CHAPTER 5

The Photo Editor Application

Unified Process:
Elaboration Phase and Third Iteration

This chapter discusses the third iteration of the project, which corresponds to the elaboration phase of the Unified Process. In this phase the goals to be achieved are to provide an architectural baseline and to formulate a project agreement with the customer to further pursue the project.

The *architectural baseline* implements a working application with limited functionality. The implemented features in our example include exception handling, loading and saving images, and basic image operations. The project agreement that is signed at the end of this phase should include the time frame, equipment, staff, and cost. This leads to the goals that are set for this iteration:

- The vision and business case are updated.
- The requirements are refined and analyzed.
- The risk assessment is updated.
- A detailed project plan is developed.
- The executable program, implementing the architectural framework with UML documentation, is developed.
- The project agreement is signed by the stakeholders to continue the project.

Most of the milestones are based on the work products that were started in the inception phase. Based on those work products, the work breakdown into the five core workflows is as follows:

- Requirements: Refine the requirements and system scope.
- Analysis: Analyze the requirements by describing what the system does.

- Design: Develop a stable architecture using UML.
- Implementation: Implement the architectural baseline.
- Test: Test the implemented architectural baseline.

In the elaboration phase the main focus lies on the requirements and analysis workflows. But as explained in Chapter 2, all five core workflows are to be worked on in all phases. This means that in this iteration we spend a considerable amount of time in the design, implementation, and test workflows as well.

5.1 The Refined Project Vision and Business Case

The refined vision is based on the analysis discussed in Chapter 4 and adds new information.

The goal, or vision, for this project is to provide software components to a printing and embossing business that allows online ordering of prints and other items, such as cups, postcards, T-shirts, and mouse pads, that can be customized with digital images.

To accomplish this task, the user must download a photo editor application that is then installed on the customer's computer. After installation the user can alter images, adding graphics, text, and special effects. Users can then save the altered images on their local computers and later upload the images via the Online Photo Shop's Web site to order prints or customized products.

The new software will open the business to a new and larger customer base, allowing for future growth.

5.2 Refined Requirements for Online Photo Shop

5.2.1 Refined Use Case Diagram

The first task in refining the requirements is to reevaluate the use case diagram. Based on the initial requirements, analysis, and design decisions described in Chapter 4, we update the use case diagram as shown in Figure 5.1.

