

9. Serializers

9.1. Prerequisites and Objectives

Before starting this chapter you should have:

1. A knowledge of programming in C++.
2. A knowledge of component interface design.

After completing this chapter you will have:

1. Rudimentary knowledge of semi-persistent objects.
2. A knowledge of how to handle file I/O, clipboard operations, and printing.
4. A knowledge of the Serializer-Component design methodology..

9.2. Introduction

Serialization is the process of writing an object, or a collection of objects, to a persistent medium such as a file. There are three distinct types of serialization, saving and restoring files, copying to and from the clipboard, and printing.

Serialization mechanisms are widely available in different object packages and languages. The MFC class system provides serialization mechanisms for many different kinds of objects. Serialization mechanisms are also available in Java and other languages, however these generic mechanisms may not be suitable for complex objects containing pointers and many aggregate objects. In this section we show how to design a serialization mechanism from scratch. We begin by showing how to make a set of classes serializable, and finish by giving examples of the three types of serialization. We will use the Simple Graphical Editor (SGE) as a basis for this material.

9.3. The Methodology

The serialization components themselves are relatively simple. For serialization operations like writing files, one passes the component a CGraphicList pointer. For deserialization operations like reading files, one passes a file name to the component and receives an CGraphicList pointer in return. Most of the work will be performed by functions built into the CGraphicList object and its aggregates. Each graphical object will know how to serialize itself. Serialization consists of extracting all non-pointer information into a buffer. If it is necessary to retain pointers, they must be converted into a machine-independent form (usually a binary index of some sort). Type codes, which can be omitted in polymorphic objects, must be included in the serialized data. One or more deserialized objects will be stored in a buffer and written to persistent storage.

Unlike serialization, which is done by member functions of the graphical objects, deserialization must be done, at least in part, by a global function. The individual objects, like rectangle and circle, have to be identified before they can be turned into internal objects of the correct type. The remainder of this chapter illustrates the serialization process with three different types of serializers for the Simple Graphics Editor.

9.4. The Simple Graphics Editor Serialization Functions

For a serialization mechanism to conform to the philosophy of object-oriented design, each object must serialize itself. This can mean many things. In some cases this means that when an object is provided with an archive pointer, it will write its state to that archive. This forces a great deal of functionality into the object, since it must be capable of dealing with all required types of archives.

We don't wish to force this much functionality into our objects. Instead, we wish to provide a serialization mechanism that converts data into a form that can be used for many different purposes. To keep the serialization mechanism as simple as possible, we will provide the component with a storage area, and expect it to serialize itself into that storage area.

We must modify the **CGraphicList** project so that the **CGraphicList** object and its subobjects serialize themselves. The general mechanism for doing this is shown in Figure 9.1. In this routine, a **CGraphicList** object is serialized into a string variable. The **CGraphicList** object passes the serialization request to each graphical object, and accumulates the results in its own string variable. Our objects are simple enough that no header information is required. If, however, this were the case, this information would be added to the **CGraphicList** string variable before adding the information from the graphical objects. (Note, **CString** is an MFC string type that permits strings to be expanded without a fixed limit. The overloaded += operator concatenates new data onto the end of an existing string.)

```
void CGraphicList::SSerialize(CString &S)
{
    S.Empty();
    CGraphicObject * Temp;
    // Serialize each graphical object
    for (Temp = Head ; Temp != NULL ; Temp = Temp->Next)
    {
        CString T;
        Temp->SSerialize(T);
        S += T;
    }
}
```

Figure 9.1. Serializing the CGraphicList Object.

At times we may not wish to serialize only selected portions of the object. The function *SSerializeSelect* of Figure 9.2 passes the serialization request to selected objects only.

```
void CGraphicList::SSerializeSelect(CString &S)
{
    S.Empty();
    CSelectItem * Temp;
    for (CTemp = Selection ; Temp != NULL ; Temp = Temp->Next)
    {
        CString T;
        Temp->Item->SSerialize(T);
        S += T;
    }
}
```

Figure 9.2. Serializing the Selection.

