**Extending Interface Based Design**



[**http://msdn.microsoft.com/rampup**](http://msdn.microsoft.com/rampup)

Contents

[Logical and Physical Design 3](#_Toc261506385)

[Using Types and Patterns to Partition an Application into Assemblies 4](#_Toc261506386)

[Using Interfaces to Guide Assembly Partitioning 4](#_Toc261506387)

[Using Patterns to Guide Assembly Partitioning 4](#_Toc261506388)

[Factory Pattern 5](#_Toc261506389)

[Using the Factory Pattern to Create an Assembly 5](#_Toc261506390)

[Assemblies as Components 6](#_Toc261506391)

[Physical Design Facilitates Testing 6](#_Toc261506392)

[Design for Testability 6](#_Toc261506393)

[Introducing Unit Tests 6](#_Toc261506394)

[Physical Design Challenges 7](#_Toc261506395)

[Summary 8](#_Toc261506396)

# Logical and Physical Design

To make software easier to change and evolve, we must write applications so that the dependencies of one type on another are eliminated or minimized. Hence, we must distinguish between essential and inessential coupling. Essential coupling results from the dependencies that exist in the real world. If a bank account has an owner, that dependency must exist in the application.

Inessential coupling comes from building the application. We could make the account class depend on the owner class, or we could use an interface so that the account class depends on the necessary behavior such as assigning an owner, or changing an owner. As we explained in the first unit, this approach minimizes the inessential coupling.

Design patterns allow us to minimize the connections caused by both essential coupling and inessential coupling. In the code example in the previous unit, we saw a simple example of using interfaces and the factory design pattern to minimize the coupling between the Customer class and the CustomerManager class. This relationship does not exist in the real world. The CustomerManager class only exists to isolate customer functionality from how customers are stored and retrieved. Design Patterns can also be used to minimize the changes needed when modifying essential coupling. Reduced essential and inessential coupling makes software easier to refactor and test.

Interface based design, design patterns, refactoring, and similar techniques are part of the logical design of software. In order to get the maximum benefit from logical design, one must pay attention to the physical design of the application as well. Integrated circuit engineers design components with regard not only to the logical design of the chip, but how it is physically implemented in silicon. Software designers must do the same if they are going to maximize the benefit from their software design choices. In .NET, this physical design is done through assemblies.

Software is written with a logical design in mind, but it is implemented through the deployment process. The fundamental idea is to group your classes into assemblies to minimize the dependencies among the assemblies with as little associated performance degradation as possible.

Paying attention to physical design brings several immediate benefits. It facilitates the division of an application among different programming groups. It minimizes the time needed to recompile the application resulting in faster development, and faster unit tests. It also makes it easier to write unit tests. It makes deployment easier because it minimizes the amount of physical changes needed to make an upgrade. It makes it easier to create and test emergency fixes whether you are in a desktop or web environment.

# Using Types and Patterns to Partition an Application into Assemblies

## Using Interfaces to Guide Assembly Partitioning

Software languages are composed out of two types of atoms: flow of control actions, and data structure primitives. Whether you have a simple loop that computes a simple sum, or a multi-threaded application that recursively builds a directed graph, the flow of control operates on the data structures. The flow of control will simply operate on the sections into which the data structures are partitioned.

As we discussed in the first unit, interfaces are the best way to extend the type system in an application. By making code depend on interfaces, we allow the implementation to evolve without impacting the calling code. Hence, the first step in partitioning the physical design is to separate the interface definitions from the code that provides the implementation.

Ideally, each assembly should have a strong name which allows the assembly to be versioned and have a unique identity. With unique identities you can then deterministically install and uninstall the application, have a bill of materials for each application, and use application and publisher policy to declare version compatibility. Nonetheless, everything we discuss here will work even if the assemblies do not have strong names.

Normally, you might think that we would put the interface definition in the same assembly as the code that implements it. This idea is wrong for two reasons. First, there might be multiple implementations of an interface. For example, you might be using interface inheritance, or have different implementation classes. An example might be having WankelEngine and InternalCombusionEngine implementations for the IEngine interface discussed in the previous unit. Second, the client would have to recompile their application if implementation changed even if the interface definition remained constant.

We could put the interface definition with the client code and make the implementation depend on the client. Here we have the interesting “upside” down idea that the implementation depends on the client. You still have a compilation dependency. A better idea would be to put the interface definitions in one or more separate assemblies, and treat them as you would the compiler types: data structure primitives of your application.

In our sample application we put ICustomer and ICustomerManager in a separate assembly. We would also put the Customer and CustomerManager classes in separate assemblies.

## Using Patterns to Guide Assembly Partitioning

A design pattern is a solution to a programming problem that has occurred so many times that it deserves a name. Patterns are not created, they are discovered.

Why should we care about patterns?

We benefit from previous successes and mistakes.

It is true that there is an enormous literature on patterns. It is often confusing, and some of it is obtuse. The pattern community sometimes seems as if it has a messianic complex, as if the study of patterns is the only key to software success. Nonetheless, understanding a few basic patterns will give you enormous insight and benefit. In subsequent units we shall be referring to several different design patterns that are extremely useful in partitioning and layering a software application. Once you get these patterns under your belt, you will be more comfortable exploring the patterns literature.

We use the Factory Pattern as our first pattern because it both nicely fits with our initial partitioning, and it is easy to understand.

### Factory Pattern

We used a CustomerManager class to encapsulate the behavior associated with mechanics of creating, updating, deleting, and reading (CRUD) customer objects. By introducing the ICustomerManager interface we made the main program depend only on the behavior, not the implementation of the CustomerManager class.

There is one problem, however, how is an instance of the CustomerManger class instantiated? If we use the *new* operator in the main program, all our partitioning work would be worthless because the main program would need access to this class to instantiate it. We use the Factory Pattern to provide the partitioning.

