1.1 Componentware, Cognition, and Software Development

It has often been said we humans make artifacts in our own image. As we gradually rise out of the current quagmire of software development, we are starting to create software along the mental model of how we think. Although human-to-human communication is not perfect, it is amazing how well we interface together. A key element is the mental objects, defined pretty much the same from human to human. We have general agreement on the idea content of each of these objects (messages) we are moving between us. This mental model has been at the core of the rise of object-oriented programming in all its forms.

The specification of software, both functionally and detail design wise, is one of the most critical steps in the creation of a new software project. Unfortunately, these specifications are often ill defined and are one of the more subjective areas currently involved in the intelligent and profitable creation of new software. Currently the human elements of background experience, personality traits, and motivation heavily bias the specifications and subsequent creation of the software. Each human brings to the project table a different set of expectations, capabilities, and desires. The current “software crisis” is influenced heavily by the human psychology profile differences in the team mixture. Of course, the increasing complexity of software is more and more becoming the amplification factor of these human traits.

One of the forms that object-oriented programming is taking is the creation of standard objects we can share between the human programmers and
other project team members. Now we are starting to create software objects that will serve the same type of communication and control functions as mental objects, with well-defined behavior to offset a majority of all the bad human traits currently involved in software creation. The creation of these mental/software objects will aid the programmer in establishing more cognitive awareness of the structure and content of the program. Why does the programmer need more cognitive contact with the software? Because the programmer is the cognizer of the software. It is the programmer’s mental functions that are being imposed upon the creation of the software. It is the programmer’s understanding of the software goals that will direct the creation of the software.

The human learns early on as a child to recognize objects. A child learns to abstract objects very early. For example, a child will understand the abstract object chair, but realizes that there are many derived chair types. Finally, this object learning paradigm has made its way into software development. We are able to create an abstract object class and then, via inheritance, derive a specific object from it. We are able to encapsulate involved functionality with the object. The cognizer then is able to start thinking about domain-specific needs and goals in terms of these abstract and derived objects. The cognizer can conceptualize and create on a much higher level of mental activity due to the object characteristics.

We focus in this book on the Microsoft .NET Framework base classes and their use with COBOL. Microsoft created the base classes to allow the new software technologies of object methodology to be standardized, to greatly enhance the overall software development cycle. The base classes encapsulate the large set of Application Program Interface functions (Win32 APIs) that are used to dialogue with the Microsoft Windows operating systems. They also handle all communication with the Common Language Runtime (CLR) that does the interface with the Win32 APIs. These predefined base classes will allow us to develop functional and detailed design specifications based upon standard objects prespecified for behavior. This greatly facilitates human communications in the software development process because all project team members can review the standard specifications. These new concepts in software development give the project team a great deal of potential technical continuity from the front end of the project to the back end of the project.

For the first time these base classes, along with the CLR, give us language interoperability so that team members working in different languages have a common foundation upon which to communicate with other team members. Fujitsu COBOL, Microsoft C#, Microsoft Visual C++, and Microsoft Visual Basic are major languages for the creation of software objects and components. The .NET Framework is one of the first major cohesive object software sets that has the scope and depth to have a major impact on current and future software development. A great deal of the large fluctuations between individual programmers is eliminated by the standardization achieved by the use of this collection of software objects. To take it one step further, it is a major cornerstone in the
Microsoft Application Framework. Again, the purpose of this application framework is to allow well-defined objects, behaving in well-known ways, to be used by unwell-behaved programmers, which will result in the creation of a better-behaved software development cycle. The Microsoft Visual Studio.NET development environment has extended its philosophy of having software development wizards aid in the creation of software to also having software development wizards to aid in the development of components.

Many current books on objects and components tell programmers how to create objects and components for inclusion in other programs. Some books mention the usage of ActiveX controls as the building blocks for creating whole applications. Not many current books take the viewpoint of the programmer as well as other project people wanting to use rather than develop objects and components. Constructing computer applications by assembling prebuilt software objects (and now components) has long been a goal of software engineering. With the advent of software (mental) objects, as defined in object-oriented programming, and the definition of components, we are starting to put in place the concepts and software tools for fulfilling this goal. But just what are the overall application project design issues raised when applications are created by assembling objects and components? What impact does this have on the cognition of understanding the software goals and design steps? There are architectural questions, like the division of functionality and location of the User Interface (UI) code. What are some of the implementation issues with the .NET Framework, when integrating many assemblies?

