**Partitioning and Layering Fundamentals**

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

Contents

[The Basic Problem 3](#_Toc252802460)

[How do you partition and layer an application? 4](#_Toc252802461)

[Fundamental Concepts 4](#_Toc252802462)

[Type 4](#_Toc252802463)

[Interface and Implementation 6](#_Toc252802464)

[Coupling 7](#_Toc252802465)

[Interface Composition vs. Class Inheritance 8](#_Toc252802466)

[Design Patterns 9](#_Toc252802467)

[Partitioning and Layering 10](#_Toc252802468)

[Summary 10](#_Toc252802469)

# The Basic Problem

Change is a fact of life in software development. Requirements change, technologies change, bugs get fixed, hardware is replaced. Nonetheless, software must adapt with these changes. Failure to build software in a way that allows for change will have severe consequences. Applications will not be used because they fail to meet current requirements. Businesses will loose their competitive edge because customers will not be happy. Ultimately, you will not be able to find work, if you cannot master writing software in a way that allows for change.

The goal of partitioning and layering an application is to reduce the connections or ***coupling*** between the different parts of code. With reduced coupling, the consequences or reprecussions from changing software is minimimzed or eliminated. With properly partitioned code layers it becomes possible to create focused unit tests. With focused unit tests it becomes possible to detect quickly any undesirable consequences of code modifications, including code modifications associated with partitioning an application.

This track will be an introduction to architecting and designing an application to minimize coupling through layering. Since applications and services are not written all at once, these techniques must be continually used, throughout the process of writing code. We will not focus on how to improve the partitioning and layering of an application where no thought was put into partitioning in the first place. You cannot, however, attempt to introduce partitioning in that kind of application without understanding what is presented in this track. For more information on partitioning and layering an existing code base, read “Working Effectively with Legacy Code” by Michael Feathers.

Partitioning an application into layers is one example of ***code refactoring***. Martin Folwer defines refactoring as “the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure. It is a disciplined way to clean up code that minimizes the chances of introducing bugs.”[[1]](#footnote-1)

Refactoring is an important concept in partitioning an application. You do not have to get it exactly right the first, or even the second or third time. You can refactor as appropriate. Nonethless, the possibility of refactoring should not hinder you from a good design.[[2]](#footnote-2)

This unit discusses the basic concepts behind partitioning and layering. Since it explains the basic principles used in all the other units, it is a little less concrete than the other units. The subsequent units will illustrate how to do this in a software application.

# How do you partition and layer an application?

## Fundamental Concepts

To understand how to partition and layer an application, we have to revisit several fundamental concepts of object oriented programming. While you may be familiar with these concepts, the best ways to partition code into layers is based on the accumulated experience gained over the past 20 years in the best way to apply object oriented principles.

### Type

Not surprisingly, object oriented programming is about programming in terms of objects. But what are objects? An object is a threefold definition of a piece of data:

* description of the data
* operations on the data, also known as the object’s behavior
* identity of the data.

The data description and the operations of the data represented the object’s ***type*** or definition. In other words, a type is a definition; an actual object represents a particular instance of that type.

In the Employee class, the name field is part of the type’s data description. The Pay method is part of the type’s behavior.

public class Employee

{

 string name;

 …

 void Pay()

 ….

}

To create an object, we instantiate an instance of the type:

Employee emp = new Employee();

This object is said to have identity because it refers to a particular employee.

With simple types that are built into a programming language such as an integer, the operations on the data are defined in the language. The data and the identity of the data are represented by a variable. Simple, built-in types are not particularly interesting for partitioning and layering.

Complex types are defined in a class[[3]](#footnote-3). These types are defined by developers, not the programming language. These types are found either in third party libraries or defined by the developers on your project. Usually these classes reference other classes; we say a particular class ***depends*** on another class.

Here is an example of an Auto class having a ***dependency*** on the Engine and Wheel classes.

class Auto

{

 Engine engine;

 Wheel[4] wheels;

void Drive() {…};

}

class Engine

{

 string manufacturer;

 int horsepower;

 void Start(){…};

 void Stop(){…};

}

class Wheel

{

 string manufacturer;

 float tirePressure;

 void Turn(){…};

}

In this example, the Auto class is built using ***object composition***. It is composed of Engine and Wheel classes.

It is likely that if we modify the Wheel or Engine class, the Auto class will have to change. This dependency however makes it hard for the Engine, Wheel, and Auto classes to be modified by different developers. Suppose we wanted to substitute a different Engine class, the Auto class at the very minimum would have to be recompiled. Further, a bug in one is likely to have an impact on the others. It is also very difficult to write unit tests that just test the functionality of each class by itself.

The entire goal of partitioning an application can now be stated simply. ***Write your applications so that the dependencies of one type on another are eliminated or minimized***.

### Interface and Implementation

The key is to make classes dependent on another class’ behavior, not its implementation. A programming interface describes the behavior of the type. The implementation describes the type’s data and how the behavior is implemented. C# uses the built-in interface type to describe the behavior of a class type. We might define the following interfaces:

interface IWheel

{

 void Turn();

}

interface IEngine

{

 void Start();

 void Stop();

}

We would then define the following classes:

class Engine : IEngine

{

 string manufacturer;

 int horsepower;

 void Start(){…};

 void Stop(){…};

}

class Wheel : IWheel

{

 string manufacturer;

 float tirePressure;

 void Turn(){…};

}

One way of defining the auto class is then:

class Auto

{

 IEngine engine;

 IWheel[4] wheels;

void Drive() {…};

}

Here we have used ***interface composition*** rather than object composition to build our type.