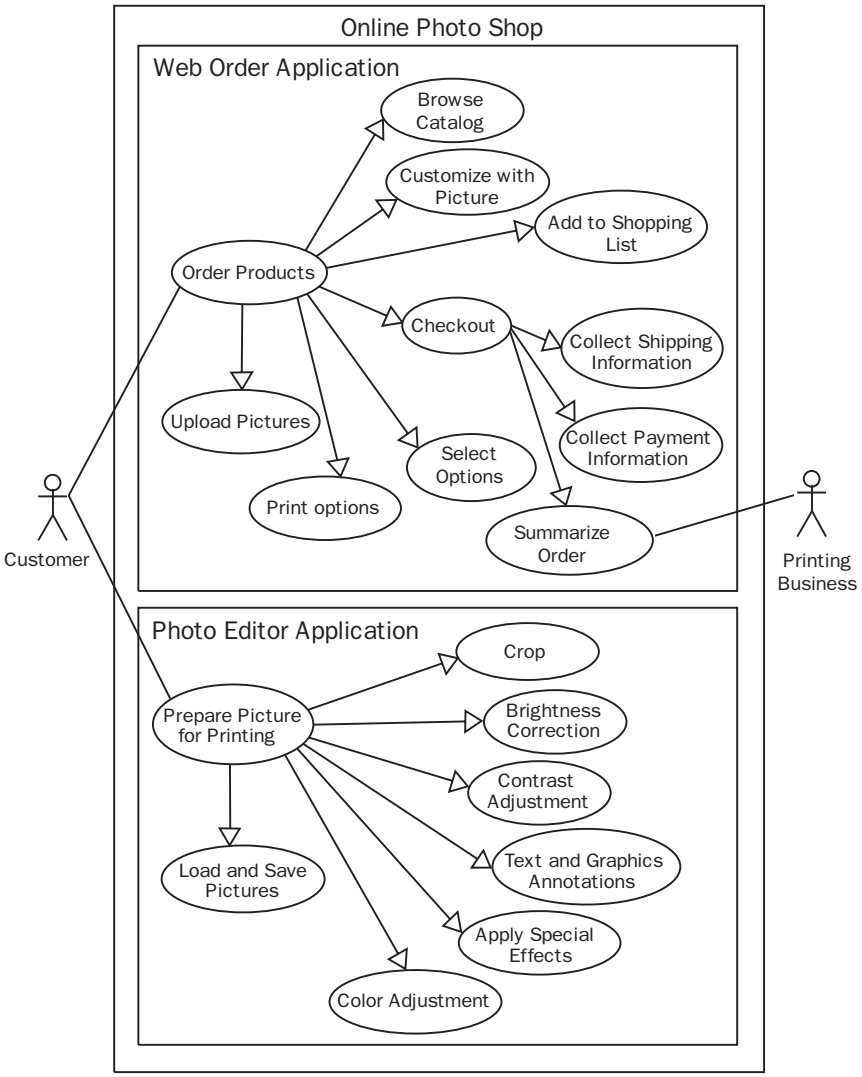

Figure 5.1 Refined Use Case Diagram for Online Photo Shop

The refined use case diagram clearly distinguishes the different parts of Online Photo Shop, as discussed in Chapter 4. Online Photo Shop provides a Web-based interface for ordering products, whereas the photo editor application is used to edit pictures on the local computer. Because of that,

the photo editor and the Web order application are in different processes. This also means that they are two independent applications and have no anticipated dependencies. In addition, the new diagram shows additional use cases for loading and saving pictures in the photo editor, as well as a use case for uploading images to the Web application.

5.2.2 Refined Requirements List

The next task is to update the requirements. In addition to adding the new requirements, it is important to put more details into the requirements. These extra details include pre- and postconditions, functional descriptions, and constraints. Refinement of requirements will not be finished in this iteration, but the goal is to put all details of the use cases known at this point into the requirements to support the following activities:

- Negotiating the deliverable functionality with the customer, including the constraints
- Analyzing the requirements and developing a good design for the implementation
- Planning the project and providing reasonably accurate effort estimates (the effort estimates will be refined throughout development)
- Enabling project management to develop an accurate contract agreement with the customer

Table 5.1 shows the refined requirements for the current iteration.

For the remainder of the book, we show only the requirements that are to be implemented in the described iteration. The alternative would be to refine all the requirements in the first construction iteration, but we have not chosen that approach for several reasons. First, it is easier to focus on the functionality that needs to be implemented in the specified iteration. Second, if requirements must be changed because of design or implementation issues, the changes usually will affect only the current iteration's requirements, and the requirements in later iterations can take all these changes into account from the beginning. (If complete planning is done up front, it is like the Waterfall model. Changes in later phases could trigger many changes.) For reference, the complete list of refined requirements can be found in the Requirement Specification XML file of each chapter on the accompanying CD.

The refined requirements include more details on the actual use case functionality and consider known constraints. In every iteration, we refine

Table 5.1 Refined Base Requirement Description

Requirement	Type	Summary
F:photo_editor	Functional	Customers shall be able to perform basic photo post-processing on their digital images. The GUI shall be similar to the GUI model provided in the file PhotoEditorGUI.vsd. The GUI model shows the screen layout and the approximate position of the necessary buttons. The buttons to be used are not defined. The buttons should be chosen so that they can be used intuitively.
F:error_handling	Functional	Errors shall be reported via exceptions. The exception-handling mechanism is based on the Microsoft Exception Management Application Block (described in detail later in this chapter) and needs to be extended to show error messages in a window on the screen. Every exception shall be displayed on the screen within a dialog window. A meaningful error message shall be displayed. For fatal errors, from which the application is not able to recover, the exception is written into the event log and the application is shut down.
F:picture_load_and_save	Functional	It shall be possible to load pictures into the photo editor application, alter them, and save them either to the same or to a new file. An error message is displayed if the image format is not known (see also image_format).
F:image_crop	Functional	If an image is loaded, then the photo editor shall provide functionality to select a region of interest and crop the image to that region.
F:image_rotate	Functional	If an image is loaded, the photo editor shall be able to rotate an image by 90 degrees in both directions.
F:image_flip	Functional	If an image is loaded, the photo editor shall be able to flip an image vertically and horizontally.
F:image_format	Functional	The supported image formats are JPG and GIF. The supported image sizes range from 100 pixels to 2,048 pixels for both width and height. An error is reported if the image format is not known.
C:platform_os	Constraint	The target platform operating system is Windows XP Professional. No special hardware requirement is necessary.

the requirements that are to be implemented in that iteration. This allows us to choose the best possible design and therefore the best possible implementation for the system.

5.3 Analysis of the Photo Editor Requirements

Next, we analyze the refined requirements. The goal in the analysis workflow is to describe what the system is supposed to do without defining how it does it (the time to define how the system is expected to be implemented is in the design workflow). For example, the `photo_editor` requirement key defines what the graphical user interface should look like without describing how it is implemented.