To serialize the graphical objects **CCircle** and **CRectangle**, we first add the following pure virtual function definition to the definition of **CGraphicObject**. Adding this definition forces us to add an *SSerialize* function to both the **CCircle** and **CRectangle** classes.

```
virtual void SSerialize(CString &rv) = 0;
```

Each graphical object is required to serialize its internal state as a human-readable line of text, and is required to terminate that line with a **return** character and a **line-feed**. To permit deserialization, the first character of the line will be used to identify the type of the object. (Some means of identifying objects is required in any serialization mechanism.) The letter “C” will be used to identify circles, and the letter “R” will be used to identify rectangles. Figure 9.3 shows the *SSerialize* function for the **CCircle** class. (Note: the *Format* function of the *CString* class works like the standard C function

sprintf. The overloaded + operator is used for concatenation.) Strictly speaking, the **CGraphicObject** class should implement its own *SSerialize* function for serializing fill and line colors. The **CCircle** class would then call its base-class function before serializing its own data. Due to the simplicity of our class structure, we have elected to serialize the entire object in the **CCircle** and **CRectangle** functions. Figure 9.4 gives the implementation of the **CRectangle** *SSerialize* function.

```
void CCircle::SSerialize(CString &rv)
{
    // String serializer
    rv = "C ";
    CString Work;
    Work.Format("%8.8x",FillColor);
    rv += Work + " ";
    Work.Format("%8.8x",LineColor);
    rv += Work + " ";
    Work.Format("%d",Center.x);
    rv += Work + " ";
    Work.Format("%d",Center.y);
    rv += Work + " ";
    Work.Format("%d",Radius);
    // Line Terminator
    rv += Work + "\r\n";
}
```

Figure 9.3. Serializing the CCircle Object.

```

void CRectangle::SSerialize(CString &rv)
{
    // String serialization function
    rv = "R ";
    CString Work;
    Work.Format("%8.8x",FillColor);
    rv += Work + " ";
    Work.Format("%8.8x",LineColor);
    rv += Work + " ";
    Work.Format("%d",Left);
    rv += Work + " ";
    Work.Format("%d",Top);
    rv += Work + " ";
    Work.Format("%d",Width);
    rv += Work + " ";
    Work.Format("%d",Height);
    // Line Terminator
    rv += Work + "\r\n";
}

```

Figure 9.4. Serializing the CRectangle Object.

This completes the serialization mechanism of the CGraphicList object. The string created by the CGraphicList SSerialize function can be written to a text file. The full implementation of the CGraphicList object also includes a binary serialization mechanism. This mechanism is a bit more difficult to implement, but the reverse operation, the deserialization mechanism, is reasonably simple. The string deserialization mechanism, on the other hand, is quite complicated.

The binary deserialization mechanism uses the two structures shown in Figure 9.5. These structures are also used by the string deserialization mechanism for temporary storage. This allows both deserialization mechanisms to use the same special constructors to create **CCircle** and **CRectangle** objects. Beyond showing these structures, we will not discuss the binary serialization mechanism.

<pre>struct SerRectangle { long Type; COLORREF FillColor; COLORREF LineColor; long Left; long Top; long Width; long Height; };</pre>	<pre>struct SerCircle { long Type; COLORREF FillColor; COLORREF LineColor; long CenterX; long CenterY; long Radius; };</pre>
--	---

Figure 9.5. Binary Serialization Structures.

The deserialization function is quite long, but repetitive. Figure 9.6 shows the important parts of this function. The caller of this function is required to break the serialized string into individual lines, and pass the lines one at a time to the *SDeserialize* function. The *SDeserialize* function will return NULL if a format error is found, otherwise it will return a pointer to a **CGraphicObject**. The *GetWord* function is used to parse the input string into white-space delimited words. This function straightforward and its implementation is omitted for brevity.

```

CGraphicObject * SDeserialize(CString LineIn)
{
    // Each input consists of a line
    // The first character of the line determines the type of object
    // The remaining parameters are space delimited words on the line
    CString Work;
    if (LineIn[0] == 'C')
    {
        // First copy all parameters into a serialization object
        SerCircle C;
        // Get word will find the first word starting at Pos
        // set Pos to skip type code
        int Pos = 1; // int is required for CString indexing
        // First Parameter: Fill Color
        // GetWord changes Pos
        Work = GetWord(LineIn,Pos);
        if (Work.IsEmpty())
        {
            return NULL; // Empty means NO WORD FOUND: Format Error!
        }
        Work = "0x" + Work;
        C.FillColor = strtol(Work,NULL,0); // color in hex
        ... // Repeat for Line Color
        // Third Parameter: Center.x
        Work = GetWord(LineIn,Pos);
        if (Work.IsEmpty())
        {
            return NULL; // Empty means NO WORD FOUND: Format Error!
        }
        C.CenterX = atol(Work);
        ... // Repeat for the other two parameters
        // Now create the circle structure
        CCircle * rv = new CCircle(C);
        return rv;
    }
    else if (LineIn[0] == 'R')
    {
        ... // Same as for CCircle
    }
    else
    {
        // Invalid type code
        return NULL;
    }
}

```

Figure 9.6. The SDeserialize Function.

