# Chapter 10

Capture The Moment

# Compositional Design

## Learning Objectives

- *State the definition of the term "simple aggregation"*
- *State the definition of the term "composite aggregation"*
- *Explain the difference between simple and composite aggregation*
- *Express simple and composite aggregation using Unified Modeling Language (UML) diagrams*
- *Express an association between two class types using a UML diagram*
- *Define the terms "has a", "contains", and "uses" in the context of compositional design*
- *Explain the difference between a dependency and an association*
- *Explain the client/server relationship between a containing and a contained class*
- *Demonstrate your ability to use both simple and composite aggregation in your program design*
- *Explain how to implement message passing between objects*
- *State the purpose and use of a UML sequence diagram*
- *Define the term "collaboration"*

## Introduction

Rarely does an application comprise just one class. In reality, applications are typically constructed from many classes, each providing a unique service. These classes *collaborate* with each other to implement the functionality of the application you are building. This chapter introduces you to the concepts and terminology associated with building complex application behavior from multiple classes. This is referred to as *compositional design* or *design by composition*.

The study of compositional design is the study of *aggregation* and *containment*. In this chapter you will learn the two primary aggregation associations: *simple* and *composite*. You will also learn how to use a UML class diagram to illustrate the static relationship between classes in a complex application. To do this you will need to know how to express simple and composite aggregation visually.

The study of aggregation also entails learning how a *whole* class accesses the services of its *part* classes. The concepts of *message passing* and *sequencing* will be demonstrated by introducing you to a new type of UML diagram known as the *sequence diagram*.

## Managing Conceptual And Physical Complexity

Programs that depend upon the services of multiple classes are inherently more complex than those that do not. This complexity takes two forms: 1) the *conceptual complexity* derived from the nature of the relationship between or among classes that participate in the design, and 2) the *physical complexity* that results from having to deal with multiple source files. In this chapter you will encounter programming examples that are both conceptually and physically more complex than previous examples.

You will manage a program's conceptual complexity by using object-oriented design principles in conjunction with UML to express a program's design. In this chapter you will learn the concepts of building complex programs using compositional design. You will also learn how to express compositional design using UML.

The physical complexity of the programs you write in this chapter must be managed through source file organization. If you have not already started to put related project source files in one folder, you will want to start doing so. This will make them easier to manage — at least for the purposes of this book. You will also need to change the way you compile your project source files. Until now, I have shown you how to compile source files one at a time. However, when you are working on projects that contain many source files, these files may have dependencies to other source files in the project. A change to one source file will require a recompilation of all the other source files that depend upon it.

If you use an integrated development environment (IDE) like Microsoft's Visual Studio, it will manage the source file dependencies automatically. If you are developing your projects with a simple text editor and the .NET Runtime Environment as recommended in chapter 2, then you can use Microsoft Build (MSBuild.exe) located in the .NET Framework folder. MSBuild is what Visual Studio uses to build projects. However, you don't need to use these tools just yet. As the next section explains, you can compile almost all of the projects in this book from the command line with the `csc` compiler tool.

### Compiling Multiple Source Files Simultaneously With csc

If you don't have time to learn MSBuild and don't want to bother with Visual Studio just yet, that's no problem. You can compile multiple source files simultaneously using the `csc` compiler tool. All you need to do is enter the names of the source files you want to compile following the `csc` command on the command line, as is shown in the following example:

  

```
csc SourceFileOne.cs SourceFileTwo.cs SourceFileThree.cs
```

List each source file on the command line, one after the other, separated by a space. You can compile as many source files as you require using this method.

If you are working on a project that has more than two or three source files, then you may want to create a batch file (Microsoft Windows) or a shell script (Linux, Mac OS X) that you can call to perform the compilation for you.

Another way to compile multiple related source files is to put them in a project directory and invoke the `csc` compiler tool as shown in the following example:

```
csc *.cs
```

This compiles all the source files in a given directory. You may, of course, provide an explicit directory path as shown in the following example:

```
csc c:\myprojects\project1\*.cs
```

If you have organized your project files into multiple subdirectories, you can use the `/recurse` compiler switch to recursively compile the source files. Recursive compilation starts with files located in the root project directory, and visits each subdirectory in turn. To compile recursively, use the `/recurse` compiler switch like so:

```
csc /recurse:*.cs
```

### Quick Review

There are two types of complexity: *conceptual* and *physical*. Manage conceptual complexity by using object-oriented design concepts and by expressing your object-oriented designs in UML. Manage physical complexity with the help of your IDE (Visual Studio) or with an automated build tool such as Microsoft Build (MSBuild). You can also manage physical complexity on a small scale by organizing your source files into project directories and compiling multiple files simultaneously using the `csc` compiler tool.

## Dependency vs. Association

A C# program built from many classes manifests several types of interclass relationships. Before continuing with this chapter, you must be clear in your understanding of the terms *dependency* and *association*.

A *dependency* is a relationship between two classes in which a change made to one class will have an effect on the behavior of the class or classes that depend on it. For example, say you have two classes, Class A and Class B. If Class A depends upon the behavior of Class B in a way that a change to Class B affects Class A, then Class A has a dependency on Class B. If you use a class in your program and write code that calls one or more of that class's interface methods, then your code is dependent upon that class. If its interface methods change, your code might break. If the behavior of those methods change, your program's behavior will change as well.

All but the most trivial programs you write will be chock full of dependency relationships between your classes and, at the very least, the classes you use in your programs that are supplied by the .NET Framework.

  

An *association* is a relationship between two classes that denotes a connection between those classes. An association implies a peer-to-peer relationship between the classes that participate in the association. If you have two classes, Class A and Class B, and there is an association between them, then Class A and Class B are linked together at the same level of importance. They may each depend on the other and the link between them will be navigable in one, or perhaps two, directions.

An association implies a dependency, eventually. Initially, an association signifies "...a semantic dependency between two entities..." (Booch) To complicate matters, the terms dependency and association have overloaded meanings in the literature. You can have many different types of dependencies, the most obvious of which is a direct dependency on the public interface offered by a particular type. Another type of dependency is inheritance, which is covered in detail in chapter 11. If you extend a type, the subtype has a dependency on its supertype. If you implement an interface, same thing. One way to break a dependency is to target an interface, which introduces a dependency on the interface type, but allows substitution of derived objects, which is itself another dependency, although more flexible.

The term peer-to-peer requires more clarification. Classes that participate in a peer-to-peer association can be said to engage in a least intimate collaboration. A peer-to-peer association occurs when one class simply relies upon the public interface methods of another class to exchange messages with that class.

An *aggregation* is a special type of association and is discussed in detail in the following section.

## Aggregation

An *aggregation* is an *association* between two objects that results in a whole/part relationship. An object comprising other objects (*i.e., the whole object*) is referred to as an *aggregate*. Objects used to build the whole object are called *part objects*. There are two types of aggregation: *simple* and *composite*. Each type of aggregation is discussed in detail below.

### Simple vs. Composite Aggregation

Aggregate objects are built from one or more part objects. An aggregate object can be a *simple aggregate*, a *composite aggregate*, or a combination of both. The difference between simple and composite aggregation is how the whole or aggregate object is linked to its part objects. The type of linking between an aggregate and its parts dictates who controls the lifetime (*creation and destruction*) of its part objects.

#### *The Relationship Between Aggregation And Object Lifetime*

The difference between simple and composite aggregation is dictated by who (*i.e.*, what object) controls the lifetime of the aggregate's part objects. This section discusses simple and composite aggregation in greater detail.

##### *Simple Aggregation*

If the aggregate object simply uses its part objects and otherwise has no control over their creation or existence, then the relationship between the aggregate and its parts is a *simple aggregation*. The part object can exist (*i.e., it can be created and destroyed*) independently of the simple aggregate object. This leads to the possibility that a part object can participate, perhaps simultaneously, in more than one simple aggregation relationship.

The .NET runtime garbage collector eventually destroys unreferenced objects, but as long as the simple aggregate object maintains a reference to its part object, the part object will be maintained in memory.

  

References to existing part objects can be passed to aggregate object constructors or other aggregate object methods as dictated by program design. This type of aggregation is also called *containment by reference*.

#### *Composite Aggregation*

If the aggregate object controls the lifetime of its part objects (*i.e., it creates and destroys them*) then it is a *composite aggregate*. Composite aggregates have complete control over the existence of their part objects. This means that a composite aggregate's part objects do not come into existence until the aggregate object is created.

Composite aggregate part objects are created when the aggregate object is created. Aggregate part creation usually takes place in the aggregate object constructor. However, in C#, there is no direct way for a programmer to destroy an object. You must rely on the .NET runtime garbage collector to collect unreferenced objects. Therefore, during an aggregate object's lifetime, it must guard against violating data encapsulation by not returning a reference to its part objects. Strict enforcement of this rule will largely be dictated by program design objectives. This type of aggregation is also called *containment by value*.

### Quick Review

An aggregation is an association between two objects that results in a whole/part relationship. There are two types of aggregation: *simple* and *composite*. The type of aggregation is determined by the extent to which the whole object controls the lifetime of its part objects. If the whole object simply uses the services of a part object but does not control its lifetime, then it's a simple aggregation. On the other hand, if the whole object creates and controls the lifetime of its part objects, then it is a composite aggregate.

## Expressing Aggregation In A UML Class Diagram

The UML class diagram can be used to show aggregate associations between classes. This section shows you how to use the UML class diagram to express simple and composite aggregation.

### Simple Aggregation Expressed In UML

Figure 10-1 shows a UML class diagram expressing simple aggregation between classes Whole and Part. The association is expressed via the line that links Whole and Part. The simple aggregation property is denoted by the hollow diamond shape used to anchor the line to the Whole class. Simple aggregation represents a *uses* relationship between the aggregate and its parts, since the lifetime of part objects are not controlled by the aggregate.