In this book, the analysis and design workflows are done *just in time,* meaning they are done within the iteration that implements the corresponding requirements. This approach is chosen because the team members are highly experienced programmers in the domain and technology we are using. Because of that, the effort estimates are based solely on the requirements and depend heavily on the experience of the team members. In addition, this approach provides a clearer structure for this book in describing the analysis, design, and implementation of the features within the chapter in which they are to be implemented. However, if a project has less-experienced team members or an unknown domain, it might be necessary to analyze all the requirements in more detail up front and to develop the complete design in the elaboration phase in order to provide a reasonable estimate. The estimates are the basis of the project agreement.

In the following subsections, we analyze, design, implement, and test the requirement keys implemented in this iteration and the requirement keys that define dependencies between the system's modules.

5.3.1 The `photo_editor` Requirement

The photo editor application will run in its own namespace. For this iteration, the photo editor application will provide basic functionality. The appearance of the GUI is defined in Figure 5.2 as a guideline.

The figure shows the screen layout and the controls used. It does not define the exact appearance or shapes of the resources that are used to implement the GUI. We're using a toolbar menu containing the items

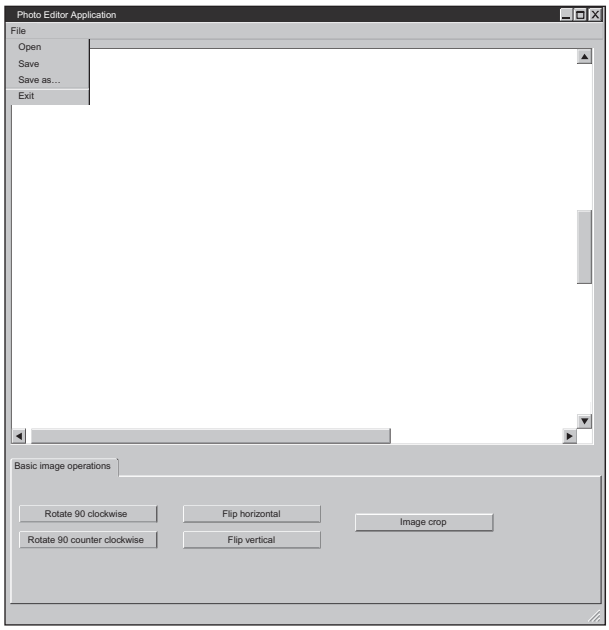

Figure 5.2 The Photo Editor GUI

Open, Save, Save As, and Exit. For the image-processing functionality, a `TabControl` with buttons is provided. The idea is to group related functionality on one tab and to provide other tabs for other groups of functionality.

5.3.2 The `error_handling` Requirement

Errors are reported via the exception-handling mechanism. Various messages will be available to identify the error that led to the message. The errors will be reported via message boxes and will contain meaningful error messages.

5.3.3 The `picture_load_and_save` Requirement

Users will be able to load and save rectangular pictures. The load and save option is part of the File menu, as shown in Figure 5.2. Users can browse for files or specify a file manually, and we use the standard Windows Forms Open/Save File dialog.

5.3.4 The `image_crop` Requirement

Image crop allows users to extract a rectangular part of the image. When the Crop button is pressed, a dialog window opens and the user is asked to type in the width and height of the cropped image. The image is cropped to the size specified by the user if the size specified is smaller than the actual image; otherwise, it is ignored. The cropping is done in a way that the midpoint of the original image is the midpoint of the cropping rectangle defined by the user. The area outside the cropping rectangle is deleted, and the new image in the new size is shown. If the defined cropping rectangle is larger than the image, no cropping is done.

5.3.5 The `image_rotate` Requirement

The image can be rotated in 90 degree steps clockwise and counterclockwise. The image will be shown in the same location and with the same midpoint as the original image.

5.3.6 The `image_flip` Requirement

By pressing the image flip buttons, users can *flip* the image either horizontally or vertically. By "flip," we mean mirroring the image. The position of the midpoint of the image stays the same. The image will be shown in the same location and with the same midpoint as the original image.

5.3.7 The `image_format` Requirement

The standard formats for pictures are supported. If an image with another format is selected, an error message will be shown.

5.3.8 The `platform_os` Requirement

All development and testing are done on Windows XP computers. Even though the application might be able to run on other versions of Windows operating systems, this will not be supported.

5.4 Design of the Photo Editor Application

We have analyzed the requirements and have defined what the system is supposed to do. The next step is to decide how the system will be implemented—in other words, to design the system.

A standard way to document software architectures is to use the Unified Modeling Language (UML). We mainly use UML class diagrams to define the structure of the photo editor software system. The design shows classes, attributes, operations, and the relationships among them. If you have a working knowledge of UML, you may want to skip the following brief introduction to UML class diagrams and go immediately to the provided class diagram of the photo editor in section 5.4.2.