9.5. The Simple Graphics Editor File Handler

To write a file we must have both a file name and something to write into the file.

Thus the SGE file handler needs at least two properties, one to supply the file name and

one to supply the content. Figure 9.7 gives the description of two properties, *FileName* and *ModelHandle*, which are used for this purpose.

Property Design Table		
Name	Type	Function
ModelHandle	Long Integer Maximum=N/A Minimum=N/A Default=None	Provides the address of the modeling object cast to a long integer. NULL values are ignored.
FileName	String Default = Empty String	Contains the file name of the file to be read or written. This property will also be set by the <i>SaveAs</i> and <i>OpenFile</i> properties.

Figure 9.7. ModelHandle And FileName Property Descriptions.

Next, we must provide methods that perform the functions required to read and write files. The *Open* method will open the file specified by the *FileName* property, read its contents, deserialize the contents and create a new model that will be available through the *ModelHandle* property. The *Save* method will serialize the content of the object specified by the *ModelHandle* property, and write the serialized data to the file specified by the *FileName* property. (Any existing file will be overwritten.) Figure 9.8 gives the method definitions of these two methods.

Method Description			
Name	Open		
Return Value	Long		
Description	Reads the file specified by FileName, creates a CGraphicList object, and makes the object available in the ModelHandle property. If the file cannot be read, an error code is returned, otherwise a success code is returned.		
Arguments	Name	Type	Description
Void			

Method Description			
Name	Save		
Return Value	Long		
Description	Serializes the model specified by ModelHandle, and writes the serialized data to the file specified by FileName. If the file cannot be written, an error code is returned, otherwise a success code is returned.		
Arguments	Name	Type	Description
Void			

Figure 9.8. Open and Save Method Descriptions.

Figure 9.9 contains the implementations of the *Open* and *Save* methods. Two utility functions, *ReadModel* and *SaveModel* perform the actual read and write operations. The *Open* and *Save* methods check that file name is not empty, while the *Save* method also checks that the model handle is not NULL.

```
long CSGEFileCtrl::Open()
{
    // We need a file name to open
    if (m_fileName.IsEmpty())
    {
        return FILE_NAME_EMPTY;
    }
    // Read a new model from the specified file
    return ReadModel();
}

long CSGEFileCtrl::Save()
{
    // Must have a file name to save
    if (m_fileName.IsEmpty())
    {
        return FILE_NAME_EMPTY;
    }
    // Must have a model to save
    if (Model == NULL)
    {
        return MODEL_IS_NULL;
    }
    // Save model to specified file
    return SaveModel();
}
```

Figure 9.9. Open and Save Implementations.

In addition to these two methods we will also create a *SaveAs* method that supplies a file name as a parameter, and an *OpenFile* method, which also supplies a file name. The names supplied by either of these methods replace the current value of the *FileName* property. The definitions of these two methods are given in Figure 9.10. The implementations are virtually identical to those of the *Open* and *Save* methods, and are omitted.

Method Description			
Name	OpenFile		
Return Value	Long		
Description	Reads the file specified by the NewFileName parameter, creates a CGraphicList object, and makes the object available in the ModelHandle property. If the file cannot be read, an error code is returned, otherwise a success code is returned.		
Arguments	Name	Type	Description
1	NewFileName	String	Supplies the name of the file to be read. This must not be the empty string.

Method Description			
Name	SaveAs		
Return Value	Long		
Description	Serializes the model specified by ModelHandle, and writes the serialized data to the file specified by FileName. If the file cannot be written, an error code is returned, otherwise a success code is returned.		
Arguments	Name	Type	Description
1	NewFileName	String	Supplies the name of the file to be written. This must not be the empty string.

Figure 9.10. OpenFile and SaveAs Method Descriptions.

Finally we will create a *Backup* method which can be used to create a backup copy of a file before the file is rewritten. This is done by renaming the existing file with a “.bak” suffix. Figure 9.11 gives the method description of this property, while Figure 9.12 gives the implementation. The implementation uses a utility function, *RenameFile* to do the actual renaming.