So, just how do the programmer and other project people go about using these new software objects? Just what are they currently, and how do we interface them to solve new software applications? What kind of a mental model should the programmer look to develop for the expected software development? This book is going to address such issues. It will cover how we may construct new software applications by assembling prebuilt software objects. It will cover the issues of constructing applications using software objects when the applications are distributed over a network. We will always keep in mind the recent advances in cognitive science to gain an appreciation of how the work in cognitive science and object methodology is starting to form a symbiosis to aid the software developer.

1.2 Object and Cognitive Activities

Intellectual activity is a creative endeavor; it is a function of innate skills, experience, motivation, and focus. It is well known that we humans can conceptualize very complex ideas, without any real understanding of the immense detail needed to implement those ideas. Software development then depends on our abilities to implement the complex ideas we have conceptualized. So the first
thing that happens is that we run head on into the immense details needed to implement our idea(s). It has been known since Aristotle's time that we can best handle complexity using the divide-and-conquer method. This is fine for the human mind, which is an outstanding analytical engine. But what about expressing and understanding a meta-representation of this immense detail, in a foreign language, on a deterministic analytical electronic calculating engine?

Structured Methodology has taken the idea of divide-and-conquer to handle complexity to a much higher level of accomplishment in software development than ever before. However, even this has not provided the overall improvement in production, accuracy, and cost-effectiveness that must occur in software development. Instead of the divide-and-conquer approach, we must group the immense details into functionally intelligent agents. These agents can operate on behalf of the software developer in implementing the developer's conceptualized idea. In the creation of any software solution, for any idea, there is a great deal of repeated techniques used from other solutions or experiences.

As we mentioned above, this brings us to the next big paradigm shift in software development, “Object Methodology.” The creation of objects, where any one object represents some functionally intelligent grouping of details, gives the software developer a higher meta-level upon which to base the solution creation. The developer can now start to think in terms of object behavior, without worrying about how this behavior is implemented or handled. They can develop clusters of objects in relationships that solve various functional requirements of their conceptualized idea. The human mind does just this type of object clustering and management in solving various mental problems.

What we are looking for in the use of objects and components in software development is a technology that will empower individual programmers with a greater and quicker ability to create. We thus want to change the software development process by empowering programmers through object and component assembly technology. The current software development process will not evolve past its current state of crisis by using the same techniques that have created the situation. So the programmer must shift to a more natural way of thinking, one in terms of objects (components) to solve and create software. It was Einstein who recognized, “The world will not evolve past its current state of crisis by using the same thinking that created the situation.” There is no greater possibility for software development than the continual, successful reduction of the rich complexity of software to simple objects, comprehensible to all programmers. We must shift from “a view in which the accent has been on parts and elements, to a configuration view, with the emphasis on wholes and patterns.” Rather than beginning with the pieces of a puzzle and proceeding to construct the puzzle, we begin with a picture of the puzzle and place each piece (object) by constantly referencing the whole.

One of the new software books, Design Patterns, Elements of Reusable Object-Oriented Software (Addison-Wesley, 1995), shows the increasing inter-
One of the fundamental ideas in software component engineering is the use of objects. But just what is an object? There doesn’t seem to be a universally accepted idea as to what an object really is. The view that computer scientist Grady Booch (1991) takes is that an object is defined primarily by three characteristics: its state, its behavior, and its identity. The fundamental unit of analysis, in most cognitive theories, is the information-processing component. A component is an elementary information process that operates on the internal representation of objects or symbols (Newell & Simon, 1972; Sternberg, 1977). If we look at the way these components work, they may translate a sensory input into a conceptual representation, transform one conceptual representation into another, or translate a conceptual representation into a motor output. The object-oriented programming (OOP) techniques for software have been around now for approximately a quarter of a century, but the phenomenon is not new. Ancient philosophers, such as Plato and Aristotle, as well as modern philosophers like Immanuel Kant have been involved in explaining the meaning of existence in general and determining the essential characteristics of concepts and objects (Rand, 1990). Minsky (1986) developed a theory of objects whose behavior closely resembles processes that take place in the human mind. Novak and
Gowin (1984) showed how objects play an important role in education and cognitive science. Their approach is one in which concepts are discovered by finding patterns in objects designated by some name. But wait, we were talking about objects and now we are talking about concepts. That is because concepts reflect the way we divide the world into classes, and much of what we learn, communicate, and reason about involves relations among these classes. Concepts are mental representations of classes, and their salient function is to promote cognitive economy. A class then can be seen as a template for generating objects with similar structure and behavior.