If you're not familiar with UML, we offer a very short introduction to the class diagram in UML. This introduction is not intended as a tutorial on UML class modeling but rather aims to build a common vocabulary to help you understand the design of the system throughout the book. Therefore, the goal of this section is to give you just enough details to understand the class diagrams developed for the application. For more detailed information on UML and class diagrams, see the references section in Chapter 4.

5.4.1 Brief Introduction to UML Class Diagrams

The basic element of a class diagram is a class, which you model using a rectangular box. This box can contain the class name by itself and can also contain the class attributes and the operations defined by the class. Figure 5.3 shows sample class definitions.

Figure 5.3A shows a class that has sample attributes and an operation definition. The `sampleOperation` call returns a `uint` value, which is shown after the operation name and separated from it by a colon. In addition, the visibility of attributes and operations is shown in the class model by means of the provided prefix. Table 5.2 lists the supported visibility prefixes and their meanings. As you can see, in `SampleClass` the attributes are `private` and `protected`, whereas `sampleOperation` is defined as `public`.

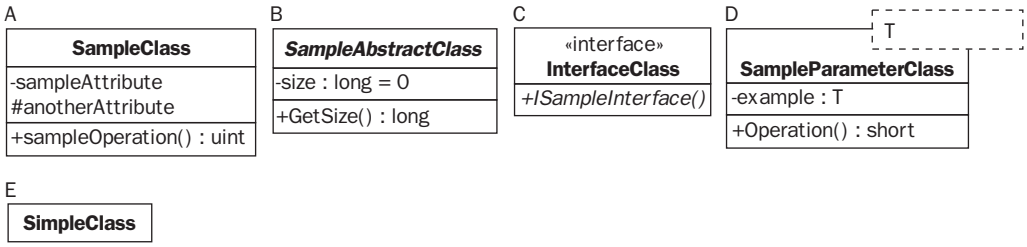

Figure 5.3 Sample Class Models

Table 5.2 Class Diagram Visibility Prefixes

Prefix	Visibility
- (hyphen)	`private`: Can be seen only within the class that defines it
#	`protected`: Visible within the class that defines it and by subclasses of the defining class
+	`public`: Visible to any class within the program

The sample class shown in Figure 5.3B is an *abstract* class. Abstract class names are shown in italics to distinguish them in the class diagram. Note, too, that the public operation `GetSize()` returns a `long` value, which actually is the `private` attribute size of `SampleAbstractClass`. The `private` attribute size is initialized to 0 (zero) and of type `long`.

An abstract class without any implementation in UML is called an *interface* class. Figure 5.3C shows a sample interface class identified by the stereotype `<<interface>>` above the class name. Interface classes provide no implementation, and this means that no section for attributes is needed in the class model. The operations in an interface class are shown in italics because they do not provide any implementation and must be implemented by the derived classes.

UML also lets you define parameterized classes. In C++ this feature is known as a *template* class (C# will provide support for parameterized or generalized classes in a coming version). Parameterized classes are identified by the parameter definition in the upper-right corner of the class model. Figure 5.3D shows a sample parameterized class.

The class shown in Figure 5.3D is a simple class showing only the class name. This type of model is usually used in overview diagrams of large systems.

To model the system, we must define relationships between the classes. The next section introduces the class relationship models in UML.

Class Relationships

Figure 5.4 shows the basic dependency principles among classes.

Figure 5.4A shows a *dependency* between `Class1` and `Class2`. The dependency indicates that changes to `Class2` could trigger changes in