Method Description			
Name	Backup		
Return Value	Long		
Description	Serializes the model specified by ModelHandle, and writes the serialized data to the file specified by FileName. If the file cannot be written, an error code is returned, otherwise a success code is returned.		
Arguments	Name	Type	Description
Void			

Figure 9.11. Backup Method Description.

```

long CSGEFileCtrl::Backup()
{
    // renames specified file with a new suffix (000, 001, ...)
    // Must have a file to rename
    if (m_fileName.IsEmpty())
    {
        return FILE_NAME_EMPTY;
    }
    // turn the existing file into a backup copy
    return RenameFile();
}

```

Figure 9.12. The Backup Method Implementation.

The real work of the component is done by the *SaveModel*, *ReadModel*, and *RenameFile* functions. The *RenameFile* function is straightforward and will be omitted. The implementation of the *SaveFile* function is given in Figure 9.13. The real work of this routine is done by the serialization facility of the **CGraphicList** object.

```

long CSGFileCtrl::SaveModel()
{
    // called by Save and SaveAs functions
    // Open specified file. If file exists, overwrite
    HANDLE SF = CreateFile(m_fileName,GENERIC_WRITE,0,NULL,
        CREATE_ALWAYS,FILE_ATTRIBUTE_NORMAL,NULL);
    if (SF == INVALID_HANDLE_VALUE)
    {
        // specified file can't be opened for writing
        return FILE_OPEN_ERROR;
    }
    unsigned long BytesWritten;
    CString S;
    // Serialize the drawing as a string,
    //and write the string to the open file
    Model->SSerialize(S);
    WriteFile(SF,S,S.GetLength(),&BytesWritten,NULL);
    // Close the file
    CloseHandle(SF);
    S.Empty();
    // Zero return indicates all OK
    return 0;
}

```

Figure 9.13. The SaveModel Implementation.

The *ReadModel* function is longer and more involved than the *SaveModel* function. The main difficulty is breaking the input data into lines, which then must be passed to the deserialization function. To do this, characters must be examined one at a time. However, for efficiency reasons we cannot read one byte at a time. Instead a character buffer is used to hold data read from the file. This data is scanned one byte at a time, and a line of data is accumulated in the line buffer. When an end of line character is found, the line is passed to the deserialization function, and the resultant data structure is chained into the model being built. When the buffer is completely processed it is replenished by reading the file. When end of file is reached, any data remaining in the line buffer is passed to the deserialization function, and the process terminates. Figure 9.14 gives the implementation of the *ReadModel* function. *ReadModel* uses a new **CGraphicList** function *AddObject*. This function inserts a graphical object into the linked list of graphical objects, without

using the Undo facility. Since an entirely new model is being built, it would be inappropriate to permit the individual object insertions to be undone.

```

long CSGEFileCtrl::ReadModel()
{
    // Attempt to open file for reading
    HANDLE RF = createFile(m_fileName,GENERIC_READ,FILE_SHARE_READ,
        NULL,OPEN_EXISTING,FILE_ATTRIBUTE_NORMAL,NULL);
    if (RF == INVALID_HANDLE_VALUE)
    {
        // If file does not exist, or other error
        return FILE_OPEN_ERROR;
    }
    // create a new model empty model
    Model = new CGraphicList;
    static char Buffer[1000];
    unsigned long BytesRead = 0;
    unsigned long BufferBytes = 0;
    unsigned long BP = 0;
    unsigned long FileLength = GetFileSize(RF,NULL);
    CString S;
    S.Empty();
    // The following loop will read the file in 1000 byte chunks
    // Each chunk will be scanned one byte at a time, searching for
    // the end of a line. Once a full line has been accumulated, the
    // line will be passed to the SDeserialize function to create a
    // drawing object. The SDeserialize is a global function that is
    // provided with the model.
    while (BytesRead < FileLength)
    {
        ReadFile(RF,Buffer,1000,&BufferBytes,NULL);
        if (BufferBytes <= 0)
        {
            // Number of bytes read is zero (or less?)
            // This must be a read error, so shut down
            // the process and report an error
            CloseHandle(RF);
            delete Model;
            Model = NULL;
            return FILE_FORMAT_ERROR;
        }
        // Scan the current chunk looking for EOL
        // For chunks after the first, there will normally
        // be some bytes left from the last chunk in CString S.
        while (BP < BufferBytes)
        {
            // accumulate current byte into S
            S += Buffer[BP];
            if (Buffer[BP] == '\n')
            {
                // If EOL found, deserialize the line,
                // erase the string
            }
        }
    }
}

