple of notes on this diagram. Firstly, notice that the Domain is at the centre of the diagram. This is to emphasize its importance and nothing more. We could flatten this diagram into a more rectangular version to show that it isn't vastly different from the traditional n-tier architecture.

Secondly, the Utilities wedge needs some explanation. They are vertical concerns, such as logging or performance monitoring, that span most or all layers of our application. Because of this, all aspects of our architecture refer to it.

The Automated Tests wedge is similar to our Utilities but in the other direction. It needs to refer to each layer rather than be referenced by each one. Specifying it in coupling metrics, the Utilities wedge has high afferent coupling while the Automated Tests wedge has high efferent coupling.

Finally, our Data Access wedge is kind of an outsider. There are no references to it. So how (and where) do we instantiate our data access classes? We alluded to this in the sidebar above. The interfaces for much of our data access classes are kept in the domain. To get concrete implementations, we rely on factories, service locators, and/or IoC containers. In the case of the first two, we would likely need to add another layer to the diagram. It would lie between the Application Services and the Data Access. Technically, an IoC takes the place of this layer so using it does not obviate the need for it. But including it in our diagram would require more arrows and there are already more than we'd like in it.

There is much more to be said about Domain-Driven Design but this is a good primer into a somewhat non-traditional layering idea.