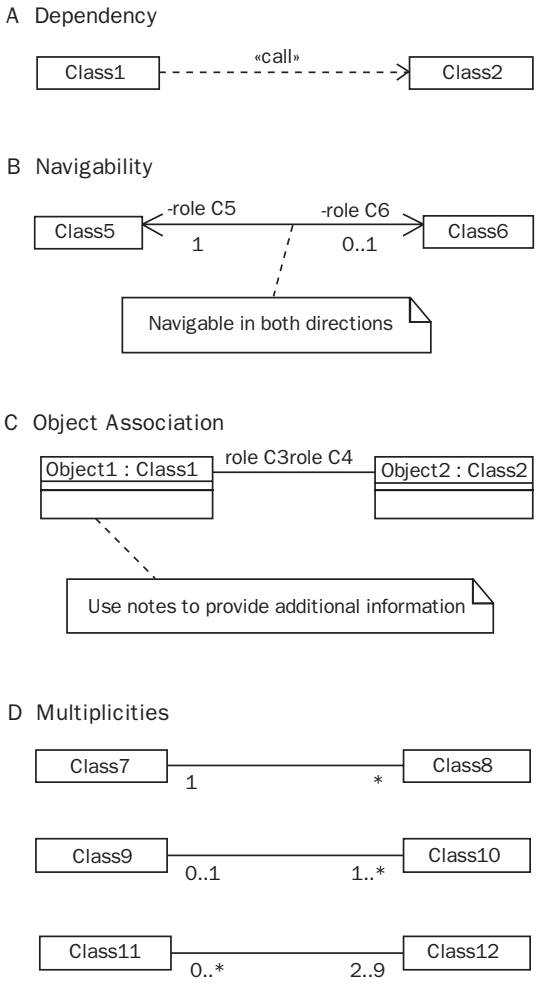

Figure 5.4 Class Dependency

`Class1`. Figure 5.4B shows navigability in both directions. *Navigability* shows the directions (unidirectional or bidirectional) in which the program can traverse the class tree. In the example, `Class6` "knows" about `Class5`, and vice versa. The navigability diagram also shows how to use *notes* to add information to the class diagram if necessary. At both ends of the navigation relationship, you can show any multiplicity of the relation (see Figure 5.4D) for more detail).

Figure 5.4C shows an association relationship between two class instances (or objects) in addition to a note. The class instances are identified by the prefix object and the class name, which are separated by a colon and underlined. Both ends of an association can be labeled. The label is called a *role name*.

Another important part of class diagrams is the ability they give you to express *multiplicities,* as shown in Figure 5.4D. Multiplicity is expressed by a number on each side of the dependency connection. If no number is supplied, then the default, which is 1, is assumed. The meanings of the multiplicities numbers are as follows:

1	Exactly one
*	Many (zero or more)
0..1	Optional (zero or one)
1..*	One or more
0..*	Optional, one or many
2..9	Example of an m..n relationship; in this case, a two to nine dependency

Now let's look at the more advanced class dependencies, as shown in Figure 5.5.

Aggregation (shown in Figure 5.5A) is a *part-of* relationship. For example, a point is part of a circle. This sounds very simple, but then what is the difference between dependency and aggregation? Aggregation can be seen as a placeholder. In other words, you can use aggregation as an indication of a relationship without specifying the details of the relationship. This is possible because UML defines very little of the semantics of aggregation.

In addition to aggregation, UML offers a more defined variation called *composition*. Composition implies that the part-of object belongs only to one whole, and the part-of object's lifetime responsibility is with the object that contains it. Figure 5.5B shows that `Class2` is part of `Class1` and that `Class1` is responsible for creating and deleting objects of `Class2`.

Another dependency defined in UML is the generalization, shown in Figure 5.5C. *Generalization* can be seen as an *is-a* relationship. For example, a diesel engine is an engine, and a gasoline engine is an engine. This holds true if "engine" is defined as the commonalities of diesel and gasoline engines. Every diesel engine and every gasoline engine can then be defined by deriving from the general `engine` class and adding the specifics according to the engine type.

