# Cohesion And Coupling

Jeremy Miller

**Contents**

Much of software design involves the ongoing question, where should this code go? I'm constantly looking for the best way to organize my code to make it easier to write, easier to understand, and easier to change later. If I structure my code well, I'll go on to fame and glory. Structure it badly, and the developers who follow me will curse my name for eternity.

In particular, I would like to achieve three specific things with my code structure:

1. Keep things that have to change together as close together in the code as possible.
2. Allow unrelated things in the code to change independently (also known as orthogonality).
3. Minimize duplication in the code.

To achieve these three goals, I need some tools to help me know where new code should go and some other tools to help me recognize when I've put code in the wrong place.

By and large, these goals are closely related to the classic code qualities of cohesion and coupling. I achieve these goals by moving toward higher cohesion and looser coupling. Of course, first we need to understand what these qualities mean and why coupling and cohesion are helpful concepts. Then I'd like to discuss some of what I'm going to call "design vectors" that help us move toward better structures and recognize when we need to move away from bad structures that have crept into our code.

Just as a note, I'm the primary developer of an open source inversion of control (IOC) tool called StructureMap. I'm going to be using some real-life examples from StructureMap to illustrate the design problems that these vectors address. In other words, don't make the same mistakes that I made.

## Decrease Coupling

You can't help but hear the terms "loose coupling" or "tight coupling" in almost any discussion on software design. Coupling among classes or subsystems is a measure of how interconnected those classes or subsystems are. Tight coupling means that related classes have to know internal details of each other, changes ripple through the system, and the system is potentially harder to understand. **Figure 1** shows a contrived sample of a business-processing module that is very tightly coupled to various other concerns besides the business logic.

**Figure 1 Tightly Coupled Code**

```
public class BusinessLogicClass {
  public void DoSomething()  {
    // Go get some configuration
    int threshold =
      int.Parse(ConfigurationManager.AppSettings["threshold"]);

    string connectionString =
      ConfigurationManager.AppSettings["connectionString"];

    string sql =
      @"select * from things
        size > ";

    sql += threshold;

    using (SqlConnection connection =
      new SqlConnection(connectionString)) {

      connection.Open();

      SqlCommand command = new SqlCommand(sql, connection);
      using (SqlDataReader reader = command.ExecuteReader()) {
        while (reader.Read()) {
          string name = reader["Name"].ToString();
          string destination = reader["destination"].ToString();

          // do some business logic in here
          doSomeBusinessLogic(name, destination, connection);
        }
      }
    }
  }
}
```

Let's say that what we really care about is the actual business processing, but our business logic code is intertwined with data-access concerns and configuration settings. So what's potentially wrong with this style of code?

The first problem is that the code is somewhat hard to understand because of the way the different concerns are intertwined. I'll discuss this further in the next section on cohesion.

The second problem is that any changes in data-access strategy, database structure, or configuration strategies will ripple through the business logic code as well because it's all in one code file. This business logic knows too much about the underlying infrastructure.

Third, we can't reuse the business logic code independent of the specific database structure or without the existence of the AppSettings keys. We also can't reuse the data-access functionality embedded in the BusinessLogicClass. That coupling between data access and business logic might not be an issue, but what if we want to repurpose this business logic for usage against data entered directly into an Excel spreadsheet by analysts? What if we want to test or debug the business logic by itself? We can't do any of that because the business logic is tightly coupled to the data-access code. The business logic would be a lot easier to change if we could isolate it from the other concerns.

To summarize, the goals behind achieving loose coupling between classes and modules are to:

1. Make the code easier to read.
2. Make our classes easier to consume by other developers by hiding the ugly inner workings of our classes behind well-designed APIs.
3. Isolate potential changes to a small area of code.
4. Reuse classes in completely new contexts.

## Code Smells

It's obviously good to know how to do the right things when designing new code, but it might be even more important to recognize when your existing code or design has developed problems. Like inappropriate intimacy, "code smell" (defined by Martin Fowler in the book

Refactoring: Improving the Design of Existing Code) is a tool that you can utilize to spot potential problems in code.