```

```

        // add the deserialized object to the model
        CGraphicObject * Obj = SDeserialize(S);
        S.Empty();
        if (Obj == NULL)
        {
            // NULL return from SDeserialize indicates a
            // format error. Delete the model, close the file
            // and report an error.
            delete Model;
            Model = NULL;
            CloseHandle(RF);
            return FILE_FORMAT_ERROR;
        }
        // Add object to model WITHOUT UNDO or setting Dirty bit
        Model->AddObject(Obj);
    }
    // go to next character
    BP++;
}
// go to next chunk
BytesRead += BufferBytes;
}
if (!S.IsEmpty())
{
    // if there are leftover bytes in S after the last EOL was found
    // it is probably an unterminated line. Go ahead and process it.
    CGraphicObject * Obj = SDeserialize(S);
    S.Empty();
    if (Obj == NULL)
    {
        // NULL return from SDeserialize indicates a
        // format error. Delete the model, close the file
        // and report an error.
        delete Model;
        Model = NULL;
        CloseHandle(RF);
        return FILE_FORMAT_ERROR;
    }
    // Add object to model WITHOUT UNDO or setting Dirty bit
    Model->AddObject(Obj);
}
// zero return indicates all OK
return 0;
}

```

Figure 9.14. The ReadModel Function.

9.6. The Simple Graphics Editor Clipboard Handler

The SGE clipboard handler will make use of the serialization and deserialization functions developed for the SGE file handler. The Windows clipboard supports several

standard types of data, and permits programs to define their own data types. To add data to the clipboard, it is necessary to create a memory block, insert the clipboard data into the memory block, and pass the memory block to the clipboard. Standard operating system functions are used to allocate the memory block and pass it to the clipboard. Like the file handler, the clipboard handler must have a *ModuleHandle* property that supplies the data to be placed on the clipboard. Figure 9.15 contains the description of this property.

Property Design Table		
Name	Type	Function
ModelHandle	Long Integer Maximum=N/A Minimum=N/A Default=None	Provides the address of the modeling object cast to a long integer. NULL values are ignored.

Figure 9.15. Clipboard Handler Property Descriptions.

The clipboard handler also requires three methods, *Copy*, *Cut*, and *Paste*. *Copy* will copy the current selection to the clipboard. *Cut* will do the same, but will delete the selection on completion of the copy. *Paste* will insert the clipboard data into the drawing. If there is a current selection, the pasted data will replace the selection. Figure 9.16 gives the method descriptions of these three methods.

Method Description			
Name	Copy		
Return Value	Void		
Description	Copies the current selection to the clipboard. If ModelHandle is NULL, does nothing.		
Arguments	Name	Type	Description
Void			

Method Description			
Name	Cut		
Return Value	Void		
Description	Copies the current selection to the clipboard, and then deletes the current selection. If ModelHandle is NULL, does nothing.		
Arguments	Name	Type	Description
Void			

Method Description			
Name	Paste		
Return Value	Void		
Description	Pastes the clipboard data into the selection. If the clipboard is empty, or ModelHandle is NULL, does nothing. If there is a current selection, it is deleted before the paste operation is done.		
Arguments	Name	Type	Description
Void			

Figure 9.16. Copy, Cut, and Paste Method Descriptions.

The implementations of *Cut* and *Copy* are similar and are given in Figure 9.17. These methods use a utility function, *DoCopy*, to copy the selection to the clipboard. *DoCopy* returns a success/failure code which is used by the *Cut* method to determine whether it should delete the current selection. The return code is ignored by the *Copy* method.

```
void CSGEClipCtrl::Copy()
{
    // called by copy and cut
    DoCopy();
}

void CSGEClipCtrl::Cut()
{
    long rv = DoCopy();
    // Don't delete the selection unless the copy was successful
    if (rv)
    {
        Model->DeleteSelection();
    }
}
```

Figure 9.17. Cut and Copy Definitions.