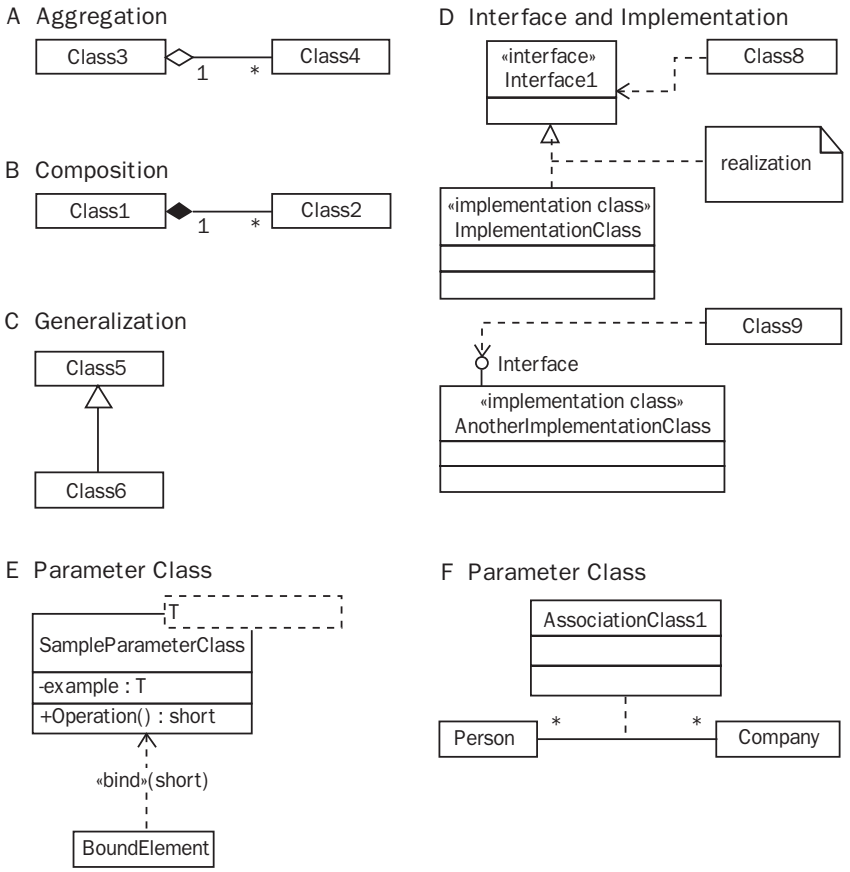

Figure 5.5 Advanced Class Dependencies

Interfaces and their implementation can be modeled within UML in two ways, as shown in Figure 5.5D. The first technique is to define an interface class that does not contain any implementation. This class is then connected to the implementation class via the *realization* dependency. In the example, `Class8` calls on the `Interface` method, which is implemented in the `ImplementationClass`. The second way to show an interface in a UML diagram is by using an `Implementation` class that shows the provided interface with the lollipop representation. In the example, `Class9` calls the `Interface` method of `AnotherImplementationClass`. In the first technique, you can explicitly show the interface definition and the implementation, whereas the second technique shows only the implementation class.

The use of parameterized classes is called *bound elements* and is known to C++ programmers as template classes. In Figure 5.5E, `BoundElement` is bound by a `short` parameter to `SampleParameterClass`. Parameterized classes cannot be extended by the bound class. `BoundElement` can use completely specified types from the parameter class. The only thing added to the parameter class is the restricting type information.

Association classes are used to add an extra constraint to a relation of two objects. This is shown in Figure 5.5F. Only one instance of the association class can be used between any two participating objects. In the example, a `Person` is working for at most one `Company` at a time (at least in theory). An association class is used to keep information about the date range over which each employee is working for each company. Because persons sometimes switch jobs, this information should not be kept within the person's class directly but rather in an association class.

In object-oriented programming, you usually try to limit dependencies to the absolute minimum. In this way, changes made to one class do not trigger endless changes to other classes. Necessary dependencies are usually kept on the interface level.

5.4.2 Design of the Photo Editor Application

The design of the photo editor application is done in Microsoft Visio. The decision to use Visio was made because in addition to providing tools to generate UML diagrams, Visio lets you generate code from class models. Also, Visio supports extracting class models from existing code, a capability that can be very useful when you're reengineering poorly documented software. Unfortunately, Visio has no support for the creation of XML documentation from the UML models. Therefore, during document review we must trace requirement keys manually through the design process.

The class diagram shows that the main application class, called `PhotoEditorForm`, is derived from the `Windows.Forms` class. The `Windows.Forms` class provides Windows-related functionalities that are used by the photo editor. In addition, `PhotoEditorForm` creates an instance of `ExceptionPublisher`, which is derived from the `IExceptionPublisher` interface provided by the Microsoft application block. In addition, an instance of the `Picture` class is created. The `Picture` class instance holds the loaded image and provides the basic image-manipulating operations (such as rotating and flipping).

Figure 5.6 shows the class diagram for the photo editor application in this iteration.