The Object Management Group (OMG) defines a class as follows:

A class is an implementation that can be instantiated to create multiple objects with the same behavior. An object is an instance of a class.

From the software point of view, by partitioning the software into classes, we decrease the amount of information we must perceive, learn, remember, communicate, and reason about.

1.4 What Is a Software Object?

In 1976, Niklaus Wirth published his book *Algorithms + Data Structures = Programs*. This heightened our awareness of the major parts of a program. In 1986, J. Craig Cleaveland published his book *Data Types*. In 1979, Bjarne Stroustrup had started the work on C with classes. By 1985, the C++ Programming Language had evolved and in 1990 the book *The Annotated C++ Reference Manual* was published. I will talk primarily about COBOL and associated .NET Framework base classes and objects in this book, because this is the main focus of this book.

When Bjarne Stroustrup published his book on C++ or C with classes, we started associating the words *class* and *object* with the term “abstract data type.” But what is the difference between data types and abstract data types? A *data type* is a set of values. Some algorithm then operates upon managing and changing the set of values. An *abstract data type* has not only a set of values, but also a set of operations that can be performed upon the set of values. The main idea behind the abstract data types is the separation of the use of the data type from its implementation. Figure 1–1 shows the four major parts of an abstract data type. Syntax and semantics define how an application program will use the abstract data type. Representation and algorithms show a possible implementation.

For an abstract data type, we have therefore defined a set of behaviors and a range of values that the abstract data type can assume. Using the data type does not involve knowing the implementation details. Representation is speci-
fied to define how values will be represented in memory. We call these representations *class member variables* in COBOL. The algorithm or programs specify how the operations are implemented. We call these programs *member functions* in COBOL. The semantics specify what results would be returned for any possible input value for each member function. The syntax specifies the COBOL operator symbols or function names, the number and types of all the operands, and the return values of the member functions. We are therefore creating our own data object (abstract data type) for the software to work with and use, as opposed to only using the data types predefined by the compiler, such as integer, character, and so on. These abstract data types or objects, as defined in Grady Booch’s book *Object-Oriented Analysis and Design*, are as follows: “An object represents an individual, identifiable item, unit, or entity, either real or abstract, with a well-defined role in the problem domain.”

We have slowly been coming to the realization of just what properties our program should have to make it work in solving complex real world problems. Having a new language paradigm like COBOL.NET and its associated capabilities to create classes and objects is not sufficient. We realized that just using the abstract data type or class was also not enough. So as part of this ongoing development, the methodology called *object-oriented technology* evolved what is called the *object model*. The software engineering foundation, whose elements are collectively called the object model, encompass the principles of abstraction, modularity, encapsulation, hierarchy, typing, concurrency, and persistence. The object model defines the use of these elements in such a way that they form a synergistic association.
As with any discipline, such as calculus in mathematics, we need a symbolism or notation in which to express the design of the objects. The creation of the COBOL.NET language, as an example, supplied one language notation needed to write our object-oriented programs. However, we still needed a notation for the design methodology to express our overall approach to the software development. In 1991 Grady Booch published his book, Object-Oriented Design with Applications, in which he defined a set of notations. These notations have become the de facto standard for object-oriented design. His second edition (1994) does an even better job of describing the overall object-oriented design notation and the object model. In this second edition he expresses all examples in terms of the C++ language, which has become a major language for object-oriented software development. We even have a Windows GUI tool based upon this notation to aid us in our thinking. This tool by Rational Corporation and Grady Booch was initially called ROSE. Quite a change from how calculus and its notation were initially used. We almost immediately have the same engine we wish to program on, aiding us in doing the programming. This tool has continued to evolve and is now called the Universal Modeling Language (UML).