A code smell is a sign that something may be wrong in your code. It doesn't mean that you need to rip out your existing code and throw it away on the spot, but you definitely need to take a closer look at the code that gives off the offending "smell." A code smell can be spotted by simple inspection. Learning about code smells is a powerful tool to remove little problems in your code and class design before those problems become bigger issues.

Many, if not most, of the commonly described code smells are signs of poor cohesion or harmful tight coupling. Here are some other examples:

**Divergent Changes** A single class that has to be changed in different ways for different reasons. This smell is a sign that the class is not cohesive. You might refactor this class to extract distinct responsibilities into new classes.

**Feature Envy** A method in ClassA seems way too interested in the workings and data fields of ClassB. The feature envy from ClassA to ClassB is an indication of tight coupling from ClassA to ClassB. The usual fix is to try moving the functionality of the interested method in ClassA to ClassB, which is already closer to most of the data involved in the task.

**Shotgun Surgery** A certain type of change in the system repeatedly leads to making lots of small changes to a group of classes. Shotgun surgery generally implies that a single logical idea or function is spread out over multiple classes. Try to fix this by pulling all the parts of the code that have to change together into a single cohesive class.

## Increase Cohesion

The academic definition of cohesion is that it is a measure of how closely related all the responsibilities, data, and methods of a class are to each other. I like to think of cohesion as a measure of whether a class has a well-defined role within the system. We generally consider high cohesion to be a good thing and repeat the words "highly cohesive" like a mantra. But why?

Let's think of coding as having a conversation with the computer. More accurately, we're having several simultaneous conversations with the computer. We're having conversations about how security is carried out, how the infrastructure concerns are supposed to behave, and what the business rules are.

When you're at a loud party with a lot of different conversations going on at once, it can be difficult to focus on the one conversation you're trying to have. It's much easier to have a conversation in a quiet setting where there's only one conversation going on.

An easy test for cohesion is to look at a class and decide whether all the contents of the class are directly related to and described by the name of the class—vague class names, such as InvoiceManager, do not count. If the class has responsibilities that don't relate to its name, those responsibilities probably belong to a different class. If you find a subset of methods and fields that could easily be grouped separately under another class name, then maybe you should extract these methods and fields to a new class.

To illustrate, if you find methods and data in your TrainStation, Train, and Conductor classes that seem to most accurately fit the theme of the TrainSchedule class, move those methods and data into the TrainSchedule. One of my favorite sayings about design is applicable here: put code where you'd expect to find it. It's most logical to me that functionality involving a train schedule would be in the TrainSchedule class.

To stretch my earlier conversation analogy, a system consisting of cohesive classes and subsystems is like a well-designed online discussion group. Each area in the online group is narrowly focused on one specific topic so the discussion is easy to follow, and if you're looking for a dialog on a certain subject, there's only one room you have to visit.

## Eliminate Inappropriate Intimacy

Inappropriate intimacy refers to a method in a class that has too much intimate knowledge of another class. Inappropriate intimacy is a sign of harmful, tight coupling between classes. Let's say that we have a business logic class that calls an instance of class DataServer1 to get the data it needs for its business-logic processing.**Figure 2** shows an example. In this case, the Process method has to know a lot of the inner workings of DataServer1 and a bit about the SqlDataReader class.

**Figure 2 Inappropriate Intimacy**

```
public void Process() {
  string connectionString = getConnectionString();
  SqlConnection connection = new SqlConnection(connectionString);
  DataServer1 server = new DataServer1(connection);

  int daysOld = 5;
  using (SqlDataReader reader = server.GetWorkItemData(daysOld)) {
    while (reader.Read()) {
      string name = reader.GetString(0);
      string location = reader.GetString(1);

      processItem(name, location);
    }
  }
}
```

Now, let's rewrite the code in **Figure 2** to remove the inappropriate intimacy:

```
public void Process() {
  DataServer2 server = new DataServer2();
  foreach (DataItem item in server.GetWorkItemData(5)) {
    processItem(item);
  }
}
```

As you can see in this version of the code, I've encapsulated all the SqlConnection and SqlDataReader object manipulation inside the DataServer2 class. DataServer2 is also assumed to be taking care of its own configuration, so the new Process method doesn't have to know anything about setting up DataServer2. The DataItem objects returned from GetWorkItemData are also strongly typed objects.