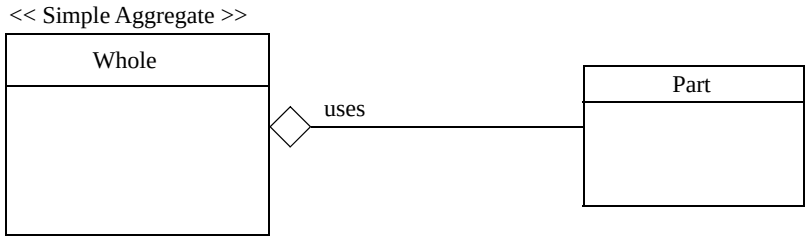


Figure 10-1: UML Class Diagram Showing Simple Aggregation

  

Figure 10-2 shows two different simple aggregate classes, Whole A and Whole B, sharing an instance of the Part class. Such a sharing situation may require thread synchronization. Threading is discussed in detail in chapter 16.

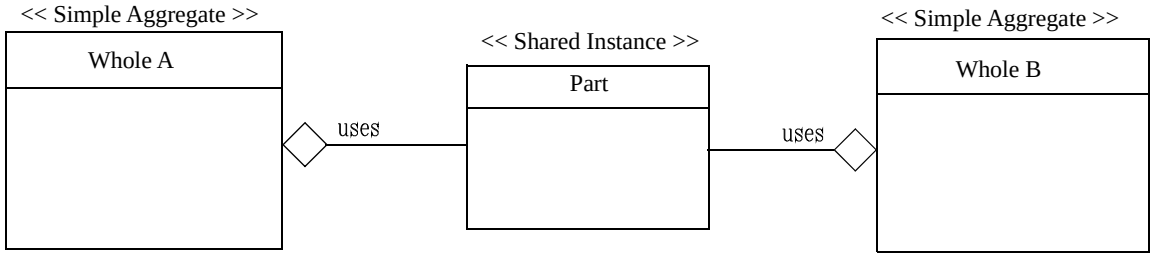


Figure 10-2: Part Class Shared Between Simple Aggregate Classes

### Composite Aggregation Expressed In UML

Composite aggregation is denoted by a solid diamond adorning the side of the aggregate class. Composite aggregate objects control the creation of their part objects, which means part objects belong fully to their containing aggregate. Thus, a composite aggregation denotes a *contains* or *has a* relationship between whole and part classes. Figure 10-3 shows a UML class diagram expressing composite aggregation between classes Whole and Part.

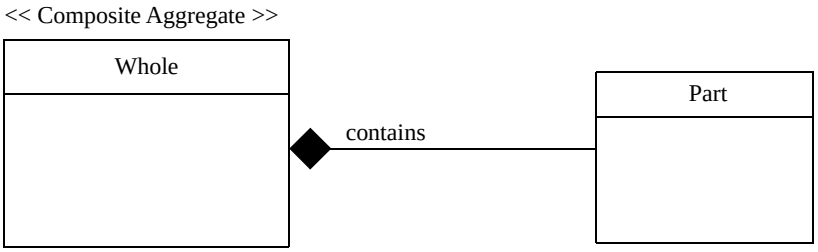


Figure 10-3: UML Class Diagram Showing Composite Aggregation

## Aggregation Example Code

At this point it will prove helpful to you to see a few short example programs that implement simple and composite aggregation before attempting a more complex example. Let's start with simple aggregation.

### Simple Aggregation Example

Figure 10-4 gives a UML class diagram showing a simple aggregate class named A that uses the services of a part class named B.

Class A has one attribute, its_b, which is a reference to an object of class B. A reference to a B object is passed into an object of class A via a constructor argument when an object of class A is created. Class A has another method named MakeContainedObjectSayHi() that returns `void`.

Class B has no attributes, one constructor method, and another method named SayHi(). Examples 10.1 and 10.2 give the code for these two classes.

  

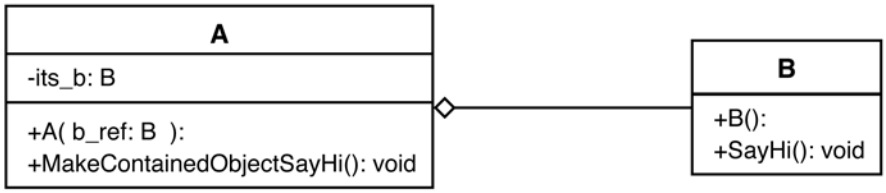


Figure 10-4: Simple Aggregation Example

*10.1 A.cs*

```
1   using System;
2
3   public class A {
4     private B its_b = null;
5
6     public A(B b_ref){
7       its_b = b_ref;
8       Console.WriteLine("A object created!");
9   }
10
11    public void MakeContainedObjectSayHi(){
12      its_b.SayHi();
13    }
14  }
```

Referring to example 10.1 — The its_b reference variable is declared on line 4 and initialized to null. The constructor takes a reference to a B object and assigns it to the its_b variable. The its_b variable is then used on line 12 in the MakeContainedObjectSayHi() method to call its SayHi() method.

*10.2 B.cs*

```
1   using System;
2
3   public class B {
4     public B(){
5       Console.WriteLine("B object created!");
6     }
7
8     public void SayHi(){
9       Console.WriteLine("Hi!");
10    }
11  }
```

The only thing to note in example 10.2 is the SayHi() method, which starts on line 8. It just prints a simple message to the console. Example 10.3 gives a short program called TestDriver that is used to test classes A and B and illustrate simple aggregation.

*10.3 TestDriver.cs*

```
1   public class TestDriver {
2     public static void Main(){
3       B b = new B();
4       A a = new A(b);
5       a.MakeContainedObjectSayHi();
6     }
7   }
```

Referring to example 10.3 — A B object is first created on line 3. The reference b is used as an argument to the A() constructor. In this manner an A object gets a reference to a B object. On line 5 a call is made to the MakeContainedObjectSayHi() method via reference a. The results of running this program are shown in figure 10-5.

  

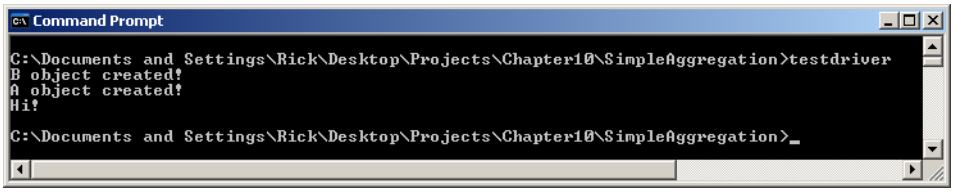


Figure 10-5: Results of Running Example 10.3

## Composite Aggregation Example

Figure 10-6 gives a UML diagram that illustrates composite aggregation between classes A and B.

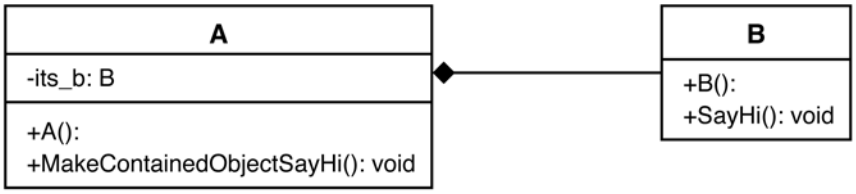


Figure 10-6: Composite Aggregation Example

Referring to figure 10-6 — Class A still has an attribute named its_b and a method named MakeContainedObjectSayHi(), however, the A() constructor has no parameters. Class B is exactly the same here as it was in the previous example. The source code for both these classes is given in examples 10.4 and 10.5.

*10.4 A.cs*

```
1   using System;
2
3   public class A {
4     private B its_b = null;
5
6     public A(){
7       its_b = new B();
8       Console.WriteLine("A object created!");
9     }
10
11    public void MakeContainedObjectSayHi(){
12      its_b.SayHi();
13    }
14  }
```

*10.5 B.cs*

```
1   using System;
2
3   public class B {
4     public B(){
5       Console.WriteLine("B object created!");
6     }
7
8     public void SayHi(){
9       Console.WriteLine("Hi!");
10    }
11  }
```

  

Referring to example 10.4 — The its_b attribute is declared and initialized to null on line 4. In the A() constructor on line 7, a new B object is created and its memory location is assigned to the its_b reference. In this manner, the A object controls the creation of the B object.

Example 10.6 gives a modified version of the TestDriver program. Compare example 10.6 to example 10.3. This version is exactly one line shorter than the previous version. This is because there's no need to create a B object before creating the A object. Figure 10-7 shows the results of running this program.

*10.6 TestDriver.cs*

```
1   public class TestDriver {
2     public static void Main(){
3       A a = new A();
4       a.MakeContainedObjectSayHi();
5     }
6   }
```

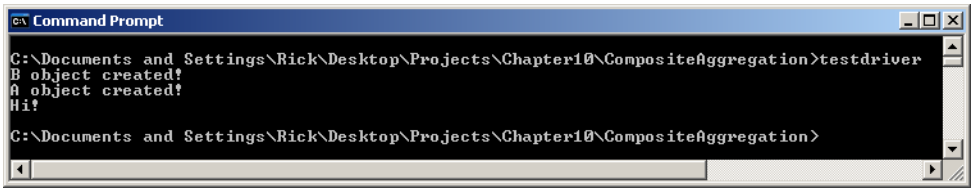


Figure 10-7: Results of Running Example 10.6

## Quick Review

The UML class diagram can be used to show aggregation associations between classes. A hollow diamond denotes simple aggregation and expresses a *uses* or *uses a* relationship between the whole and part classes. A solid diamond denotes composite aggregation and expresses a *contains* or *has a* relationship between whole and part classes.

## Sequence Diagrams

A sequence diagram is another type of UML diagram that illustrates the order or sequence of execution events that occur between objects in an application. Sequence diagrams become extremely helpful and important, especially when using compositional design. Figure 10-8 shows the sequence diagram for the simple aggregation program given in examples 10.1 through 10.3 in the previous section.

Referring to figure 10-8 — Read the sequence diagram from left to right. The objects that participate in the sequence of events appear along the top of the diagram. Each object has an object lifeline. An object can be an instance of a class or an external system (actor) that participates in the event sequence.

