Microsoft

Adaptive Code

Agile coding with design patterns and SOLID principles

Second Edition

Best practices

Gary McLean Hall

Microsoft

Adaptive Code: Agile coding with design patterns and SOLID principles

Second Edition

Gary McLean Hall

The user interface layer sits on top of the data access layer and can make use of it. However, as discussed previously in relation to assembly references, the user interface layer should not make direct reference to any of the data access layer's implementation assemblies. There should be a strict separation between the interface and implementation assemblies of the two layers. This makes the layering diagram look a little more like Figure 3-19.

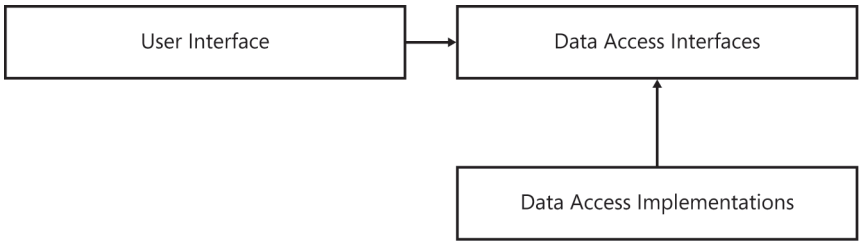

FIGURE 3-19 The two layers are separated into implementation assemblies and interface assemblies.

Each layer is the combination of an abstraction of the functionality that a higher layer depends on, along with an implementation of this abstraction. If a layer above starts referencing part of the implementation of a layer, that layer is called a *leaky abstraction*. The dependencies of that layer's implementation will begin to leak into layers further up the stack, resulting in avoidable dependencies.

Data access The responsibilities of the data access layer are:

- Servicing queries for data.
- Serializing and deserializing object models to and from a relational model.

The implementation of the data access layer can be just as varied as that of the user interface layer. This layer typically includes some kind of persistent data store that could include a relational database such as Microsoft SQL Server, Oracle, MySQL, or PostgreSQL or a document database such as MongoDB, RavenDB, or Riak. In addition to the data storage mechanism, there is likely to be one or more supporting assemblies for executing queries or insert/update/delete commands by calling stored procedures, or for mapping data to a relational database via Entity Framework or NHibernate.

Data access layers should be hidden behind interfaces that do not depend on any of these technology choices. As with all interfaces, there should be no reference to a third-party dependency, thus keeping clients separated from the choice of implementation.

A well-designed data access layer is reusable across multiple applications. If two different user interfaces require the same data but present it in different forms, the same data access layer can be shared between them. Imagine an application that runs across multiple platforms: Windows 10 and Windows Phone 10. Both have different user interface requirements, but each could use the same data access layer.

As with any architecture, using only two layers has some tradeoffs that must be considered carefully before adoption. The two-layer architecture is a good choice when:

- There is little or no logic to the application beyond some trivial validation. This can easily be encapsulated in either the data access layer or the user interface layer.
- The application performs mainly CRUD operations on data. Creating, reading, updating, and deleting data becomes more difficult with every additional layer placed between the user interface and the data itself.
- Time is short. If only a prototype or a bootstrap needs to be developed, limiting the number of layers can save a lot of time, and the feasibility of a proof of concept can be ascertained.

However, the two-layer architecture has some obvious drawbacks and is a bad choice when:

- There is significant logic in the application, or logic is subject to change. Any logic placed in the user interface layer or data access layer is technically a pollution of that layer and decreases its flexibility and maintainability.
- The application is certain to outgrow two layers within one or two sprints. Any concessions made to obtain quick feedback are not worth the investment if that architecture will only last a matter of weeks.

The two-layer architecture is still very much a viable alternative. Too many developers are enchanted by the latest architectural trend and overlook simpler designs. This causes an otherwise trivial application to receive feedback too late and makes it fragile and hard to maintain. Often, the simplest thing possible is the right thing to do.

Three layers

The three-layer architecture adds an extra layer between the user interface and the data access layer. This is the logic layer. The addition of the logic layer allows the application to encapsulate more complex processing. The logic layer, like the data access layer, can be reused across multiple applications, so it need not be implemented multiple times. Figure 3-20 shows the typical three-layer architecture.