The Factory Pattern is an example of a creational pattern.[[1]](#footnote-1) A creational pattern introduces a level of indirection into the way a class is created. In our application, the CustomerManagerFactory’s CreateCustomerManager class returns the ICustomerManager interface. The main routine now depends on the CustomerManagerFactory class which is not as likely to change compared to the CustomerManager class itself.

We will have more to say about factory classes when we discuss Inversion of Control in a subsequent unit. Let us note for now that we can pass interfaces as parameters to a factory to make object construction very flexible.

### Using the Factory Pattern to Create an Assembly

The Factory Pattern divides the code into three natural sections:

the code that implements the pattern

the code that the pattern isolates

the code that uses the pattern.

In our example this corresponds to three classes: CustomerManagerFactory, CustomerManager, and the main application class. In then becomes natural to place each one of these and their associated classes in separate assemblies.

## Assemblies as Components

Now that we have divided our application into assemblies based on the appropriate logical design characteristics, we can evolve these assemblies independently of each other. While not all assemblies should be considered as components, groups of assemblies that can be compiled and test together can be considered as independent components.

If these components are reusable, that is an added benefit. Our real goal, however, is not to promote reuse, but partitioning that allows applications to evolve. Real reuse is very difficult to achieve.

In fact, we can now divide the application into completely separate solutions. Each solution references only the assemblies that it needs.

# Physical Design Facilitates Testing

## Design for Testability

Another reason for using patterns is to increase the testability of the software application. We discussed using the Factory Pattern to partition the main program from the CustomerManager functionality. By using this partitioning to create separate assemblies, it allows both the main program and the CustomerManager classes to be tested independently.

It is easy to see how the CustomerManager class can be tested independently by creating a test program that creates an instance of the class and tests its methods.

It is a little more difficult to see how the Factory Pattern allows for independent testing of the main program. In a subsequent unit we will discuss dependency injection to demonstrate how this could be done. For the moment we will just note that dependency injection would allow us to return an interface from the factory that implements an ICustomerManager implementation that uses a different implementation than CustomerManager. We might do this, for example, if the real CustomerManager uses a web service that is not always available, or is too time consuming for frequent unit testing. Returning a stub, or mock object, would allow us to independently test the main class.

## Introducing Unit Tests

To illustrate the independent testing of the CustomerManager class, we have used the Visual Studio Test Edition to create a test project in the CustomerManager solution.

[TestMethod]

public void CustomerManagerTest()

{

ICustomerManager cm = CustomerManagerFactory.CreateCustomerManager();

ICustomer customer = cm.AddCustomer("Peter Paul");

ICustomer customer2 = cm.GetCustomer(customer.Id);

Assert.AreEqual(customer.Name, customer2.Name, "Different customer

found.");

}

Obviously, in a real application we would have much more elaborate testing. Note how the use of the Factory Pattern and interfaces contribute to independent testing. Other patterns can be used to create the same effect.

# Physical Design Challenges

Nothing is an unmixed blessing in software design. Too many assemblies in an application can be problematic. Potential objections to this approach fall into two categories: assembly management and performance.

Minimizing the number of dependencies doesn't mean putting each class in a separate assembly. That would not only increase the runtime loading and introduce an assembly management problem, but it also actually would increase the number of dependencies.

Partition classes into assemblies based on the probability of change. With fewer dependencies it becomes easier to do pinpoint deployment, and create versions for smaller parts of the application. Different installations can deploy different features or bug fixes. While this may not always be a good idea, applications might be built with different features based on a license fee. One site might need a critical emergency fix that might break other parts of the application that other sites require. With better physical design it becomes clearer what versions of the software the developers are working against, and minimizes the need to "rebuild the world." Your compile, build, and test/debug cycles get faster.

If you specify assembly versions, increase the version number of a changed assembly before deploying it to quality assurance. Now development is working on the next version. If a bug is reported it is clear which version of the assembly has the problem.

Putting everything into separate assemblies is not a panacea. Imagine what would happen in a large application if every class and interface went into a separate assembly. Encapsulation is lost because some methods and classes that could be marked internal have to be marked public. We may have increased the amount of dependencies beyond your ability to manage them. Too many finely grained assemblies mean more assemblies have to be redeployed when you make only a small change to your application.

Loading an assembly imposes a performance penalty. You have to analyze the use patterns of your application. Do all the assemblies load at once during the application startup, or is the load distributed over the lifetime of the application? Does it matter to the users of the application? It obviously depends on whether you have a UI-intensive app or a web service. How long do the users spend with a UI-intensive app? Some users are willing to tolerate a little longer startup time if they are going to spend 10 minutes to an hour with the app. Other times they want the quickest startup possible.

More assemblies do increase the working set of the application. Does that matter? Again it depends on how big your working set is to begin with, and on the locality of reference in your application.

The problem with performance-related statements, of course, is that you can always find a set of circumstances to which they apply or do not apply. In applications, the biggest performance bottleneck is often the network latency, or the time the user thinks about what to do next. In such circumstances, any performance penalty from having more assemblies would be unnoticed.

# Summary

Good physical design facilitates the development of a software application by mulitiple teams, as well as increasing the testability of the application. It becomes much easier to enhance, or test parts of the application independent of each other.

Noticing what the fundamental types and patterns used in your application are a major factor in deciding how to partition your logical design into physical assemblies.

We will see the power of physical design as we evolve our assemblies in subsequent units.

1. Chapter 3 of *Design Patterns*, by Gamma, Helm, Johnson, and Vlissides, has several other examples. In the language of that book this is the Abstract Factory pattern because it provides an interface for creating objects without specifying the implementation classes. [↑](#footnote-ref-1)