The User initiates the event sequence by starting the program. This is done by running the TestDriver program. The TestDriver program creates an instance of class B by calling the B() constructor method. Upon the constructor call return, the TestDriver program then creates an A object by calling the A() constructor method passing the reference b as an argument. The TestDriver then sends the MakeContainedObjectSayHi() message to the A object. (*i.e., It calls a method.*) The A object in turn sends the SayHi() message to the B object. This results in the message "Hi!" being printed to the console. After the message is printed the program terminates.

The sequence of events is a little different for the composite aggregate version of this program, as figure 10-9 illustrates.

In the composite aggregate sequence, the TestDriver program creates the A object first. The A object then creates the B object. This is shown in message numbers 2 and 3.

  

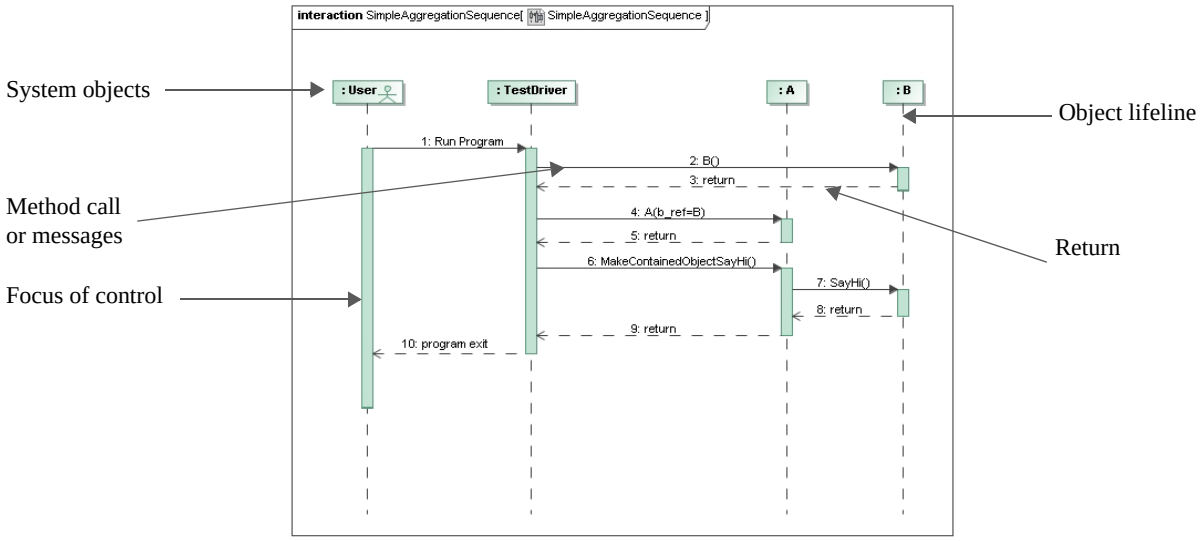


Figure 10-8: Sequence Diagram — Simple Aggregation

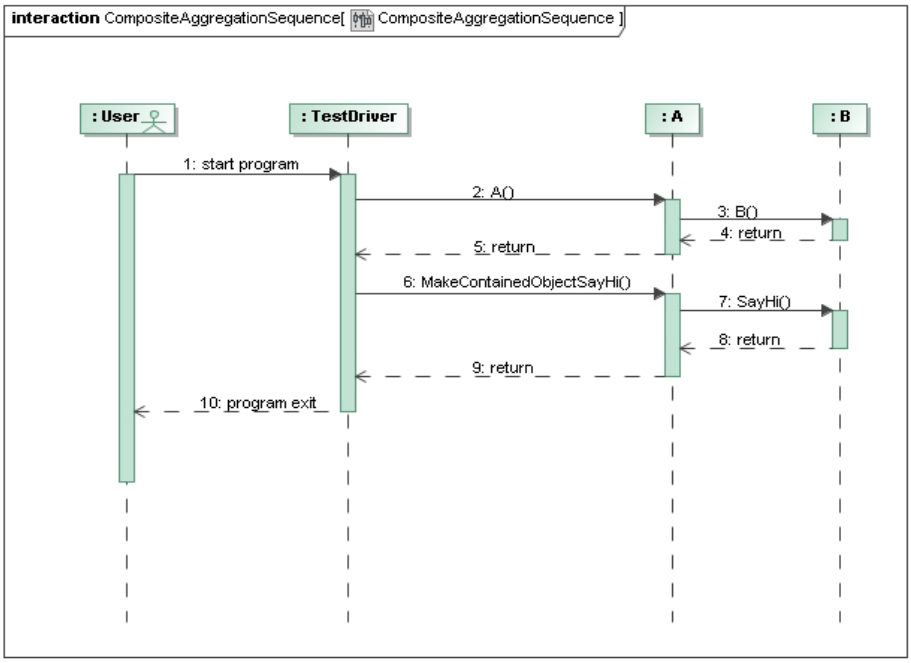


Figure 10-9: Sequence Diagram — Composite Aggregation

The sequence diagrams shown here are unique in that the example program is small enough to show the whole sequence of events in one picture. In reality, however, most software systems are so complex that sequence diagrams usually focus on one piece of functionality at a time. You'll see an example of this in the next section.

  

### Message Passing

The object-oriented term used to denote the action of calling a method or a function on an object is *message passing*. When you call a method on an object in C# you can also say that you are sending that object a message.

### Magic Draw

I used a UML design tool called Magic Draw to create the sequence diagrams in this chapter. You can get more information about Magic Draw at www.magicdraw.com

### Quick Review

UML Sequence diagrams graphically illustrate a sequence of system events. Sequence event participants can be internal system objects or external actors. Sequence diagrams do a good job of illustrating the complex message passing between objects that participate in a compositional design.

## The Engine Simulation: An Extended Example

In this section I will ramp up the complexity somewhat and present an extended example of compositional design: the engine simulation. Don't panic! It's a simple simulation designed primarily to get you comfortable with an increased level of both conceptual and physical complexity. This means there are more classes that participate in the design which means there are more source code files to create, compile, and maintain.

Figure 10-10 gives the project specification for the engine simulation. Read it carefully before proceeding to the next section.

The project specification offers good direction and several hints regarding the project requirements. The engine simulation consists of seven classes. The Engine class is the composite. An engine has a fuel pump, oil pump, compressor, oxygen sensor, and temperature sensor. These parts are represented in the design by the FuelPump, OilPump, Compressor, OxygenSensor, and TemperatureSensor classes respectively. Each of these classes has an association with the PartStatus enumeration, as the class diagram in figure 10-11 illustrates.

### The Purpose Of The Engine Class

The purpose of the Engine class is to embody the behavior of this thing we are modeling called an engine. A UML class diagram for the Engine class is given in figure 10-12.

### Engine Class Attributes And Methods

Referring to figure 10-12 — The Engine class has several attributes which are implemented as fields: *_itsCompressor*, *_itsFuelPump*, *_itsOilPump*, *_itsOxygenSensor*, and *_itsTemperatureSensor*. Each maps to its respective part class. However, several more attributes are required to complete the class. Two of these, *_itsEngineNumber* and *_isRunning*, are value type variables. The last attribute, *_itsStatus*, is an enumeration variable of type PartStatus.

As you can tell from looking at the class diagram in figure 10-12, all of the Engine class fields are declared to be private. *This is denoted by the '-' symbol preceding each attribute's name*. It has one public read-only property: EngineNumber.

  

```
                    Project Specification
                      Engine Simulation

   Objectives:
   - Apply compositional design techniques to create a C# program
   - Create a composite aggregation class whose functionality is derived from
its various part classes
   - Employ UML class and sequence diagrams to express your design ideas
   - Derive user-defined data types to model a problem domain
   - Employ inter-object message passing


   Tasks:
   - Write a program that simulates the basic functionality of an engine. The
engine should contain the following parts: fuel pump, oil pump, compressor,
temperature sensor, and oxygen sensor. Limit the functionality to the follow-
ing functions:
      • Set and check each engine part status
      • Set and check the overall engine status
      • Start the engine
      • Stop the engine


   Hints:
   - Each engine should have an assigned engine number, and each part con-
tained by the engine should register itself with the engine number.
   - When checking the engine status while the engine is running, the engine
should stop running if it detects a fault in one of its parts.
   - When an individual part's status changes, the engine must perform a com-
prehensive status check on itself.
   - If, when trying to start, the engine detects a faulty part, the engine
must not start until the status on the faulty part changes.
   - Limit the user interface to simple text messages printed to the console.
```

Figure 10-10: Engine Simulation Project Specification

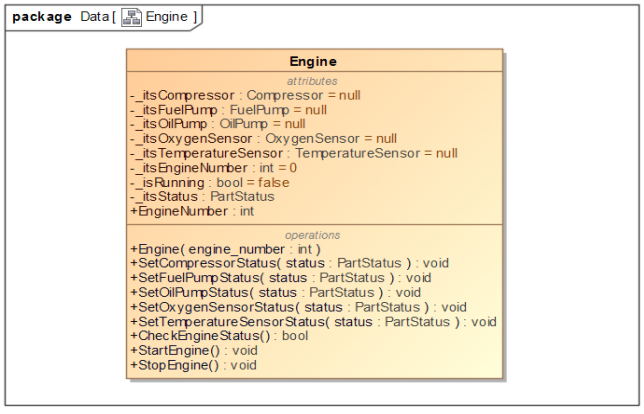


Figure 10-12: Engine Class Diagram

  