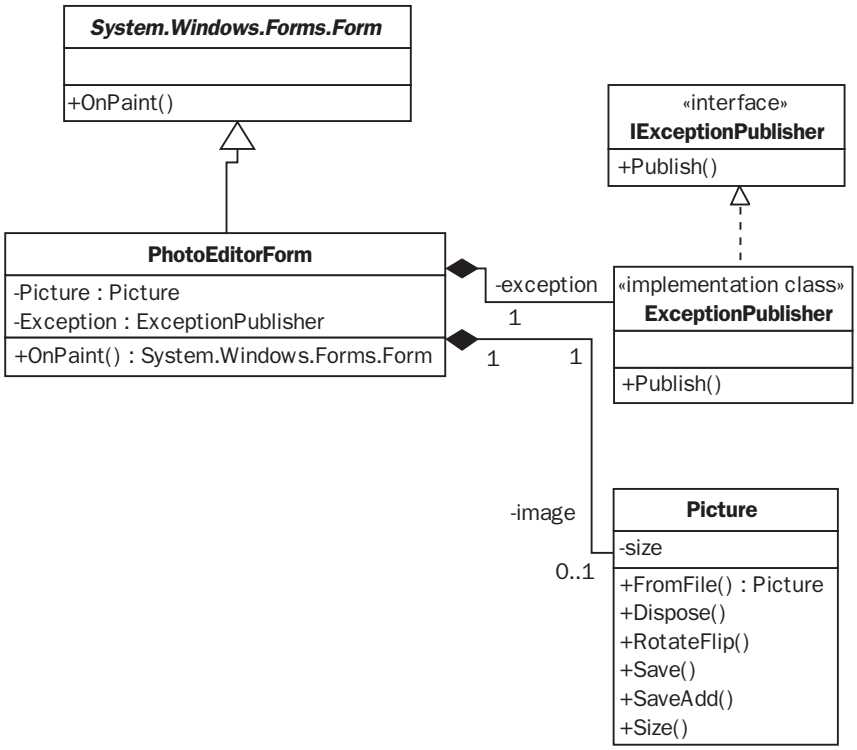

Figure 5.6 The Photo Editor Class Diagram

5.5 The Detailed Project Schedule

As mentioned earlier, it is important to track the project and update the plan constantly. In this iteration we develop a more refined project schedule, as shown in Figure 5.7. The project schedule is kept in Microsoft Visio as a Gantt chart, showing the various tasks, their dependencies, and the project milestones. Online Photo Shop is the overall project, and we indent all other tasks to show that they are defined as subtasks.

For easier readability, the requirements are grouped into blocks of related functionality. The name of the developer who is responsible for a block is shown in the resource column. Most of the blocks contain dependent or very small functionalities (or both). By combining these items into blocks, we make it easier to do tracking on a daily instead of an hourly basis.

Detailed Project Schedule: Online Photo Shop

ID	Task Name	Start	Finish	Duration	Resource Names
1	Milestone 1 (progress review)	11/11/2002	11/11/2002	0d	
2	Milestone 2 (progress review)	11/25/2002	11/25/2002	0d	
3	Milestone 3 (progress review)	12/19/2002	12/19/2002	0d	
4	**Online Photo Shop**	**10/10/2002**	**1/31/2003**	**82d**	
5	photo_editor	10/10/2002	1/7/2003	64d	GL, KC
6	order_online	10/10/2002	1/7/2003	64d	TM
7	error_handling, picture_load_and_save, image_format, image_crop, image_rotate	10/22/2002	11/8/2002	14d	GL
8	Image_flip	11/4/2002	11/8/2002	5d	Reader
9	image_brightness, image_contrast, image_color,	11/11/2002	11/20/2002	8d	GL
10	image_graphics_annotations, image_text_annotations	11/11/2002	11/20/2002	8d	KC, Reader
11	order_products, product_browse, product_options, product_customize	11/11/2002	11/20/2002	8d	TM, Reader
12	Risk Buffer	11/20/2002	11/22/2002	3d	
13	image_processing_direct_3d, editor _optimizations	11/25/2002	12/13/2002	14d	GL
14	image_graphics_special_effects, image_special_effects	11/25/2002	12/13/2002	14d	KC, Reader
15	product_shopping_cart, order_checkout, checkaout_shipping, checkout_payment, checkout_summerize.	11/25/2002	12/13/2002	14d	TM, Reader
16	Risk Buffer	12/13/2002	12/18/2002	4d	
17	editor_system_test	12/19/2002	1/20/2003	22d	GL, TM, KC, Reader
18	Risk Buffer	1/20/2003	1/31/2003	10d	

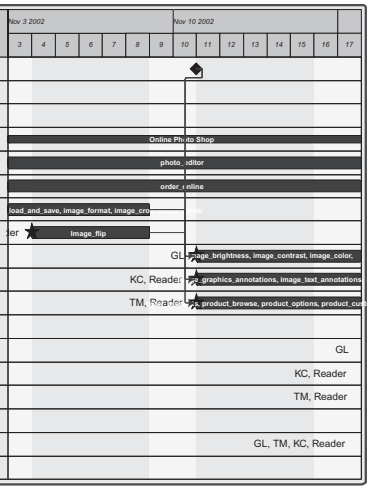

Figure 5.7 Detailed Schedule

In addition to the tasks themselves, including their estimated duration, other milestones are defined and connected to the related tasks (as indicated by the lines that go from the milestones to the tasks). The milestones are review points at which the status of the project is reviewed and necessary adjustments are made. In the case shown in Figure 5.7, the milestones coincide with the end (or the beginning) of the iterations. This works well if the requirements are broken into small enough pieces (no longer than two weeks' worth of effort). If you cannot do this, you must either break the tasks into smaller pieces or define interim milestones.

From the detailed schedule, you can see that even though the iterations are described in a serialized fashion, iterations are actually worked on in parallel. This practice shortens the overall duration of the project. In our case, working on development tasks in parallel works very well because the photo editor application and the online shop have no anticipated dependency. The schedule shown is constantly refined and updated throughout the development cycle. Also, we have added risk buffers to the schedule. These buffers give the project additional time to react to unforeseen problems. Nevertheless, the risk buffers should not be used for adding new requirements or implementing new features.