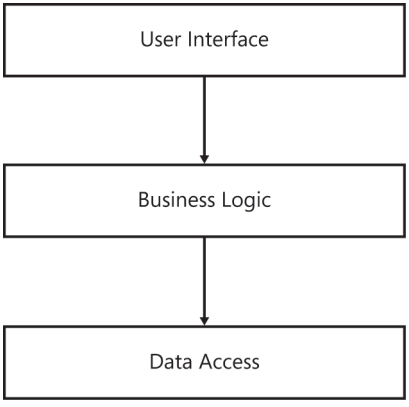

FIGURE 3-20 The third layer contains processing or business logic for the application.

Again, like the data access layer, the logic layer provides interface and implementation assemblies to clients, to avoid a leaky abstraction.

Note Although the three-layer architecture is very common for web applications, it is typically deployed in only two *tiers*. One node handles the database, and another node handles almost everything else: the user interface, the logic, and even part of the data access.

Business logic The business logic layer's responsibilities are to:

- Handle commands from the user interface layer.
- Model the business domain to capture business processes, rules, and workflow.

The logic layer might be a command processor that receives commands from the user via the user interface layer and, by collaborating with the data access layer, solves a specific problem or executes a particular task. It could also be a fully developed domain model that aims to map a business's processes into software. For the latter, it is common for the data access layer to include an Object/Relational Mapping (ORM) component so that the logic layer can be implemented directly into classes, possibly by using domain-driven design (DDD). In a domain model, there should be no dependencies, either further down the stack or via some implementation-specific technology. For example, the domain model's assemblies should have no dependencies on an ORM library. Instead, a separate mapping assembly should be created that is implementation-specific and instructs the ORM how to map to the domain model. This allows the domain model's core classes to be reused without depending on the ORM, and the ORM could be replaced without affecting the domain model or its clients. Figure 3-21 shows a possible implementation of a logic layer that uses a domain model.

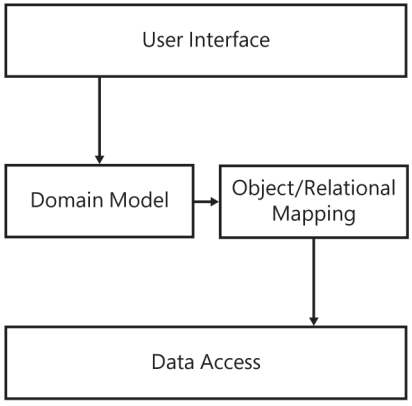

FIGURE 3-21 The assemblies of a domain model collaborate to form a logic layer.

The addition of a logic layer is necessary when there is complex logic in the application, such as business rules that aim to reflect the real-world workflows of people's jobs. Even if the logic is not particularly complex but changes often, this is a good argument for introducing a separate layer for encapsulating this behavior. It simplifies the user interface and data access layers, allowing them to concentrate fully on their only purpose.

Cross-cutting concerns

Sometimes a component's responsibilities are not easily limited to a single layer. Functions such as auditing, security, and caching can permeate through the entire application, because they are applicable at *every* layer. Tracing the code's actions at each method call and return, for example, is a useful debugging tool when the application has been deployed and if the Visual Studio debugger cannot be attached to step through the code. You can manually produce an output of the values of parameters as they are passed around, and the return values of various methods, as shown in Listing 3-24.

LISTING 3-24 Manually applying cross-cutting concerns quickly swamps the real intent of the code.

```
public void OpenNewAccount(Guid ownerID, string accountName, decimal openingBalance)
{
    log.WriteInfo("Creating new account for owner {0} with name '{1}' and an opening
   balance of {2}", ownerID, accountName, openingBalance);

    using(var transaction = session.BeginTransaction())
    {
        var user = userRepository.GetByID(ownerID);
        user.CreateAccount(accountName);
        var account = user.FindAccount(accountName);
        account.SetBalance(openingBalance);

        transaction.Commit();
    }
}
```

This is laborious and error-prone, and it instantly pollutes every method with irrelevant boilerplate code, increasing the noise-to-signal ratio. Instead, you can factor out such cross-cutting concerns into encapsulated functionality and apply them to the code in a much less invasive fashion. The most common way of adding functionality non-invasively is through aspect-oriented programming.

Aspect-oriented programming (AOP) is the application of cross-cutting concerns—or aspects—to multiple layers in the code. The .NET Framework has several AOP libraries to choose from (search NuGet for *AOP*), but the examples given here are for PostSharp, which has a free Express version, though with reduced functionality. Listing 3-25 shows tracing code factored out into a PostSharp aspect and applied as an attribute to some methods.