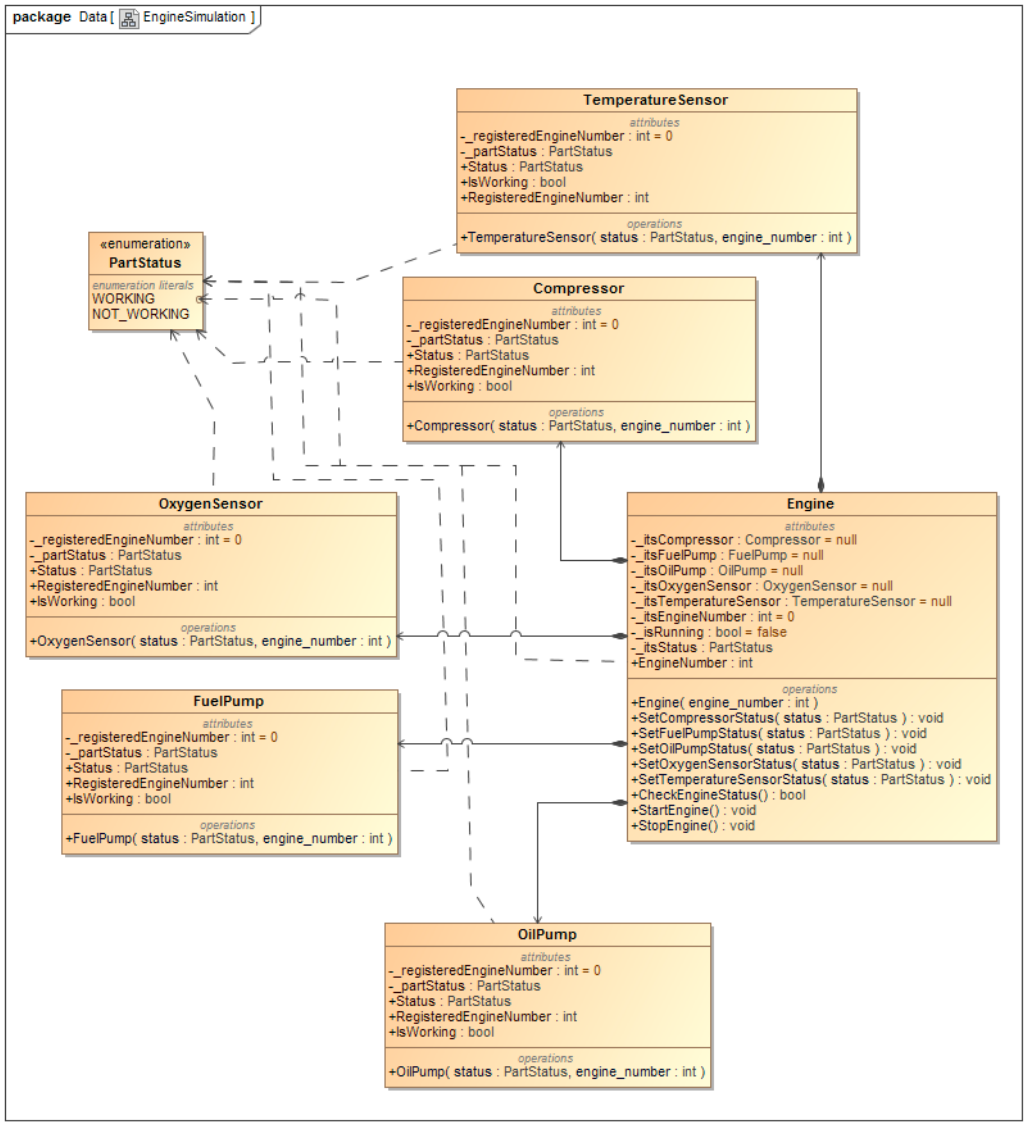


Figure 10-11: Engine Simulation Class Diagram

The design of Engine class guards against returning a reference to any of its part objects. This ensures that when a reference to an Engine object goes out of scope, the Engine object it referenced, along with all its component objects, will be collected by the garbage collector. *Although, if you remember, you cannot guarantee when this garbage collection event will occur.*

The Engine class has one constructor that takes an integer as an argument. When created, an Engine object will set *_itsEngineNumber* attribute using this number.

The remaining Engine class methods map pretty much directly to those specified or hinted at in the project specification. The Engine class interface allows you to set the status of each of an engine's component parts, check the status of an engine, and start and stop an engine.

## Engine Simulation Sequence Diagrams

The engine simulation is sufficiently complex to warrant focused sequence diagrams. Figure 10-13 gives the sequence diagram for the creation of an Engine object.

  

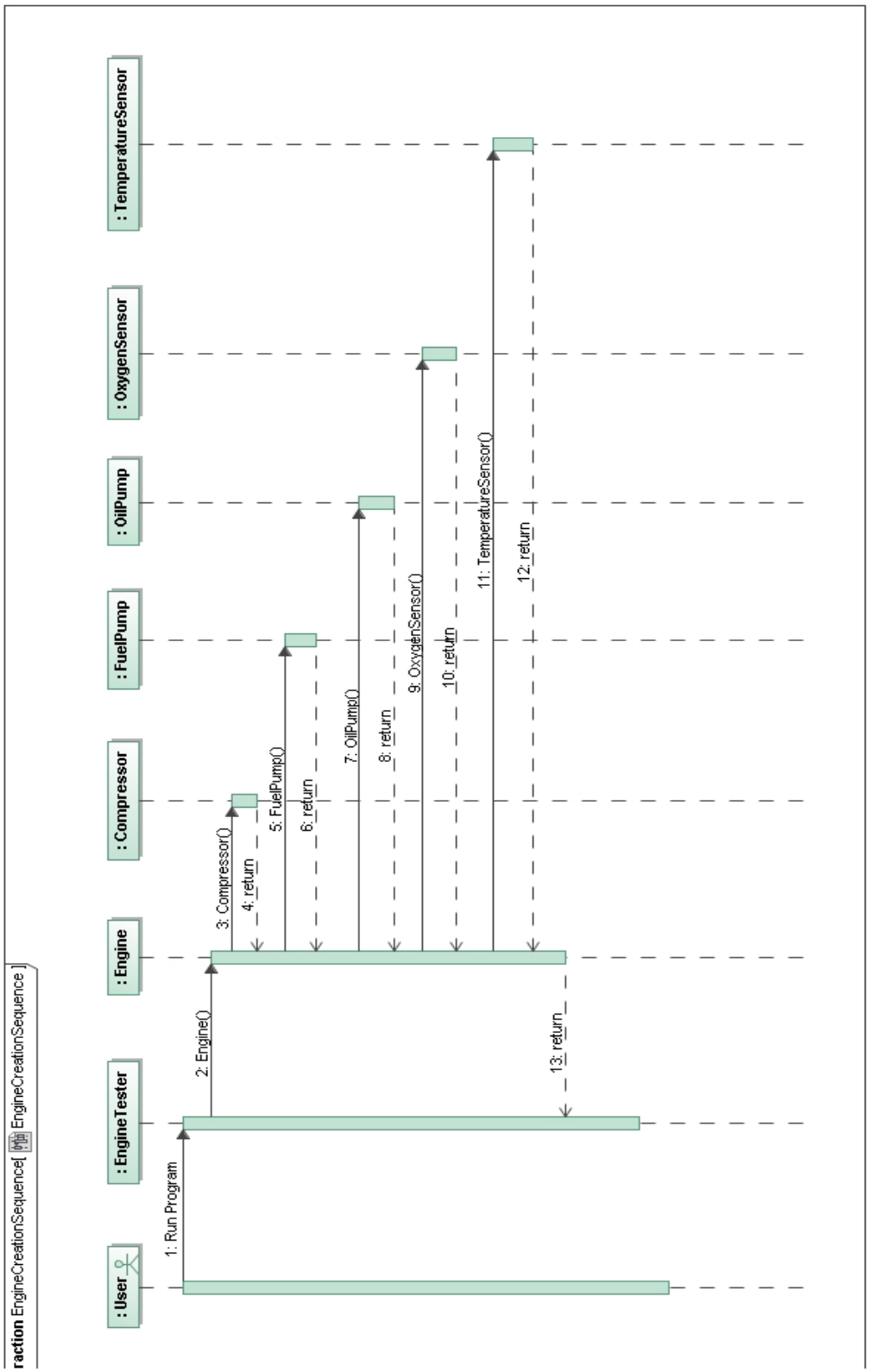


Figure 10-13: Create Engine Object Sequence

Referring to figure 10-13 — A User starts the sequence of events by running the EngineTester application. *The EngineTester class is covered in the next section*. The EngineTester application creates an Engine object by calling the Engine() constructor with an integer argument. The Engine constructor in turn creates all its required part objects. When all the part object constructor calls return, the Engine constructor call returns to the EngineTester application. At this point, the Engine object is ready for additional interface method calls from the EngineTester application.

As I said earlier, this sequence diagram represents a focused part of the simulation program. In fact, this diagram only accounts for the sequence of events resulting from the execution of line 5 of example 10.7 in the next section. The creation of additional sequence diagrams for the simulation program is left for you to do as an exercise.

## Running The Engine Simulation Program

To run the engine simulation, you'll need to create a test driver program. My version of the test driver program is a class named EngineTester, and is shown in example 10-7. To compile the engine simulation program, navigate to the project directory and run `csc *.cs` to compile all the source files together. This creates one assembly called EngineTester.exe. The result of running this program is shown in figure 10-14.

*10.7 EngineTester.cs*

```
1   using EngineSimulation;
2
3   public class EngineTester {
4     public static void Main(){
5       Engine e1 = new Engine(1);
6       e1.StartEngine();
7       e1.SetCompressorStatus(PartStatus.NOT_WORKING);
8       e1.StartEngine();
9       e1.SetCompressorStatus(PartStatus.WORKING);
10      e1.StartEngine();
11    }
12  }
```

Referring to example 10.7 — Note first that on line 1 the `using EngineSimulation` namespace statement. I'm doing this because I have placed all the engine simulation code in a namespace called EngineSimulation. (See examples 10.8 through 10.15.) Continuing with example 10.7, an Engine object is created on line 5. As noted above, this line kicks off the sequence of events illustrated in figure 10-13. All of the Engine's part objects are initially created with WORKING part status. The StartEngine() method call on line 6 initiates a CheckEngineStatus() method call. Refer to the Engine class code listing in the next section. If all goes well, the engine reports that it is running.

On line 7, a fault is introduced to the engine's compressor component. This causes the engine to shut down. The attempt on line 8 to start the engine fails due to the failed compressor. When the compressor fault is removed, the engine starts fine. This sequence of events can be traced through the source code and by carefully reading the program's output shown in figure 10-14.

## Quick Review

The Engine class is a composite aggregate. Its behavior is dictated by the behavior of its contained part class objects. Its part classes include Compressor, FuelPump, OilPump, OxygenSensor, and TemperatureSensor. All the classes in the simulation use the functionality of the PartStatus enumeration.