Now, let's analyze the two versions of Process method against some of the goals of loose coupling. First, what about making the code easier to read? The first and second versions of Process perform the same basic task, but which is easier to read and understand? Personally, I can more easily read and understand the business logic processing without wading through data-access code.

How about making our classes easier to consume? Consumers of DataServer1 have to know how to create a SqlConnection object, understand the structure of the DataReader returned, and iterate through and clean up the DataReader. A consumer of DataServer2 only has to call a no-argument constructor and then call a single method that returns an array of strongly typed objects. DataServer2 is taking care of its own ADO.NET connection setup and cleaning up open DataReaders.

And as for isolating potential changes to a small area of code? In the first version of the code, almost any change in the way DataServer works would impact the Process method. In the second, more encapsulated version of DataServer, you could switch the data store to an Oracle database or to an XML file without any effect on the Process method.

## The Law of Demeter

The Law of Demeter is a design rule of thumb. The terse definition of the law is: only talk to your immediate friends. The Law of Demeter is a warning about the potential dangers of code, as in **Figure 3** .

## ⊟ Figure 3 Breaking the Law

```
public interface DataService {
  InsuranceClaim[] FindClaims(Customer customer);
}

public class Repository {
  public DataService InnerService { get; set; }
}

public class ClassThatNeedsInsuranceClaim {
  private Repository _repository;

  public ClassThatNeedsInsuranceClaim(Repository repository) {
    _repository = repository;
  }

  public void TallyAllTheOutstandingClaims(Customer customer) {
    // This line of code violates the Law of Demeter
    InsuranceClaim[] claims =
      _repository.InnerService.FindClaims(customer);
  }
}
```

The ClassThatNeedsInsuranceClaim class needs to get InsuranceClaim data. It has a reference to the Repository class, which itself has a DataService object. ClassThatNeedsInsuranceClaim reaches inside Repository to grab the inner DataService object, then calls Repository.FindClaims to get its data. Note that the call to _repository.InnerService.FindClaims(customer) is a clear violation of the Law of Demeter because ClassThatNeedsInsuranceClaim is directly calling a method on a property of its Repository field. Now please turn your attention to **Figure 4** , which shows another sample of the same code, but this time it is following the Law of Demeter.

## ⊟ Figure 4 Better Decoupling

```
public class Repository2 {
  private DataService _service;

  public Repository2(DataService service) {
    _service = service;
  }

  public InsuranceClaim[] FindClaims(Customer customer) {
    // we're simply going to delegate to the inner
    // DataService for now, but who knows what
    // we want to do in the future?
    return _service.FindClaims(customer);
  }
}

public class ClassThatNeedsInsuranceClaim2 {
  private Repository2 _repository;

  public ClassThatNeedsInsuranceClaim2(
    Repository2 repository) {
    _repository = repository;
  }
  public void TallyAllTheOutstandingClaims(
    Customer customer) {
    // This line of code now follows the Law of Demeter
    InsuranceClaim[] claims = _repository.FindClaims(customer);
  }
}
```

What did we accomplish? Repository2 is easier to use than Repository because you have a direct method to call for the InsuranceClaim information. When we were violating the Law of Demeter, the consumers of Repository were tightly coupled to the implementation of Repository. With the revised code, I have more ability to change the Repository implementation to add more caching or swap out the underlying DataService with something completely different.

The Law of Demeter is a powerful tool to help you spot potential coupling problems, but don't follow the Law of Demeter blindly. Violating the Law of Demeter does add tighter coupling to your system, but in some cases you may judge that the potential cost of coupling to a stable element of your code is less expensive than writing a lot of delegation code to eliminate the Law of Demeter violations.

## Tell, Don't Ask

The Tell, Don't Ask design principle urges you to tell objects what to do. What you don't want to do is ask an object about its internal state, make some decisions about that state, then tell that object what to do. Following the Tell, Don't Ask style of object interaction is a good way to ensure that responsibilities are put in the right places.