An object (or component), then, is an entity based upon abstract data type theory, implemented as a class in a language such as COBOL.NET, and the class incorporates the attributes of the object model. What we have been describing, however, is just the tip of the iceberg relative to objects. The description so far has described the static definitions and has not talked about objects talking with one other. Let’s look at one of the object model attributes, inheritance. Inheritance is our software equivalent of the integrated electronic circuit (IC) manufacturing technique of large-scale integration (LSI) that has allowed such tremendous advances in electronic system creations. Software using inheritance allows the creating of what I will call a small-scale integration (SSI) black box in software. This SSI creates what I will call an encapsulated software cluster of objects directed toward the solution of some function needed for the application. We have thus abstracted away a large amount of the complexity, and the programmer works only with the interfaces of the cluster. The programmer then sends messages between these clusters, just like the electronic logic design has wires between ICs, over which signals are sent.

While we allude to software components having an analogy to hardware chips, this is only true in a most general sense. Software components created with the rich vocabularies of the programming language, and based upon the constructs created by the programmer’s mind, have a far greater range of flexibility and power for problem solving than hardware chips. Of course, therein lies a great deal of the complex nature of software programs. However, the software components ride on top of the hardware chips, adding another complete level of abstraction. I grant you that the deterministic logic involved in a
complex LSI chip is very impressive. But the LSI chip is very limited in the possibility of forming any synergistic relationship with a human mental object. The more I dwell upon the direction of the .NET Framework base classes, in all its technologies, the more I feel we are externalizing the mind’s use of mental object behavior mechanics. Certainly the object relationships formed with linking and embedding software objects, via interfaces, doesn’t look much like the dendrite distribution of influences on clusters of neurons. But certainly now one software object is starting to effect one or more other software objects to accomplish its goal.

Let’s look at a control object or collection of control objects from an everyday practical standpoint that we are using in other engineering fields. One of our early loves is the automobile. We can hardly wait to learn how to drive one. Notice we said drive one, any one. We have done such a great job on our encapsulation and interface exposure that we can learn to drive any kind and be able to drive any other kind. The automobile object we interact with has three primary interface controls: steering wheel, throttle, and brake. We realize that encapsulated within that automobile object is many internal functions. We can be assured that these control interfaces will not change from automobile object to automobile object. In other words, if we go from a General Motors car to a Ford car we can depend on the same functionality of these control interfaces.

Another characteristic of a software object is persistence. Persistence of an object is learned very early by a child. Eventually when we show a child a toy and then hide it behind our back, the child knows the toy still exists. The child has now conceptualized the toy object as part of his mental set of objects. As the programmer does a mental conceptualization of various software objects, this will lead to a high level of persistence of the objects in the programmer’s mind. Since one of the main features of standard software objects is reusability, the efficiency of the programmer will continue to increase as the standard objects are conceptualized in the programmer’s mental model.

Polymorphic behavior is another characteristic that can be implemented in a software object. Probably one of the earlier forms that a child realizes has different behavior, based upon form, is the chair object. The chair object is polymorphic in that its behavior depends on its form. We have rocking chairs, kitchen chairs, lounge chairs, and so on. This idea of form and related behavior has created a whole field of study called morphology. Certainly this is a key idea in how we relate cognitively to various objects. Not only does the clustering of our objects have form relationships, the internal constructs of the objects have a form relationship. There is a definite relationship between the logic flow of a program and the placement of the various meaningful chunks of a program. This is somewhat different than a pure polymorphic nature of a function, but does point out that we should be aware of the morphology of our objects and their parts.
1.5 Object-Oriented Programming with COBOL

In the above discussion we have been talking a bit about the philosophy, psychology, and the nature of an object. We will now take a look at what are called the “pillars” of object-oriented programming, that is, encapsulation, inheritance, and polymorphism. These are the core principals of object-oriented programming. COBOL is a newcomer to the world of object-oriented languages (OOLs). C++, Java, Object Pascal, and to some extent Visual Basic 6.0 are but a small sample of the popularity of the object paradigm. Before we dig into the COBOL syntactic details of each pillar at a high level, we will give you an overview of each of the core principals.