  

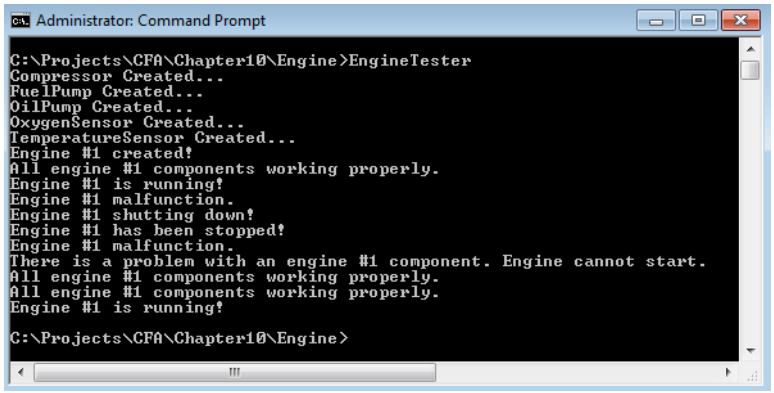


Figure 10-14: Result of Running Example 10.7

## Complete Engine Simulation Code Listing

*10.8 Engine.cs*

```
1   using System;
2
3   namespace EngineSimulation {
4
5     public class Engine {
6       private Compressor _itsCompressor = null;
7       private FuelPump _itsFuelPump = null;
8       private OilPump _itsOilPump = null;
9       private OxygenSensor _itsOxygenSensor = null;
10      private TemperatureSensor _itsTemperatureSensor = null;
11      private int _itsEngineNumber = 0;
12      private bool _isRunning = false;
13      private PartStatus _itsStatus;
14
15      public int EngineNumber {
16        get { return _itsEngineNumber; }
17      }
18
19      public Engine(int engine_number) {
20        _itsEngineNumber = engine_number;
21        _itsCompressor = new Compressor(PartStatus.WORKING, _itsEngineNumber);
22        _itsFuelPump = new FuelPump(PartStatus.WORKING, _itsEngineNumber);
23        _itsOilPump = new OilPump(PartStatus.WORKING, _itsEngineNumber);
24        _itsOxygenSensor = new OxygenSensor(PartStatus.WORKING, _itsEngineNumber);
25        _itsTemperatureSensor = new TemperatureSensor(PartStatus.WORKING, _itsEngineNumber);
26        _itsStatus = PartStatus.WORKING;
27        Console.WriteLine("Engine #" + _itsEngineNumber + " created!");
28      }
29
30      public void SetCompressorStatus(PartStatus status) {
31        _itsCompressor.Status = status;
32        CheckEngineStatus();
33      }
34
35      public void SetFuelPumpStatus(PartStatus status) {
36        _itsFuelPump.Status = status;
37        CheckEngineStatus();
38      }
39
40      public void SetOilPumpStatus(PartStatus status) {
```

  

```
41      _itsOilPump.Status = status;
42      CheckEngineStatus();
43    }
44
45    public void SetOxygenSensorStatus(PartStatus status) {
46      _itsOxygenSensor.Status = status;
47      CheckEngineStatus();
48    }
49
50    public void SetTemperatureSensor(PartStatus status) {
51      _itsTemperatureSensor.Status = status;
52      CheckEngineStatus();
53    }
54
55    public bool CheckEngineStatus() {
56      if (_itsCompressor.IsWorking && _itsFuelPump.IsWorking &&
57        _itsOilPump.IsWorking && _itsOxygenSensor.IsWorking &&
58        _itsTemperatureSensor.IsWorking) {
59        _itsStatus = PartStatus.WORKING;
60        Console.WriteLine("All engine #" + _itsEngineNumber + " components working properly.");
61      } else {
62          _itsStatus = PartStatus.NOT_WORKING;
63          Console.WriteLine("Engine #" + _itsEngineNumber + " malfunction.");
64          if (_isRunning) {
65            Console.WriteLine("Engine #" + _itsEngineNumber + " shutting down!");
66            StopEngine();
67          }
68        }
69      return (_itsStatus == PartStatus.WORKING);
70    }
71
72    public void StartEngine() {
73      if (!_isRunning) {
74        if (CheckEngineStatus()) {
75          _isRunning = true;
76          Console.WriteLine("Engine #" + _itsEngineNumber + " is running!");
77        } else {
78            Console.WriteLine("There is a problem with an engine #" + _itsEngineNumber
79                              + " component. Engine cannot start.");
80        }
81      } else {
82          Console.WriteLine("Engine #" + _itsEngineNumber + " is already running!");
83        }
84    }
85
86    public void StopEngine() {
87      _isRunning = false;
88      Console.WriteLine("Engine #" + _itsEngineNumber + " has been stopped!");
89    }
90  } // end class definition
91 } // end namespace
```

*10.9 Compressor.cs*

```
1  using System;
2
3  namespace EngineSimulation {
4
5    public class Compressor {
6      private int _registeredEngineNumber = 0;
7      private PartStatus _partStatus;
8
9      public PartStatus Status {
10       get { return _partStatus; }
11       set { _partStatus = value; }
```

  

```
12     }
13
14     public int RegisteredEngineNumber {
15       get { return _registeredEngineNumber; }
16       set { _registeredEngineNumber = value; }
17     }
18
19     public bool IsWorking {
20       get {
21         if (Status == PartStatus.WORKING) {
22           return true;
23         }
24           return false;
25         }
26     }
27
28     public Compressor(PartStatus status, int engine_number) {
29       RegisteredEngineNumber = engine_number;
30       Status = status;
31       Console.WriteLine("Compressor Created...");
32     }
33   } // end class definition
34 } // end namespace
```

*10.10 FuelPump.cs*

```
1  using System;
2
3  namespace EngineSimulation {
4
5    public class FuelPump {
6      private int _registeredEngineNumber = 0;
7      private PartStatus _partStatus;
8
9      public PartStatus Status {
10       get { return _partStatus; }
11       set { _partStatus = value; }
12     }
13
14     public int RegisteredEngineNumber {
15       get { return _registeredEngineNumber; }
16       set { _registeredEngineNumber = value; }
17     }
18
19     public bool IsWorking {
20       get {
21         if (Status == PartStatus.WORKING) {
22           return true;
23         }
24         return false;
25       }
26     }
27
28     public FuelPump(PartStatus status, int engine_number) {
29       RegisteredEngineNumber = engine_number;
30       Status = status;
31       Console.WriteLine("FuelPump Created...");
32     }
33   } // end class definition
34 } // end namespace
```

  

*10.11 OilPump.cs*

```
using System;

namespace EngineSimulation {

  public class OilPump {
    private int _registeredEngineNumber = 0;
    private PartStatus _partStatus;

    public PartStatus Status {
      get { return _partStatus; }
      set { _partStatus = value; }
    }

    public int RegisteredEngineNumber {
      get { return _registeredEngineNumber; }
      set { _registeredEngineNumber = value; }
    }

    public bool IsWorking {
      get {
        if (Status == PartStatus.WORKING) {
          return true;
        }
        return false;
      }
    }

    public OilPump(PartStatus status, int engine_number) {
      RegisteredEngineNumber = engine_number;
      Status = status;
      Console.WriteLine("OilPump Created...");
    }
  } // end class definition
} // end namespace
```

*10.12 OxygenSensor.cs*

```
using System;

namespace EngineSimulation {

  public class OxygenSensor {
    private int _registeredEngineNumber = 0;
    private PartStatus _partStatus;

    public PartStatus Status {
      get { return _partStatus; }
      set { _partStatus = value; }
    }

    public int RegisteredEngineNumber {
      get { return _registeredEngineNumber; }
      set { _registeredEngineNumber = value; }
    }

    public bool IsWorking {
      get {
        if (Status == PartStatus.WORKING) {
          return true;
        }
        return false;
      }
    }

```

  

```
28     public OxygenSensor(PartStatus status, int engine_number) {
29       RegisteredEngineNumber = engine_number;
30       Status = status;
31       Console.WriteLine("OxygenSensor Created...");
32     }
33   } // end class definition
34 } // end namespace
```

*10.13 TemperatureSensor.cs*

```
1  using System;
2
3  namespace EngineSimulation {
4
5    public class TemperatureSensor {
6      private int _registeredEngineNumber = 0;
7      private PartStatus _partStatus;
8
9      public PartStatus Status {
10       get { return _partStatus; }
11       set { _partStatus = value; }
12     }
13
14     public bool IsWorking {
15       get {
16         if (Status == PartStatus.WORKING) {
17           return true;
18         }
19         return false;
20       }
21     }
22
23     public int RegisteredEngineNumber {
24       get { return _registeredEngineNumber; }
25       set { _registeredEngineNumber = value; }
26     }
27
28     public TemperatureSensor(PartStatus status, int engine_number) {
29       RegisteredEngineNumber = engine_number;
30       Status = status;
31       Console.WriteLine("TemperatureSensor Created...");
32     }
33   } // end class definition
34 } // end namespace
```

*10.14 PartStatus.cs*

```
1  using System;
2
3  namespace EngineSimulation {
4
5    public enum PartStatus { WORKING, NOT_WORKING }
6
7  }
```

*10.15 EngineTester.c*

```
1  using EngineSimulation;
2
3  public class EngineTester {
4    public static void Main(){
5      Engine e1 = new Engine(1);
6      e1.StartEngine();
7      e1.SetCompressorStatus(PartStatus.NOT_WORKING);
8      e1.StartEngine();
9      e1.SetCompressorStatus(PartStatus.WORKING);
```

  

```
10      e1.StartEngine();
11    }
12  }
```

## The Managed Extensibility Framework (MEF)

The engine simulation presented in the previous section is an example of a role-your-own composite application. Its design, while it does illustrate the concepts of compositional design, is fairly inflexible in that to change the behavior of a part requires modifying the part code and recompiling the entire application. A more flexible approach would implement a plug-in architecture which would allow parts to be updated more easily.