LISTING 3-25 Aspects are a great way to implement cross-cutting concerns.

```
[Logged]
[Transactional]
public void OpenNewAccount(Guid ownerID, string accountName, decimal openingBalance)
{
    var user = userRepository.GetByID(ownerID);
    user.CreateAccount(accountName);
    var account = user.FindAccount(accountName);
    account.SetBalance(openingBalance);
}
```

The two attributes decorating the `OpenNewAccount` method provide the same functionality as shown in Listing 3-24, but the intent of the method is clearer. The `Logged` attribute writes information about the method call to a log, including parameter values. The `Transactional` attribute wraps the method in a database transaction and commits the transaction on success or rolls back the transaction on failure. The key here is that both of these attributes are generic enough to be applied to *any* method, not specifically this one, so they can be reused many times.

Asymmetric layering

All of the users' requests to an application occur through the provided user interface. However, the path that the requests follow after that is not necessarily always the same. The layering could be asymmetrical, depending on the type of request being made. This is motivated by the need to be pragmatic and to consider whether the layering in place is overkill or even insufficient for some requests.

A pattern of asymmetric layering that has rapidly gained popularity in the last few years is Command/Query Responsibility Segregation (CQRS). Before discussing CQRS, which is an architectural pattern, a grounding in its method-level influencer, command/query separation, is required.

Command/query separation

Bertrand Meyer, in his book *Object-Oriented Software Construction* (Prentice Hall, 1997), used the phrase *command/query separation (CQS)* to explain that all object methods should be one of only two things: a *command* or a *query*.

Commands are imperative calls to action, requiring the code to *do* something. These methods are allowed to change the state of a system but should not also return a value. Listing 3-26 shows an example of a CQS-compliant command method, followed by one that is noncompliant.

LISTING 3-26 CQS-compliant and non–CQS-compliant command methods.

```
// Compliant command
Public void SaveUser(string name)
{
    session.Save(new User(name));
}
// Non-compliant command
public User SaveUser(string name)
{
    var user = new User(name);
    session.Save(user);
    return user;
}
```

Queries are requests for data, requiring the code to *get* something. These methods return data to calling clients but should not also change the state of a system. Listing 3-27 shows an example of a CQS-compliant query method, followed by one that is noncompliant.

LISTING 3-27 CQS-compliant and non–CQS-compliant query methods.

```
// Compliant query
Public IEnumerable<User> FindUserByID(Guid userID)
{
    return session.Get<User>(userID);
}
// Non-compliant query
public IEnumerable<User> FindUserByID(Guid userID)
{
    var user = session.Get<User>(userID);
    user.LastAccessed = DateTime.Now;
    return user;
}
```

Commands and queries are thus differentiated by the presence of a return value. If a method returns a value (and is CQS-compliant) you can safely assume that it does not change any state of the object. The advantage here is that you can reorder query calls knowing that they have no other effect on the object. A method having no return value (and that is CQS-compliant) indicates that you can assume that it does change the state of the object. For these calls, you would have to be more careful in your call order.

Command/Query Responsibility Segregation

The Command/Query Responsibility Segregation pattern is attributed to Greg Young. The pattern is the application of CQS at an architectural level and is an example of asymmetric layering. Commands and queries follow much the same rules as with CQS, but CQRS goes one step further: it acknowledges that commands and queries might need to follow different paths through the layering in order to be best handled.

An example of where minimal CQRS can be applied is when you are developing a three-layer architecture with a domain model. In this instance, the domain model is only ever used by the commanding side of the application, with a much simpler two-layer architecture used for the querying side. Figure 3-22 exemplifies this design.

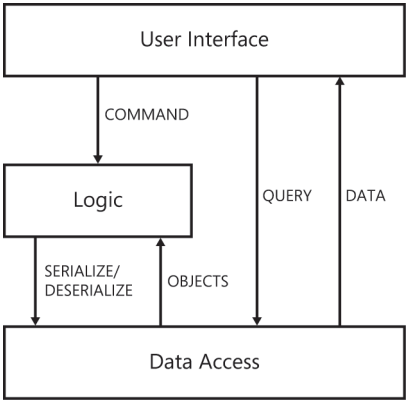

FIGURE 3-22 Domain models should only be used for handling commands.