1.5.1 Encapsulation Services

The first pillar of OOP is called *encapsulation*. This is the ability of the language to hide unnecessary implementation details from the object user. For example, if you had created a class named DBWriter (database writer) that has two primary methods, Open() and Close():

```cobol
Identification Division
Class-Id Book.
    <methods and data>
Object.
    <method and data>
End Object.
End Class Book.
```

The fictitious DBWriter class has encapsulated the inner details of locating, loading, manipulating, and closing the data file. Notice that this encapsulation pillar of OOP keeps programming tasks simple. The user does not have to worry about the numerous lines of code working behind the scenes to carry out the work of the DBWriter class. The user just has to create an instance and send the appropriate message (e.g., “open the named file”). Closely related to the notion of encapsulating programming logic is the idea of data hiding. An object’s state data should ideally be specified as private. The outside world then must ask to change or get the underlying value. Public data points can easily become corrupted, so the private aspect is a good thing.

*Note* The database writer encapsulates the details of opening and closing a database.
1.5.2 Inheritance: The “is-a” and “has-a” Relationships

Inheritance is the language’s ability to allow you to build class definitions based on existing class definitions. Inheritance therefore allows you to extend the behavior of a base (parent) class by inheriting core functionality into a subclass (also called a child class). Figure 1–2 shows a simple example. Systems.Object is always the top-most node in any .NET hierarchy. The Shape class extends Object.Shape then defines some number of properties, fields, methods, and events that are common to all shapes. The Circle class extends Shape and inherits the core functionality defined by Shape and Object, as well as defines additional Circle-related details of its own. You then read this diagram as “a circle is-a shape that is-a object.” Thus when you have classes related by this form of inheritance, you establish “is-a” relationships between types. The is-a relationship is often termed classical inheritance.

The other form of code reuse is the containment/delegation model, also known as the “has-a” relationship. A given class can contain another class and expose part or all of its functionality to the user. For example, if you are modeling a truck, you might wish to express the idea that a truck “has-a” heater. It would not be logical to attempt to derive the Truck class from a Heater, or vice versa (a Truck “is-a” heater). You have two independent classes working together where the outer (or containing) class creates and exposes the inner (or contained) class functionality (Figure 1–3).

Here the outer object (Truck) is responsible for creating the inner (Heater) object. If the Truck wishes to make the Heater’s behavior accessible from a Truck instance, it must extend its own public interface. The object user has no idea that the Truck class is making use of an inner object.
Chapter 1 • Review and Introduction to Object-Oriented COBOL Programming

1.5.3 Polymorphism

The last major pillar of OOP is *polymorphism*. This is the language's ability to treat related objects the same way. To illustrate polymorphism, let's revisit the shapes hierarchy. Assume that the Shape class has defined a function named `Draw()`, taking no parameters and returning nothing. Given the fact that every shape needs to render itself in a unique manner, subclasses (such as Circle and Rectangle) are free to reinterpret this method for their own needs (see Figure 1–4).

Polymorphism allows a base class to enforce a given behavior on all descendents. From Figure 1–4, you can assume that any object derived from the Shape class has the ability to be rendered. This is a great capability for any language because you don't need to create redundant methods to perform similar operations (e.g., `DrawCircle()`, `DrawRectangle()`, etc.).

Let's wrap up this review of the foundations of OOP. Recall that every object-oriented language needs to address how it contends with encapsulation, polymorphism, and inheritance. The bulk of this chapter will explore the exact COBOL syntax that represents each trait.
In some cases we may have several methods (similar to functions) that do different things to the same data, say a record. COBOL collects these methods in what is called a *class*. A class describes the data, along with the methods that operate on the data. An object is really just an instance of a class. We can have one or more instances of a given class. You would code a class as you would any COBOL program, with the four divisions, that is, Identification, Environment, Data, and Procedure. Appendix A.4 shows the general rules of the Fujitsu COBOL language. Appendix A.5 shows the definition of the program structure for a Fujitsu object COBOL program. Let's look at a COBOL class listing:

```
000010 CLASS_ID. BIGDOG AS "BigDog" INHERITS DOG.
000020 ENVIRONMENT DIVISION.
000030 CONFIGURATION SECTION.
000040 REPOSITORY.
000050 CLASS DOG AS "Dog".
000060 OBJECT.
000070 PROCEDURE DIVISION.
000080 METHOD-ID. BARK AS "Bark" OVERRIDE
000090 DATA DIVISION.
000100 WORKING-_STORAGE SECTION.
000110 77 I PIC S9(9) COMP-5.
```
Chapter 1 • Review and Introduction to Object-Oriented COBOL Programming