Creating a viable plug-in architecture is a lot of work and requires deeper understanding of interfaces and inheritance, two topics not covered until chapter 11. As it happens, Microsoft supplies us with a composition library known as the *Managed Extensibility Framework* or MEF that we can use now to greatly simplify the engine simulation application while at the same time making it more flexible. Later, when you are exposed to interfaces and inheritance, you can use the MEF to create truly powerful, extensible applications.

This section provides a breathless introduction to the compositional capabilities of the MEF using a re-implementation of the engine simulation as the primary example. To understand the MEF requires understanding some key terms. First, think of everything as a part. A part can be comprised of other parts. A composite part, meaning a part that has dependencies on other parts, like the engine in the previous section, specifies one or more dependencies via `[Import]` attributes. A part can also advertise capability via one or more `[Export]` attributes. A composite part can be told where to look for parts whose Exports match its Imports. This will become clear when you examine the source code in the following sections.

### EngineSimulationMEF Solution

I've structured this project differently from previous examples. First, I used Visual Studio to create a solution called EngineSimulationMEF and three sub-projects: EngineParts, Engines, and EngineTester. EngineParts and Engines are *class library* project types, meaning they generate separate dynamic-linked library files or DLLs. Even though this example is conceptually less complex than the previous engine simulation example, it has greater physical complexity because of the use of the multi-project solution. Visual Studio greatly simplifies project management of physically complex solutions.

Second, the EngineParts project contains the code for all the engine parts, with the exception of the Engine class itself, which appears in the Engines project. I also decided to place the PartStatus enumeration within the EngineParts project as well. This has design ramifications and I could have created a separate project just for the PartStatus enumeration alone which could be shared by all the other projects, but the approach I take here forced the issue of type translation between library boundaries, which means using a type found in all libraries to translate between a type found only in one library to minimize inter-component dependencies.

Lastly, the Engine class looks for parts that advertise capability with an `[Export]` to satisfy its `[Import]` requirements in the working directory, which means the directory in which the EngineTester application executes. If you execute the code using Visual Studio, the working directory will be either `EngineSimulationMEF/EngineTester/bin/Debug` or `../Release`, depending on the solution's build configuration.

You can download the Visual Studio solution from the GitHub repository: [ https://github.com/pulp-freepress/CSharpForArtists3rdEdition/tree/master/Chapter10 ]

  

### EngineParts Project (Assembly Name: EngineParts, Output Type: Class Library)

The EngineParts project (and namespace), contains the following components: *PartStatus*, *OilPump*, *FuelPump*, *Compressor*, *TemperatureSensor*, and *OxygenSensor*. I chose for this example to put all the parts, with the exception of the Engine, in one library. (**NOTE:** *I could have placed each part in its own library for maximum flexibility, with the burden of additional physical complexity. I'll take this approach in chapter 11*.) Since all the parts share a similar structure and interface, I'll discuss only the OilPump class in detail.

*10.16 PartStatusEnum.cs*

```
1  namespace EngineParts {
2    public enum PartStatus { NOT_WORKING, WORKING }
3  }
```

Referring to example 10.16 — This serves the same function as PartStatus in the previous example, but I did reverse the values from `{ WORKING, NOT_WORKING }`, to `{ NOT_WORKING, WORKING }`. I did this for personal aesthetic reasons. I just couldn't sleep knowing WORKING was 0 and NOT_WORKING was 1, though enumerations let us abstract away such details. It would work fine either way, actually.

*10.17 OilPump.cs*

```
1  using System;
2  using System.ComponentModel.Composition;
3
4  namespace EngineParts {
5    [Export]
6    public class OilPump {
7      public int EngineNumber {
8        get;
9        set;
10     }
11
12     public PartStatus Status {
13       get;
14       set;
15     } = PartStatus.WORKING;
16
17     public bool IsWorking {
18       get {
19         return Status == PartStatus.WORKING ? true : false;
20       }
21       set {
22         if (value) Status = PartStatus.WORKING;
23         else Status = PartStatus.NOT_WORKING;
24       }
25     }
26
27     [ImportingConstructor]
28     public OilPump(int engine_number) {
29       EngineNumber = engine_number;
30       Console.WriteLine(this.GetType().Name + " Created!");
31     }
32   }
33 }
```

Referring to example 10.17 — All the parts in the EngineParts project have the same essential structure. First, line 2 adds a using directive for the System.ComponentModel.Composition namespace. Second, an [Export] attribute is placed on line 5 above the class definition. This essentially indicates the entire OilPump type is exported. Third, the Status property starting on line 12 is a read-write auto-property and is initialized on line 15 to PartStatus.WORKING. Next, the IsWorking property is of type bool. It, too, is a read-write property. When read, it checks the Status property and if it's set to PartStatus.WORKING, it returns true, otherwise, it returns false. When set, or written, it translates the incoming bool value to the appropriate internal PartStatus setting and set's the part's Status property accordingly. Finally, the [ImportingConstructor] attribute is placed above the constructor to signal this is the constructor to use to initialize

  

an OilPump. If the [ImportingConstructor] attribute were left out, MEF would expect a default, no-argument constructor.

*10.18 FuelPump.cs*

```
1   using System;
2   using System.ComponentModel.Composition;
3
4   namespace EngineParts {
5     [Export]
6     public class FuelPump {
7       public int EngineNumber {
8         get;
9         set;
10      }
11
12      public PartStatus Status {
13        get;
14        set;
15      } = PartStatus.WORKING;
16
17      public bool IsWorking {
18        get {
19          return Status == PartStatus.WORKING ? true : false;
20        }
21        set {
22          if (value) Status = PartStatus.WORKING;
23          else Status = PartStatus.NOT_WORKING;
24        }
25      }
26
27      [ImportingConstructor]
28      public FuelPump(int engine_number) {
29        EngineNumber = engine_number;
30        Console.WriteLine(this.GetType().Name + " Created!");
31      }
32    }
33  }
```

*10.19 Compressor.cs*

```
1   using System;
2   using System.ComponentModel.Composition;
3
4   namespace EngineParts {
5     [Export]
6     public class Compressor {
7       public int EngineNumber {
8         get;
9         set;
10      }
11
12      public PartStatus Status {
13        get;
14        set;
15      } = PartStatus.WORKING;
16
17      public bool IsWorking {
18        get {
19          return Status == PartStatus.WORKING ? true : false;
20        }
21        set {
22          if (value) Status = PartStatus.WORKING;
23          else Status = PartStatus.NOT_WORKING;
24        }
25      }
```

  

```
26
27      [ImportingConstructor]
28      public Compressor(int engine_number) {
29        EngineNumber = engine_number;
30        Console.WriteLine(this.GetType().Name + " Created!");
31      }
32    }
33  }
```

*10.20 TemperatureSensor.cs*

```
1   using System;
2   using System.ComponentModel.Composition;
3
4   namespace EngineParts {
5     [Export]
6     public class TemperatureSensor {
7       public int EngineNumber {
8         get;
9         set;
10      }
11
12      public PartStatus Status {
13        get;
14        set;
15      } = PartStatus.WORKING;
16
17      public bool IsWorking {
18        get {
19          return Status == PartStatus.WORKING ? true : false;
20        }
21        set {
22          if (value) Status = PartStatus.WORKING;
23          else Status = PartStatus.NOT_WORKING;
24        }
25      }
26
27      [ImportingConstructor]
28      public TemperatureSensor(int engine_number) {
29        EngineNumber = engine_number;
30        Console.WriteLine(this.GetType().Name + " Created!");
31      }
32    }
33  }
```

*10.21 OxygenSensor.cs*

```
1   using System;
2   using System.ComponentModel.Composition;
3
4   namespace EngineParts {
5     [Export]
6     public class OxygenSensor {
7       public int EngineNumber {
8         get;
9         set;
10      }
11
12      public PartStatus Status {
13        get;
14        set;
15      } = PartStatus.WORKING;
16
17      public bool IsWorking {
18        get {
19          return Status == PartStatus.WORKING ? true : false;
```

  

```
20       }
21       set {
22         if (value) Status = PartStatus.WORKING;
23         else Status = PartStatus.NOT_WORKING;
24       }
25     }
26
27     [ImportingConstructor]
28     public OxygenSensor(int engine_number) {
29       EngineNumber = engine_number;
30       Console.WriteLine(this.GetType().Name + " Created!");
31     }
32   }
33 }
```

### *Engines Project (Assembly Name: Engines, Output Type: Class Library)*

The Engines project (and namespace) contains the Engine class. The Engine class serves as a composite part, meaning it declares [Import]s. The Engine itself is declared as an [Export], meaning some other part, let's say, an automobile, or aeroplane, could [Import] a property of type Engine.