The Implementation of the *DoCopy* function is given in Figure 9.18. This function first serializes the selection and then, if the serialized selection is not empty, it opens the clipboard, clears it, creates a memory block, copies the serialized data into the memory block, passes the memory block to the clipboard with a private data type, and finally closes the clipboard. The private data type is created when the file handler component is instantiated, and is destroyed when all programs referring to the type terminate, and the clipboard does not contain data of this type. The process of copying data to the clipboard consists of a set of standard Windows operating system function calls. *DoCopy* returns 1 if the operation was successful, and 0 otherwise.

```

long CSGEClipCtrl::DoCopy()
{
    CString Buffer;
    if (Model != NULL)
    {
        // Serialize each object, and accumulate the data in Buffer
        Model->SSerializeSelect(Buffer);
        // If there was something to serialize, put it in the clipboard
        if (!Buffer.IsEmpty())
        {
            // can't use MFC functions, control might not have a window.
            if (!::OpenClipboard(::GetActiveWindow()))
            {
                // Couldn't get the clipboard
                return 0;
            }
            ::EmptyClipboard();
            // Add first format to clipboard: Private format allowing us
            // to paste objects from the clipboard into a drawing
            HGLOBAL Mem = GlobalAlloc(GMEM_MOVEABLE, Buffer.GetLength()+1);
            if (Mem == NULL)
            {
                // couldn't allocate memory, return error
                ::CloseClipboard();
                return 0;
            }
            char *MemAddr = (char *)GlobalLock(Mem);
            memcpy(MemAddr, Buffer, Buffer.GetLength()+1);
            GlobalUnlock(Mem);
            // clipboard now owns memory block
            ::SetClipboardData(ClipID, Mem);
            // Done with clipboard.
            ::CloseClipboard();
        }
    }
    // All OK
    return 1;
}

```

Figure 9.18. The DoCopy Function.

The implementation of the *Paste* method is shown in Figure 9.19. This function is similar to the deserialization function used by the SGE file manager. The clipboard is queried for the proper type of data, the private type used by the DoCopy function. If data is present, the address and size of the memory block are obtained from the clipboard. The block is scanned one character at a time looking for end of line characters. The content of each line is accumulated in the line buffer. When a complete line has been found, it is

passed to the deserialization function to create a drawing object. This drawing object is inserted into the model using a new **CGraphicList** function *InsertObject*. The *InsertObject* function uses the Undo facility, so a *Paste* operation can be undone.

```

void CSGEClipCtrl::Paste()
{
    HGLOBAL Mem;

    if (Model == NULL)
    {
        // can't paste to nothing
        return;
    }
    if (!IsClipboardFormatAvailable(ClipID))
    {
        // Nothing to paste -- no data available
        return;
    }
    // can't use MFC functions, this control might not have a window.
    if (!::OpenClipboard(::GetActiveWindow()))
    {
        // can't paste because we can't get the clipboard
        return;
    }
    Mem = ::GetClipboardData(ClipID);
    if (Mem == NULL)
    {
        // There was supposed to be data, but we couldn't get it
        ::CloseClipboard();
        return;
    }
    // paste replaces current selection
    Model->DeleteSelection();
    char *Buffer = (char *)GlobalLock(Mem);
    long BP = 0;
    long BufferBytes = GlobalSize(Mem);
    CString S;
    S.Empty();
    // clear redos
    Model->ClearRedo();
    // This must be called before the Model->InsertObject function
    // is called. InsertObject will place all redos into the empty
    // command created by NewUndoCommand
    Model->NewUndoCommand();
    // Scan the clipboard data one byte at a time looking for EOLs
    // Accumulate characters in CString S
    // when an EOL is found, pass the line to SDeserialize
    // SDeserialize is a global function provided with the model
    while (BP < BufferBytes)
    {
        S += Buffer[BP];
        if (Buffer[BP] == '\n')
        {
            S.Empty();
            CGraphicObject * Obj = SDeserialize(S);
            // Test for format error. We will ignore
            // any bad lines and paste what we can.
            // (Bad lines indicate an error in DoCopy)
            if (Obj != NULL)
            {
                // Add object with UNDO and dirty bit set

```

```

        Model->InsertObject(Obj);
    }
}
BP++;
}
// release the clipboard memory block
GlobalUnlock(Mem);
// if there is data left in S, it must be an unterminated line
// attempt to process it.
if (!S.IsEmpty())
{
    CGraphicObject * Obj = SDeserialize(S);
    S.Empty();
    if (Obj != NULL)
    {
        Model->InsertObject(Obj);
    }
}
// release the clipboard
::CloseClipboard();
}

```

Figure 9.19. The Paste Method Implementation.