**Figure 5** illustrates a violation of Tell, Don't Ask. The code is taking a purchase of some sort, checking for the possibility of a discount for purchases of more than $10,000, and finally examining the account data to decide whether there are sufficient funds. The previous DumbPurchase and DumbAccount classes are, well, dumb. The account and purchase business rules are both encoded in ClassThatUsesDumbEntities.

**Figure 5 Asking Too Much**

```
public class DumbPurchase {
  public double SubTotal { get; set; }
  public double Discount { get; set; }
  public double Total { get; set; }
}

public class DumbAccount {
  public double Balance { get; set;}
}

public class ClassThatUsesDumbEntities {
  public void MakePurchase(
    DumbPurchase purchase, DumbAccount account) {
    purchase.Discount = purchase.SubTotal > 10000 ? .10 : 0;
    purchase.Total =
      purchase.SubTotal*(1 - purchase.Discount);

    if (purchase.Total < account.Balance) {
      account.Balance -= purchase.Total;
    }
    else {
      rejectPurchase(purchase,
        "You don't have enough money.");
    }
  }
}
```

This type of code can be problematic in a couple of ways. In a system like this, you potentially have duplication because the business rules for an entity are scattered in procedural code outside of the entities. You might duplicate logic unknowingly because it's not obvious where the previously written business logic would be.

**Figure 6** shows the same code, but this time following the Tell, Don't Ask pattern. In this code, I moved the business rules for purchasing and accounts into the Purchase and Account classes themselves. When we go to make a purchase, we just tell the Account class to deduct the purchase from itself. Account and Purchase know about themselves and their internal rules. All a consumer of Account has to know is to call the Account.Deduct(Purchase, PurchaseMessenger) method.

**Figure 6 Telling Your App What to Do**

```
public class Purchase {
  private readonly double _subTotal;

  public Purchase(double subTotal) {
    _subTotal = subTotal;
  }

  public double Total {
    get {
      double discount = _subTotal > 10000 ? .10 : 0;
      return _subTotal*(1 - discount);
    }
  }
}

public class Account {
  private double _balance;

  public void Deduct(
    Purchase purchase, PurchaseMessenger messenger) {
    if (purchase.Total < _balance) {
      _balance -= purchase.Total;
    }
    else {
      messenger.RejectPurchase(purchase, this);
    }
  }
}

public class ClassThatObeysTellDontAsk {
  public void MakePurchase(
    Purchase purchase, Account account) {
    PurchaseMessenger messenger = new PurchaseMessenger();
    account.Deduct(purchase, messenger);
  }
}
```

The Account and Purchase objects are easier to use because you don't have to know quite so much about these classes to execute our business logic. We've also potentially reduced duplication in the system. The business rules for Accounts and Purchases can be effortlessly reused throughout the system because those rules are inside the Account and Purchase classes instead of being buried inside the code that uses those classes. In addition, it should be easier to change the business rules for Purchases and Accounts as those rules are found in just one place in the system.

Closely related to Tell, Don't Ask is the Information Expert pattern. If you have a new responsibility for your system, in what class should the new responsibility go? The Information Expert pattern asks, who knows the information necessary to fulfill this responsibility? In other words, your first candidate for any new responsibility is the class that already has the data fields affected by that responsibility. In the purchasing sample, the Purchase class knows the information about a purchase that you would use to determine any possible discount rates, so the Purchase class itself is the immediate candidate for calculating discount rates.

## Say It Once and Only Once

As an industry we've learned that writing reusable code deliberately is very expensive, yet we still try for reuse because of the obvious benefits. It may behoove us to look for duplication in our system and find ways to eliminate or centralize that duplication.

One of the best ways to improve cohesion in your system is to eliminate duplication wherever you spot it. It's probably best if you assume that you don't know exactly how your system is going to change in the future, but you can improve your code's ability to accept change by maintaining good cohesion and coupling in class structures.

I was peripherally involved years ago with a large shipping application that managed the flow of boxes on a factory floor. In its original incarnation, the system polled a message queue for incoming messages, then responded to those messages by applying a large set of business rules to determine where the boxes went next.