*10.22 Engine.cs*

```
1   using EngineParts;
2   using System;
3   using System.ComponentModel.Composition;
4   using System.ComponentModel.Composition.Hosting;
5   using System.Text;
6
7
8   namespace Engines {
9     [Export]
10    public class Engine {
11      /***** Parts *****/
12      [Import]
13      public OilPump _oilPump;
14
15      [Import]
16      private FuelPump _fuelPump;
17
18      [Import]
19      private Compressor _compressor;
20
21      [Import]
22      private OxygenSensor _oxygenSensor;
23
24      [Import]
25      private TemperatureSensor _temperatureSensor;
26
27      private CompositionContainer _container;
28
29      /***** Properties *****/
30      [Export] // Must be exported because it's used as a parameter to part constructors
31      public int EngineNumber {
32        get;
33        set;
34      }
35
36      public bool IsRunning {
37        get;
38        set;
39      } = false;
40
41      public bool IsWorking {
```

  

```
42        get {
43          return _compressor.IsWorking && _oilPump.IsWorking && _fuelPump.IsWorking
44             && _temperatureSensor.IsWorking && _oxygenSensor.IsWorking;
45        }
46      }
47
48      public int OilPumpEngineNumber {
49        get { return _oilPump.EngineNumber; }
50      }
51
52      public int FuelPumpEngineNumber {
53        get { return _fuelPump.EngineNumber; }
54      }
55
56      public int CompressorEngineNumber {
57        get { return _compressor.EngineNumber; }
58      }
59
60      public int TemperatureSensorEngineNumber {
61        get { return _temperatureSensor.EngineNumber; }
62      }
63
64      public int OxygenSensorEngineNumber {
65        get { return _oxygenSensor.EngineNumber; }
66      }
67
68      public bool IsOilPumpWorking {
69        get { return _oilPump.IsWorking; }
70        set {
71          _oilPump.IsWorking = value;
72          if (!IsWorking) StopEngine();
73        }
74      }
75
76      public bool IsFuelPumpWorking {
77        get { return _fuelPump.IsWorking; }
78        set {
79          _fuelPump.IsWorking = value;
80          if (!IsWorking) StopEngine();
81        }
82      }
83
84      public bool IsCompressorWorking {
85        get { return _compressor.IsWorking; }
86        set {
87          _compressor.IsWorking = value;
88          if (!IsWorking) StopEngine();
89        }
90      }
91
92      public bool IsTemperatureSensorWorking {
93        get { return _temperatureSensor.IsWorking; }
94        set {
95          _temperatureSensor.IsWorking = value;
96          if (!IsWorking) StopEngine();
97        }
98      }
99
100     public bool IsOxygenSensorWorking {
101       get { return _oxygenSensor.IsWorking; }
102       set {
103         _oxygenSensor.IsWorking = value;
104         if (!IsWorking) StopEngine();
105       }
106     }
```

  

```
107
108     /***** Constructor *****/
109     [ImportingConstructor]
110     public Engine(int engine_number) {
111       EngineNumber = engine_number;
112       // Check runtime directory for DLLs that contain parts with matching imports
113       var catalog = new DirectoryCatalog(".");
114       // Create the catalog
115       _container = new CompositionContainer(catalog);
116
117       try {
118         this._container.ComposeParts(this);
119       } catch (Exception e) {
120           Console.WriteLine(e);
121       }
122     } // end constructor
123
124
125
126     public string[] CheckEngine() {
127       StringBuilder status_messages = new StringBuilder();
128       if (_oilPump.IsWorking)
129         status_messages.Append("OilPump -> Working,");
130       else status_messages.Append("OilPump -> Falty,");
131
132       if (_fuelPump.IsWorking)
133         status_messages.Append("FuelPump -> Working,");
134       else status_messages.Append("FuelPump -> Falty,");
135
136       if (_compressor.IsWorking)
137         status_messages.Append("Compressor -> Working,");
138       else status_messages.Append("Compressor -> Falty,");
139
140       if (_temperatureSensor.IsWorking)
141         status_messages.Append("TemperatureSensor -> Working,");
142       else status_messages.Append("TemperatureSensor -> Falty,");
143
144       if (_oxygenSensor.IsWorking)
145         status_messages.Append("OxygenSensor -> Working");
146       else status_messages.Append("OxygenSensor -> Falty");
147
148       return status_messages.ToString().Split(',');
149     }
150
151
152     public void StartEngine() {
153       if (!IsRunning) {
154         Console.WriteLine("Checking Engine No. {0} ", EngineNumber);
155         if (IsWorking) {
156           Console.WriteLine("Starting Engine No. {0} ", EngineNumber);
157           IsRunning = true;
158           Console.WriteLine("Engine No. {0} is running!", EngineNumber);
159         } else {
160           Console.WriteLine("Engine No. {0} has a problem...", EngineNumber);
161           foreach (string s in CheckEngine()) {
162             Console.WriteLine(s);
163           }
164         }
165       } else {
166         Console.WriteLine("Engine No. {0} is already running!", EngineNumber);
167       }
168     } // end StartEngine()
169
170
171     public void StopEngine() {
```

  

```
172      IsRunning = false;
173      Console.WriteLine("Engine No. {0} is shut down.", EngineNumber);
174    }
175
176  }
177 }
```

Referring to example 10.22 — Note on lines 3 and 4 the using directive for both the System.ComponentModel.Composition and System.ComponentModel.Composition.Hosting namespaces. The [Export] attribute is added above the Engine class definition on line 9, and [Import] attributes are added above each engine part property. So, the Engine will import components of type OilPump, FuelPump, Compressor, TemperatureSensor, and OxygenSensor.

On line 27 a private field of type CompositionContainer is declared. This field is used later in the Engine() constructor.

On line 30 an [Export] attribute is applied to the EngineNumber property. This is required because EngineNumber is used to supply a parameter to engine part constructors.

The [ImportingConstructor] attribute is applied to the Engine() constructor. Within the Engine() constructor a DirectoryCatalog is created and points to the current working directory (i.e., "."). This means that when an Engine object is created, the working directory from which the application was launched will be searched for any DLLs that contain types that [Export] functionality expected by Engine's corresponding [Import] attributes. Thus, the process of composition takes place automagically. Honestly, this is pretty cool.

### *EngineTester Project (Assembly Name: EngineTester, Output Type: Console Application)*

The EngineTester project contains the main application class called MainApp.

*10.23 MainApp.cs*

```
1  using Engines;
2  using System;
3
4
5  namespace EngineTester {
6    public class MainApp {
7      private static void Main(string[] args) {
8        Console.WriteLine("Instantiating Engine Object...");
9        Engine e1 = new Engine(1);
10       Console.WriteLine("-----------------------------------------------------");
11       Console.WriteLine("Reading Part Engine Numbers...");
12       Console.WriteLine("OilPump Engine Number: {0}", e1.OilPumpEngineNumber);
13       Console.WriteLine("FuelPump Engine Number: {0}", e1.FuelPumpEngineNumber);
14       Console.WriteLine("Compressor Engine Number: {0}", e1.CompressorEngineNumber);
15       Console.WriteLine("TemperatureSensor Engine Number: {0}",
16         e1.TemperatureSensorEngineNumber);
17       Console.WriteLine("OxygenSensor Engine Number: {0}", e1.OxygenSensorEngineNumber);
18       Console.WriteLine("-----------------------------------------------------");
19
20       Console.WriteLine("\nPress Enter to Start Engine Number: {0}", e1.EngineNumber);
21       Console.ReadKey();
22
23       if (e1.IsWorking) e1.StartEngine();
24
25       Console.WriteLine("\nPress Enter to Set FuelPump Fault...");
26       Console.ReadKey();
27
28       e1.IsFuelPumpWorking = false;
29
30       Console.WriteLine("\nPress Enter to Try to Start Engine Number: "
31         + "{0} with Faulty FuelPump", e1.EngineNumber);
32       Console.ReadKey();
33
```

  

```
34        e1.StartEngine();
35
36        Console.WriteLine("\nPress Enter to Fix FuelPump and Restart Engine...");
37        Console.ReadKey();
38
39        e1.IsFuelPumpWorking = true;
40        e1.StartEngine();
41
42        Console.WriteLine("\nPress Enter to Stop Engine...");
43        Console.ReadKey();
44
45        e1.StopEngine();
46
47        Console.WriteLine("\n\nPress Enter to Exit the Engine Demo...");
48        Console.ReadKey();
49      }
50    }
51  }
```

Referring to example 10.23 — In this example, I instantiate an object of type Engine manually on line 9. When created, the Engine object scans the working directory for any and all DLLs that may contain parts that fulfill its [Import] requirements, and composes itself. The rest of the program tests some of Engine's functionality, namely, setting a fault in a part and starting and stopping the engine with and without a part fault.

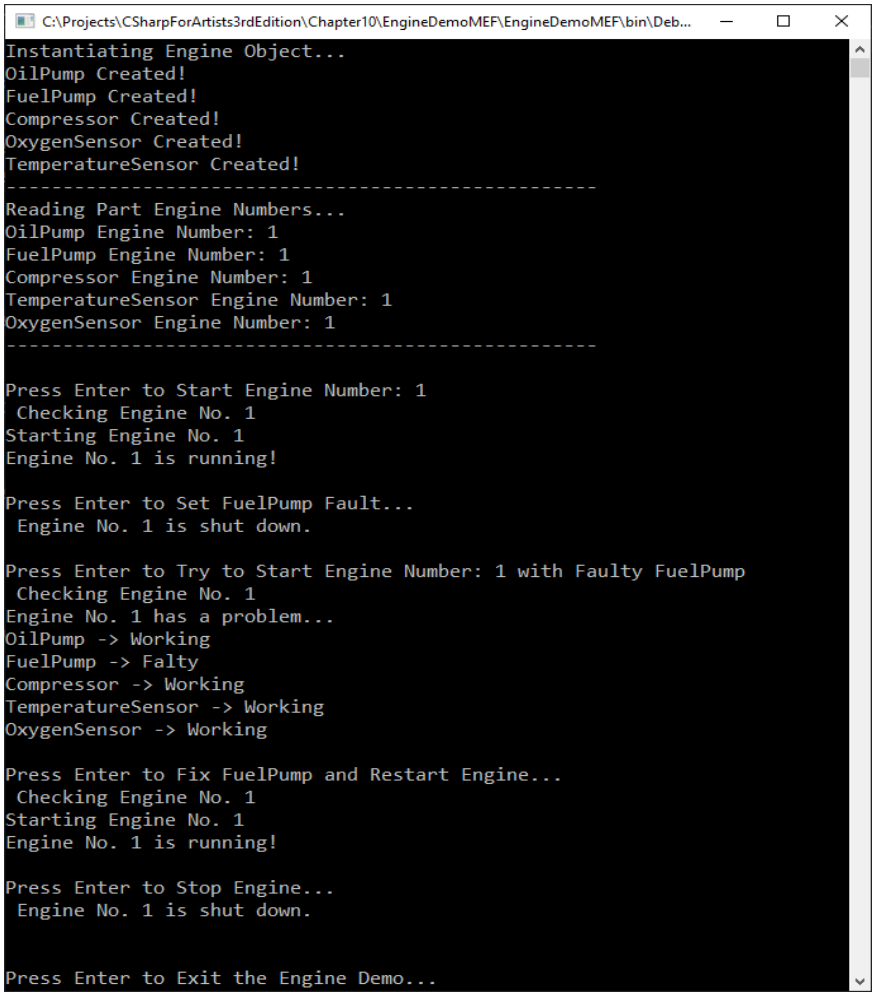


Figure 10-15: Results of Running Example 10.23

  

### Retrospective

The MEF takes the drudgery out of compositional design but still, there's room for improvement. For example, the Engine class specifies imports of specific types, namely, OilPump, FuelPump, Compressor, OxygenSensor, and TemperatureSensor. Thus, even if we have improved the flexibility of the application from a compositional design perspective, there still remain hard dependencies between the Engine class and its part object types. We can break these dependencies with the use of interfaces, a topic I will cover in chapter 11.