9.7. The Simple Graphics Editor Print Control

The SGE Print Control is the simplest of our three serializers. It provides two properties, *ModelHandle* and *PrintDC*. (See Figure 9.20 for the formal descriptions.) Once a model has been passed to the control through the *ModelHandle* property, the contents of the drawing can be printed by assigning a printer drawing context handle through the *PrintDC* property. The Windows operating system uses the same method for printing as is used for drawing. A drawing context is created for the desired printer. Just like a window drawing context, the printer drawing context encapsulates the standard drawing functions and translates them into printer commands. To print a drawing, we can use the same drawing routine that we use to display the drawing on the screen.

In most cases the printer drawing context will be obtained from another ActiveX control, the Microsoft Common Dialogs control, for example. The Print Control will not retain a copy of this drawing context, since printer drawing contexts are usually

temporary items that become invalid after a period of time. Each time a print operation is performed, a new drawing context handle must be supplied.

The biggest problem with printing is scaling. Our drawing routine runs in TEXT mode, which translates drawing coordinates into screen coordinates without scaling. Therefore the two points (10,10) and (10,20) will be in the same vertical column, ten pixels apart. If this mode is used for printing, the drawings will be tiny. All control windows have an assumed resolution of 100 pixels per inch, while printers have a resolution of 300-600 dots per inch (DPI) and more. (The actual resolution of the window varies with the screen resolution.) The difference in physical resolution will cause printed drawings to appear much smaller than they appear on the screen. Thus it is necessary to scale printer drawings to make them the proper size.

Property Design Table		
Name	Type	Function
ModelHandle	Long Integer Maximum=N/A Minimum=N/A Default=None	Provides the address of the modeling object cast to a long integer. NULL values are ignored.
PrintDC	Handle to a Printer Drawing Context.	Provides the drawing context for a print operation. The print operation is performed immediately. The drawing context handle is discarded after the print operation is complete. Restrictions: Write Only, Run Time Only

Figure 9.20. Print Handler Property Descriptions.

The general problem of scaling is discussed in Chapter 7. A similar technique is used to scaling an entire drawing. Let us assume that we wish to scale a drawing from R_1 DPI to R_2 DPI. A point (x, y) must be transformed to $(xR_2 / R_1, yR_2 / R_1)$. If integer arithmetic is used, the multiplication must be performed first to avoid problems with round-off

error. We could modify the drawing routines of the `CGraphicList` model to add scaling parameters, but this is not the best way to solve the problem. In fact, if we expand the `CGraphicList` model to include text, this scaling method will no longer work. The size of text is determined by the size of the font being used. There are no size parameters in the drawing commands.

Fortunately the Windows operating system provides a mechanism that can be used to scale and translate *all* drawing operations. To use this mechanism, we must use the *anisotropic* drawing mode. This mode allows drawings to be scaled by different amounts in the *x* and *y* directions. Two functions are used to control scaling. The *SetWindowExt* function of the drawing context object (**CDC**) has two parameters that give the *x* and *y* resolution of the drawing parameters that are supplied to the drawing functions. In our case we will use 100 for each of these parameters. The *SetViewportExt* function of the **CDC** object supplies the *x* and *y* resolutions of the device. If we were printing on a 600 DPI printer, we would use 600 for each of these parameters. The **CDC** object has a query function, *GetDeviceCaps*, that can supply the *x* and *y* resolutions of the device to which it is attached. Translation, which is not required by our print routine, is handled by the functions *SetWindowOrigin*, and *SetViewportOrigin*.