This program (bigdog.cob) and many other examples are on the companion FTP site for this book, http://www.phptr.com/reeves. Notice that this class BIG-DOG inherits from another class called DOG. It turns out that the class DOG is a Visual Basic or VB.NET class. This program is an example of one of the outstanding new features of the .NET Framework, that is, any .NET-enabled language can inherit from any other .NET-enabled language. Let’s look at the VB class listing, dog.vb, for the Dog class.

Imports System

Public Class Dog
    Public Sub RollOver()
        Dim R As New Random()
        Console.WriteLine("Scratch my tummy.")
        Bark(r, Next(1, 4))
    End Sub

    Public Overridable Sub Bark(ByVal times As Integer)
        Dim I As Integer
        For I = 1 To times
            Console.WriteLine("WOOF WOOF (VB)")
        NEXT I
    End Sub
End Class

To further illustrate this interoperability between .NET-enabled language, let’s look at the C# (pronounced “C Sharp”) driver class for this example.

using System;

class public Demo
{
    public static void Main()
    {
        Dog d = new Dog();
        BigDog b = new BigDog();

        d.RollOver();
        b.RollOver();
    }
}
Notice how this class is instantiating a Visual Basic object \( d \) and then instantiating a COBOL object \( b \). It then calls the RollOver method from both the Visual Basic and the COBOL classes.

With a very simple example we have formed a relationship between three different .NET-enabled languages. By working with the three different classes and then the instantiated objects we have been able to communicate between the languages and perform some programming work. All three languages are handled in the same fashion using the Visual Studio.NET IDE. Now the powerful tools for Windows development is available to the COBOL developer just as in the past they were for C++ and Visual Basic (VB). By the way, the C# class could just as well have been a Visual C++ class. (This project is on the FTP site under the folder MultiDog.) What we would do then is bring up the Visual Studio.NET. Under the File pull-down, we would open the solution to the folder MultiDog. We would then open the project icon for the dog project. At this point we would see a screen, as shown in Figure 1–5. Under the Build pull-down we would click on Build.
Solution. Once it shows a good build we would use the Debug heading and the pull-down Step Into. This would show the screen in Figure 1–6. Using the F10 function key, we would step down to the line just after the Console.WriteLine. Then we would click on the icon on the status bar for the console screen. This would show the Console screen and the results of running the program. See how we have outputs from all three objects for each language—VB, C#, and COBOL? At this point under the Debug pull-down we would stop the debugger. Then we would go over to File pull-down and close the solution. I have thrown a lot at you at this point, but it certainly shows how easy it is to work with classes from different .NET-enabled languages and perform work.

1.7 COBOL Classes and the .NET Base Classes

We have seen above how the different .NET-enabled languages can inherit and communicate with each other. This is a powerful feature of the .NET Framework, however, the real power for the developer in developing new
applications is in the thousands of .NET base classes available. Using these base classes is how the programmer communicates with the Common Language Runtime (CLR), which then controls the powerful operating system engine. The CLR works with an extensive set of Win32 APIs to implement the functionality desired. Figure 1-7 shows this diagram of the layers of classes.

Let's look at an example of using several of the .NET base classes by a COBOL program. The WalkTheDog example is on the FTP site. The program is WalkTheDog.cob, which we show next.