## Coupling

How does using interface composition help in reducing a dependency?

The Auto class is now independent of any particular engine or wheel class. If we change the Engine class to fix a bug or enhance it, or use a WankelEngine class that also implements IEngine, the Auto class does not have to be rebuilt. In addition, we can write focused unit tests for the Engine or Wheel classes to make sure they implement the proper behavior. Using interfaces allows objects to be dynamically composed at runtime.

So long as the engine or wheel class implements the appropriate interface, we have minimized the dependency. We have not elimninated the dependency, because the auto class is still dependent on the behavior of the engine or wheel.

This dependency on the behavior is ***essential coupling***. It is coupling based on the part of the real world that you are modelling. You cannot eliminate it because without it you would not have an application. In partitioning an application you want to remove ***inessential coupling***. This kind of coupling is an artifact of how you build your application. The programming details of how the objects are implemented should not be a dependency.

Consider an electrical analogy to make this clearer. A wall socket is an interface that minimizes the inessential coupling due to the physical shape of plugs and appliances. This wall socket interface cannot remove the essential behavioral coupling of voltage and amperage of standard current.

Nonetheless, we have made our application more complicated by introducing interfaces. Every time you insert a level of indirection in a program you increase the complexity. This is one of the enduring dilemmas of application partitioning that we will see over and over again: the tension between simplicity and ease of change.

Before you start putting interfaces on everything, you have to think where they are appropriate. As we will illustrate in each unit of this track, you must think of what the fault lines in your program are, and use interfaces there. Of course these fault lines will change over time, and you can refactor appropriately. Nonetheless, it is very hard to refactor yourself out of a bad design.

## OBJECT Composition vs. Class Inheritance

The previous Auto example demonstrated the advantage of using interface composition over object composition when designing types for partitioning and layering. What about inheritance? Is it a good idea to have an inheritance chain such as this?

class RacingCar : HighPerformanceCar : Auto

Inheritance defines the implantation of one type in terms of another. Class inheritance is defined statically at runtime. You cannot change it dynamically at runtime. Inheritance, however, has a much worse disadvantage. In general, you need to understand the behavior of the base classes in order to override behavior. Avoiding behavior dependencies is what we are trying to avoid when partitioning an application.

Let’s expand our RacingCar example.

class HighPerformanceCar

{

 virtual void Start()

{

 TurnIgnition();

 Press GasPedal();

}

…

}

class RacingCar : HighPerformanceCar

{

…

}

Suppose we want to override the Start method in the RacingCar class. Do we need to call the Start () method in our derived class, or are we going to completely replace it? To answer these and other similar questions we need to know the details of implementation of the Start () method in the base class. Hence, the often cited phrase: “inheritance breaks encapsulation.”

It would be much better to use object composition instead of class inheritance to avoid these problems. Even better would be to use interface composition.

## Design Patterns

Restricting yourself to essential coupling is not enough. Nor is using interface inheritance or object composition enough.

You want to make your interfaces easy to modify. Even within the implementation you want to minimize dependencies to make the implementation easier to modify. Using design patterns increases the flexibility of your application.

A design pattern is a solution to a programming problem that has occurred so many times that it deserves a name. In the subsequent units we shall be referring to several different design patterns.

Let’s consider a design pattern for our wall socket example to demonstrate how a pattern can be used to minimize essential coupling. Even with a wall socket interface, how do you design an applicance to work with 110 volt or 220 volt electrical grids? An electrical transformer is a pattern to modify the behavior to minimize effect of the essential coupling of voltage and current.

# Partitioning and Layering

***To partition and layer a software application, reduce the coupling or dependencies of complex types on each other.***

The code cast associated with this unit will demonstrate how to use interface based design to minimize the dependencies in an application.

# Summary

Partitioning and layering is achieved through a mixture of design and programming techniques. This will help you minimnize the impact of change on your application, as well as build well focused unit tests.

Develop your classes using interfaces rather than concrete classes.

Prefer composing your classes with other classes rather than using class inheritance.

Use patterns to further minimize dependencies.

# Futher Reading

Here is a bibliography for the entire track, not just this unit.

*Design Patterns*, Erich Gamma, et. al. Section 1.6, Chapter 3, Chapter 4 Façade Pattern

*Domain-Driven Design*, Eric Evans Chapters 4, 6

*Patterns of Enterprise Application Architecture*, Martin Fowler Chapters 1-3, 13, 18

*Working Effectively with Legacy Code*, Michael Feathers

1. Fowler, Martin, “Refactoring” p xvi. [↑](#footnote-ref-1)
2. How much up front design you need to do is not a discussion I have any intention of starting here. [↑](#footnote-ref-2)
3. A structure is a complex type. I am ignoring them because the reference type / value type dichotomy is not important here. [↑](#footnote-ref-3)