Figure 9.21 shows how all of this is put together into a drawing routine. Figure 9.21 contains the implementation of the *SetPrintDC* function. One line, the call to the **CGraphicList** *Draw* routine, does the actual printing. The other lines are used for scaling and for issuing the required operating system function calls.

```

void CSGEPrintCtrl::SetPrintDC(OLE_HANDLE nNewValue)
{
    // We don't (and shouldn't!) save the device context.
    // We simply use it for printing and throw it away.
    // This could also be a method.
    // Attach the hDC to a CDC
    CDC Thing;
    CDC * MyDc = &Thing;
    MyDc->Attach((HDC)nNewValue);
    // Save current context
    MyDc->SaveDC();
    if (Model == NULL)
    {
        // Nothing to print
        return;
    }
    // Set up DOCINFO structure
    DOCINFO MyDoc;
    MyDoc.cbSize = sizeof(DOCINFO);
    MyDoc.fwType = 0;
    MyDoc.lpszDatatype = NULL;
    MyDoc.lpszDocName = "Simple Graphics Editor";
    MyDoc.lpszOutput = NULL;
    // Get the document rectangle
    CRect MyRect;
    Model->GetRect(MyRect);
    if (MyRect.top > 0)
    {
        MyRect.top = 0;
    }
    if (MyRect.left > 0)
    {
        MyRect.left = 0;
    }
    // Get the horizontal and vertical pixels per inch for scaling
    // Each dimension is scaled separately
    // For a 600 DPI printer, PixX and PixY will get the value 600.
    long PixX = MyDc->GetDeviceCaps(LOGPIXELSX);
    long PixY = MyDc->GetDeviceCaps(LOGPIXELSY);
    // Start a document and a page
    MyDc->StartDoc(&MyDoc);
    MyDc->StartPage();
    // This stuff MUST DEFINITELY follow StartDoc and StartPage calls
    // MM_ANISOTROPIC allows independent scaling of X and Y dimensions.
    MyDc->SetMapMode(MM_ANISOTROPIC);
    // The WindowExt and ViewportExt provide a scaling function
    // WindowExt is the divisor, ViewportExt is the multiplier
    // WindowOrg and ViewportOrg provide a translation of coordinates
    // WindowOrg is the subtractor, ViewportOrg is the adder
    // Without this scaling, most printers would print 1/3 to 1/6 the
    // proper size.
    MyDc->SetWindowOrg(0,0);
    MyDc->SetWindowExt(100,100);
    MyDc->SetViewportOrg(0,0);
    MyDc->SetViewportExt(PixX,PixY);
    // Call the model drawing function
    Model->Draw(MyDc,MyRect,MyRect,0);
}

```

```
// Finish the page
MyDc->EndPage();
MyDc->EndDoc();
// restore previous context
MyDc->RestoreDC(0);
// detach context from CDC
MyDc->Detach();
}
```

Figure 9.21. The Printing Routine.

9.8. Conclusion

As our examples show, serializers are simple components that perform simple operations. It would have been possible to implement these operations directly in the Simple Graphical Editor. There are, however, compelling reasons for not doing so. The first of these is versatility. Using serializers, it is possible to implement many different file formats for the same editor, without complicating the design of the editor. Each of these file formats would be handled by its own Serializer. By the same token, once a Serializer is created, it can be used with several other components. The three components developed in this chapter could be used with both the SGE and the SGE Background Editor. By implementing the serializers as separate components, we provide file handling, clipboard access, and printing to both components at the same time.

It is probably most tempting to incorporate the clipboard and printing facilities into the SGE. These facilities are simple and easily implemented. The reason we have chosen to separate them into separate components is *Isolation of Function*. Component-level programming provides a means for completely isolating a facility to eliminate unwanted coupling between features. Other than sharing the same base object, there should be no coupling between drawing, file I/O, clipboard operations, and printing. By isolating these

facilities into separate components, we guarantee that there are no “sneak paths” between these features.

9.9. Exercises

1. Create a text-file serializer that can be used to read the entire contents of a text file into a string. The component will have two properties, *FileName* and *Contents*. *FileName* will be persistent. *Contents* will not be persistent, and will be run-time only. Once *FileName* has a value, the file can be read by reading the *Contents* property, and can be written by writing the *Contents* property. When the file is written, it must be completely replaced by the new value of *Contents*.
2. Modify the component of Exercise 1 to permit incremental reads and writes. Add a property *BlockSize* to determine the number of bytes read for each *Read* operation. Add two methods *StartRead* and *StartWrite* to initiate incremental reading and writing, and a third *StartBlockMode*, to return to reading and writing the entire file at once. Block mode will be the default state. The *StartRead* and *StartWrite* functions initiate incremental read and write modes. When in incremental read mode, reading the value of *Contents* reads the next block. When in incremental write mode writing a new value to *Contents* will append the value to the end of the file. In incremental read mode, writing the value of *Contents* is illegal, and will cause an event to be fired. In incremental write mode, reading the value of *Contents* is legal, and will retrieve the last string assigned to *Contents*. The current contents of the file is erased when *StartWrite* is called.

3. Modify the component of Exercise 2 to permit random access reads and writes. The design of the component, including any new properties and methods, is your responsibility.