000010 CLASS-ID.  Walk INHERITS Form.
000020 ENVIRONMENT DIVISION.
000030 CONFIGURATION SECTION.
000040 REPOSITORY.
000050  CLASS Form AS "System.Windows.Forms.Form"
000070  CLASS Container AS "System.ComponentModel.Container"
000090  CLASS Timer AS "System.Windows.Forms.Timer"
000100  CLASS EventHandler AS "System.EventHandler"
000110  CLASS EventArgs AS "System.EventArgs"
000120 CLASS Point AS "System.Drawing.Point"
000130 CLASS Drawing-Size AS "System.Drawing.Size"
000140 CLASS Sys-Object AS "System.Object"
000150 CLASS MessageBox AS "System.Windows.Forms.MessageBox"
000160 CLASS Control-Collection AS "System.Windows.Forms.Control+ControlCollection"
000170 PROPERTY Interval AS "Interval"
000180 PROPERTY Enabled AS "Enabled"
000190 PROPERTY Location AS "Location"
000200 PROPERTY Button-Size AS "Size"
000210 PROPERTY TabIndex AS "TabIndex"
000220 PROPERTY Button-Text AS "Text"
000230 PROPERTY App-Controls AS "Controls"
000240 PROPERTY App-Text AS "Text"
000250 PROPERTY AutoScaleBaseSize AS "AutoScaleBaseSize".
000260 STATIC.
000270 PROCEDURE DIVISION.
000280 METHOD-ID. MAIN.
000290 DATA-DIVISION.
000300 WORKING-STORAGE SECTION.
000310 77 Obj OBJECT REFERENCE Walk.
000320 PROCEDURE DIVISION.
000330 INVOKE Walk "New" RETURNING Obj.
000340 INVOKE Application "Run" USING BY VALUE Obj.
000350 END METHOD MAIN.
000360 END STATIC.
000370 OBJECT.
000380 DATA DIVISION.
000390 WORKING-STORAGE SECTION.
000400 77 Components OBJECT REFERENCE Container.
000410 77 Button1 OBJECT REFERENCE Button.
000420 77 Timer1 OBJECT REFERENCE Timer.
000430 PROCEDURE DIVISION
000440 METHOD-ID. NEW.
000450 PROCEDURE DIVISION.
000460 INVOKE Self "InitializeComponent".
000470 END METHOD NEW.
000480 METHOD-ID. DISPOSE AS "Dispose" OVERRIDE.
000490 PROCEDURE DIVISION.
000500 INVOKE SUPER "Dispose".
000510 INVOKE Components "Dispose".
000520 END METHOD DISPOSE.
000530 METHOD-ID. INITIALIZECOMPONENT AS "InitializeComponent".
000540 DATA DIVISION.
000550 WORKING-STORAGE SECTION.
000560 77 TICK-EVENT OBJECT REFERENCE EventHandler.
000570 77 Click-Event OBJECT REFERENCE EventHandler.
000580 PROCEDURE DIVISION.
Let's build this project and run it. The results are shown Figure 1–8. Kind of reminds me of my wife telling me to go walk the dog over and over until I say stop it (push the button) and I go walk the dog. The program is
using the .NET base classes to implement all the code for creating the MessageBox, displaying it on a timed interval, inserting the text, and picking up when the Stop It button is pushed. It also uses the base classes for drawing and sizing the displays. This is but a minor example of the many classes and the tremendous amount of methods and data that are available in the classes for creating applications. Another interesting aspect is the creating of shorthand descriptions for activities, which saves a lot of code writing. For example, line 00090 shows CLASS Timer AS “System.Windows.Forms.Timer”, however, in the code such as line 000420 when we defined the timer object, we used Timer instead of System.Windows.Forms.Timer. If you were to look at Section 2.2 in Chapter 2, you would see some of the main .NET Framework namespaces. For example, in the listing of namespaces item 9 you would see System.Drawing. Line 000130 above shows CLASS Drawing-Size AS “System.Drawing.Size”. We have thus used one of the namespace methods “Size” of System.Drawing and assigned it a shorthand definition of Drawing-Size. If you look at line 000750 SET AutoScaleBaseSize OF SELF TO Drawing-Size::“New” (5 13), we are setting up the drawing size for display. More insight will be provided in Chapter 2 on how the .NET base classes are
used. These base classes almost completely encapsulate the some 1,925 Win32 APIs of Windows, and we never have to worry about that amount of complexity. However, in Chapter 8 we cover some of the more fundamental Win32 APIs to give you a feel for this complexity and to let you gain some understanding of the Windows engine underneath the .NET Framework.