The next year the business needed to start the box-routing logic from a desktop client. Unfortunately, the business logic code was far too tightly coupled with the mechanisms for reading and writing to MQ Series queues. It was judged too risky to untangle the original business logic code from the MQ Series infrastructure, so the entire corpus of business rules were duplicated in a parallel library for the new desktop client. That decision made the new desktop client feasible, but it also made all future efforts more difficult because each change to the box-routing logic required a parallel change to two very different libraries—and those kinds of business rules change frequently.

This real scenario presents us with a couple of lessons. Duplication in the code had a real cost to the organization that built the system, and that duplication was largely caused by poor coupling and cohesion qualities in their class structure. This stuff can directly affect a company's profit and loss.

One way of looking at duplication in your code is as an opportunity to improve your design later. If you find yourself with two or more classes that duplicate some piece of functionality, you can say that the duplicated functionality must be a completely different responsibility. One of the best ways to improve the cohesion quality of your codebase is to simply pull out duplication into separate classes that can be shared across the codebase.

I learned the hard way that even innocuous-looking duplication can cause headaches. With the advent of generics in the Microsoft .NET Framework 2.0, many people started creating parameterized Repository classes that looked something like this:

```
public interface IRepository<T> {
  void Save(T subject);
  void Delete(T subject);
}
```

In this interface, T would be a domain entity like Invoice or Order or Shipment. Users of StructureMap wanted to be able to call this code and get a fully formed repository object that could handle a specific domain entity, such as an Invoice object:

```
IRepository<Invoice> repository =
  ObjectFactory.GetInstance<IRepository<Invoice>>();
```

It sounded like a great feature to have, so I sat down to add support for these kinds of parameterized types.

Making this change to StructureMap turned out to be very difficult because of code like this:

```
_PluginFamilies.Add(family.PluginType.FullName, family);
```

And code like this:

```
MementoSource source =
  this.getMementoSourceForFamily(pluginType.FullName);
```

And code like this as well:

```
private IInstanceFactory this[Type PluginType] {
  get {
    return this[PluginType.FullName];
  }
  set {
    this[PluginType.FullName] = value;
  }
}
```

Have you spotted the duplication yet? Not to get too involved with this sample, but I had an implicit rule that stated that objects related to a System.Type were stored by using the Type.FullName property as the key in a Hashtable. It's a tiny little bit of logic, but I had duplicated it all over the codebase. When I implemented generics, I determined that it would be better if I stored objects internally by the actual type instead of Type.FullName.

This seemingly minor change in behavior took me days instead of the hours I assumed up front because of how many times I had duplicated this little bit of data. The lesson learned here is that any rule in your system, no matter how seemingly trivial, should be expressed once, and only once.

## Wrapping Up

Cohesion and coupling applies to every level of a design and architecture, but I've mostly focused on fine-grained details at the class and method levels. Sure, you better get the big architectural decisions right—technology selection, project structure, and physical deployment are all important, but usually these are fairly constrained in the number of choices and the trade-offs are generally well understood.

I've found that the cumulative effect of the hundreds and thousands of little decisions you make at the class and method levels add up to have a more profound impact on the success of a project—and you've got a far greater range of choices and alternatives in regard to the little things. While it may not be true of life in general, in software design, sweat the small things.

There's a common attitude with developers that worrying about all this cohesion and coupling stuff is ivory tower theory that detracts from getting the work done. My feeling is that good cohesion and coupling qualities in your code will sustain productivity in your code over time. My strong advice is to internalize your recognition of cohesion and coupling qualities to the point where you don't have to consciously think about these qualities. Moreover, one of the best exercises I can recommend for improving your design skills is to revisit your previous coding efforts, try to find ways you could have improved the old code, and try to remember the elements of your past design that either made changing the code easy or made adjusting the code difficult.

Send your questions and comments to mmpatt@microsoft.com .

**Jeremy Miller** , a Microsoft MVP for C#, is also the author of the open source StructureMap (structuremap.sourceforge.net ) tool for Dependency Injection with .NET and the forthcoming StoryTeller ( storyteller.tigris.org ) tool for supercharged FIT testing in .NET.

---