### Quick Review

The Managed Extensibility Framework (MEF) is a Microsoft library available since version 4 of the ,NET Framework. The MEF enables automatic application composition by specifying a composite application's expected Imports using the [Import] attribute. Part classes that can fill those expected Imports signal their capabilities by specifying Exports using the [Export] attribute. Use the [ImportingConstructor] attribute to indicate which constructor should be called when instantiating the part.

## Summary

There are two types of complexity: *conceptual* and *physical*. Conceptual complexity is managed by using object-oriented concepts and expressing your object-oriented designs in UML. Physical complexity can be managed by your IDE or with an automated build tool such as Microsoft Build (MSBuild). You can also manage physical complexity on a small scale by organizing your source files into project directories and by compiling multiple files simultaneously using the `csc` compiler tool.

A *dependency* is a relationship between two classes in which a change to the depended-upon class will affect the dependent class. An *association* is a peer-to-peer structural link between two classes.

An *aggregation* is a special type of association between two objects that results in a whole/part relationship. There are two types of aggregation: *simple* and *composite*. The type of aggregation is determined by the extent to which the whole object controls the lifetime of its part objects. If the whole object simply uses the services of a part object but does not control its lifetime, then it's a simple aggregation. On the other hand, if the whole object creates and controls the lifetime of its part objects, then it is a composite aggregate.

A UML class diagram can be used to show aggregation associations between classes. A hollow diamond denotes simple aggregation and expresses a *uses* or *uses a* relationship between the whole and part classes. A solid diamond denotes composite aggregation and expresses a *contains* or *has a* relationship between whole and part classes.

Sequence diagrams are a type of UML diagram used to graphically illustrate a sequence of system events. Sequence event participants can be internal system objects or external actors. Sequence diagrams do a good job of illustrating the complex message passing between objects that participate in a compositional design.

The Engine class is a composite aggregate. Its behavior is dictated by the behavior of its contained part class objects. Its part classes include Compressor, FuelPump, OilPump, OxygenSensor, and TemperatureSensor. All classes in the engine simulation use the PartStatus enumeration.

The Managed Extensibility Framework (MEF) is a Microsoft library available since version 4 of the ,NET Framework. The MEF enables automatic application composition by specifying a composite application's expected Imports using the [Import] attribute. Part classes that can fill those expected Imports signal their capabilities by specifying Exports using the [Export] attribute. Use the [ImportingConstructor] attribute to indicate which constructor should be called when instantiating the part.

  

## Skill-Building Exercises

1. **Study The UML:** Obtain a reference on UML and conduct further research on how to model complex associations and aggregations using class diagrams. Also study how to express complex event sequences using sequence diagrams.

2. **Obtain a UML Modeling Tool:** If you haven't already done so, procure a UML modeling tool to help you draw class and sequence diagrams.

3. **Discover Dependency Relationships:** Study the .NET Framework API. Write down at least five instances in which one class depends on the services of another class. Describe the nature of the dependency and explain the possible effects that changing the depended-upon class might have on the dependent class.

4. **Programming Exercise — Demonstrate Simple Aggregation:** Write a program that implements the simple aggregation shown in figure 10-15.

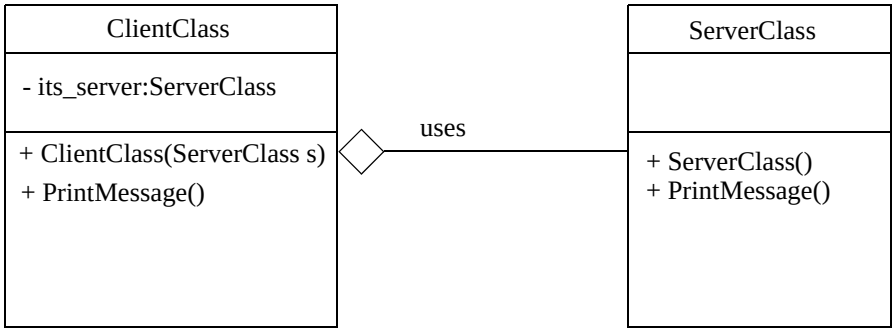


Figure 10-16: Simple Aggregation Class Diagram

**Hint:** Implement the PrintMessage() method in ServerClass to print a short message of your choosing to the console. You'll need to write a test driver program to test your code. Study examples 10.1 through 10.3 for clues on how to implement this exercise.

5. **Programming Exercise — Demonstrate Composite Aggregation:** Write a program that implements the composite aggregation shown in figure 10-16.

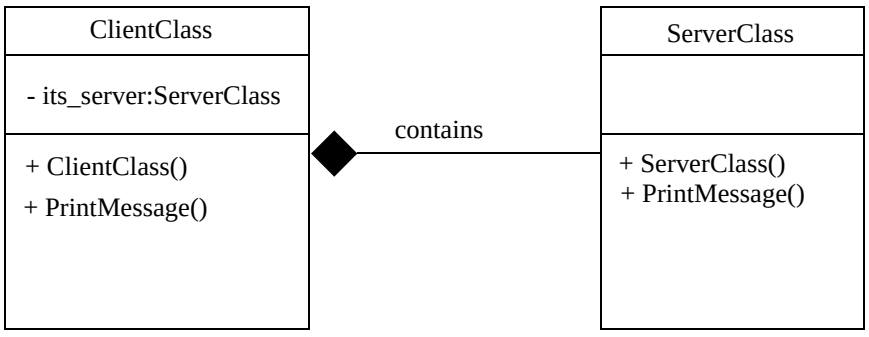


Figure 10-17: Composite Aggregation Class Diagram

  

**Hint:** Implement the PrintMessage() method in ServerClass to print a short message of your choosing to the console. You'll need to write a test driver program to test your code. Study examples 10.4 through 10.6 for clues on how to implement this exercise.

6. **Compile Multiple Source Files:** Find the engine simulator project source code presented in this chapter on the PulpFreePress C# For Artists' SupportSite®. http://www.pulpfreepress.com Use the `csc` compiler tool to compile all the source files in the project directory using the '*' wildcard character.

7. **Managed Extensibility Framework (MEF):** Research the MEF online. Start with Microsoft's documentation: [ https://docs.microsoft.com/en-us/dotnet/framework/mef/ ]

## Suggested Projects

1. **Create Sequence Diagrams For Engine Simulator:** Using your UML modeling tool, create a set of sequence diagrams to model the StartEngine(), StopEngine(), and CheckEngineStatus() method events.

2. **Programming Project — Aircraft Simulator:** Using the code supplied in the engine simulation project write a program that simulates an aircraft with various numbers of engines. Give your aircraft the ability to start, stop, and check the status of each engine. Implement the capability to inject engine part faults. Utilize a simple text-based menu to control your aircraft's engines. Draw a UML class diagram of your intended design.

3. **Programming Project — Automobile Simulator:** Write a program that implements a simple automobile simulation. Perform an analysis and determine what parts your simulated car needs. Modify the engine simulator code to use in this project, or adopt it as-is. Control your car with a simple text-based menu. Draw a UML class diagram of your intended design.

4. **Programming Project — Gasoline Pump Simulation:** Write a program that implements a simple gas pump system. Perform an analysis to determine what components your gas pump system requires. Control your gas pump system with a simple text-based menu. Draw a UML class diagram of your intended design.

5. **Programming Project — Hubble Space Telescope:** Write a program that controls the Hubble Space Telescope. Perform an analysis to determine what components the telescope control system will need. Control the space telescope with a simple text-based menu. Draw a UML class diagram of your intended design.

6. **Programming Project — Mars Rover:** Write a program that controls the Mars Rover. Perform an analysis to determine what components your rover requires. Control the rover with a simple text-based menu. Draw a UML class diagram of your intended design.

7. **Managed Extensibility Framework (MEF):** Implement any of the above Programming Projects 2 through 6 using MEF.

  

## Self-Test Questions

1. What is a dependency relationship between two objects?

2. What is an association relationship between two objects?

3. What is an aggregation?

4. List the two types of aggregation.

5. Explain the difference between the two types of aggregation.

6. How do you express simple aggregation using a UML class diagram?

7. How do you express composite aggregation using a UML class diagram?

8. Which type of aggregation denotes a *uses* or *uses a* relationship between two objects?

9. Which type of aggregation denotes a *contains* or *has a* relationship between two objects?

10. What is the purpose of a UML sequence diagram?

11. A class built from other class types is referred to as a/an _____.

12. In this type of aggregation, part objects belong solely to the whole or containing class. Name the type of aggregation.

13. In this type of aggregation, part object lifetimes are not controlled by the whole or containing class. Name the type of aggregation.

14. In a UML class diagram, the aggregation diamond is drawn closest to which class, the whole or the part?

15. In a MEF application, what do [Import] and [Export] attributes signify?

## References

Grady Booch, et. al. *The Unified Modeling Language User Guide*. Addison-Wesley, Reading, MA, 1998. ISBN: 0-201-57168-4

Grady Booch. *Object-Oriented Analysis and Design with Applications*. Second Edition. The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1994. ISBN: 0-8053-5340-2

Sinan Si Alhir. *UML In A Nutshell: A Desktop Quick Reference*. O'Reilly and Associates, Inc., Sebastopol, CA. ISBN: 1-56592-448-7

  

Rick Miller. *Java For Artists: The Art, Philosophy, And Science Of Object-Oriented Programming*. Pulp Free Press, Falls Church, VA. ISBN: 1-932504-05-2

Microsoft MEF Documentation Site: https://docs.microsoft.com/en-us/dotnet/framework/mef/

## Notes

  