Tracking is done in small meetings. We hold one weekly meeting to discuss problems, ideas, and management-related items; in addition, *stand-up*

meetings (which last about 12 minutes) are held every other day for frequent status updates (no discussions are allowed at these meetings). Stand-up meetings often lead to the calling of a regular meeting at another time so that team members can quickly discuss problems found.

From the Gantt chart we developed in Visio, we can export a variety of diagrams, such as a timeline diagram. These exported diagrams are very useful for reports to the customer or upper management (assuming that the plan is kept up-to-date).

5.6 Implementation of the Photo Editor Application

After all the ground work is set, the implementation can start. To begin the implementation, a project workspace must be set up and added to configuration management. As mentioned in Chapter 4, the configuration management administrator is the person who should set up the directory structure. Based on the directory structure (see Figure 4.3), a solution is created that will contain all the subprojects of the software developed during the course of this book.

To create an empty Visual Studio solution, open Visual Studio.NET (see Figure 5.8). Then select Visual Studio Solutions and Blank Solution, and name the solution `Photo Editor`. The solution should be crated in the

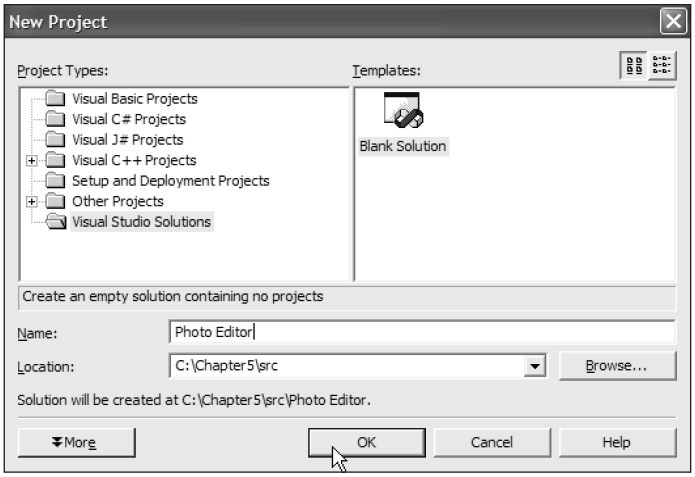

Figure 5.8 Creating an Empty Solution

src directory, so browse to src in your project directory tree and click on the OK button. The Visual Studio solution will be created.

5.6.1 The Basic Photo Editor Application

Now that we have an empty Visual Studio solution, we can add other projects to it.

Creating a Basic Application

We will develop the GUI and most of the functionality of the photo editor application in C# using Microsoft Visual Studio 2003. The first implementation step is to create a Windows application project; we will then extend it to provide exception handling and image loading, saving, and displaying. The application is created as a project within the Photo Editor solution.

Therefore, with the Photo Editor solution opened in Visual Studio, click on the File menu and then Add Project. Then choose New Project. In the dialog window that appears, choose Visual C# Projects and Windows Application. In the Name field, type `Photo Editor Application`. Click on OK, and Visual Studio.NET creates a new project within the Photo Editor solution.

The Microsoft wizard in the solution generates three files: `App.ico`, `AssemblyInfo.cs`, and `Form1.cs`. You can explore all three files by double-clicking on the file name in Solution Explorer. Obviously, `App.ico` is the icon associated with the photo editor application and can be customized. The `AssemblyInfo.cs` file contains assembly-related information such as version and binding. The version information is updated before all releases (including intermediate releases); binding will be discussed when the product is deployed to the customer in the final release. The more interesting file that was generated is `Form1.cs`. This file contains the form design and the code for the photo editor application. To see the generated code, go to the `Form1.cs [Design]` view, choose the form, right-click on the form, and choose View Code.

The top of the file shows various namespaces that are used in the application (similar to the kind of information you see in `#include` statements in C++). These `using` statements define shortcuts for the compiler to use in resolving externally defined namespaces. `Extern` in this case relates to objects that are not defined in the project's namespace. If a call to a method cannot be resolved, the compiler will try to resolve the call by checking the namespaces defined by the `using` statement.

Listing 5.1 shows the namespaces used by the code generated by Visual Studio.NET. The `System` namespace contains the basic .NET Framework types, classes, and second-level namespaces. In contrast, the second- and third-level namespaces contain types, classes, and methods to support various kinds of development, such as GUI, the runtime infrastructure, .NET security, component model, and Web services development, to name only a few categories.

In the photo editor application, you can see that several second-level namespaces are automatically included. The `System.Drawing` namespace, for example, provides rich two-dimensional graphics functionality and access to Microsoft's GDI+ functionalities. (GDI+ is explained in more detail in Chapter 6.) For the remainder of this chapter, we use GDI+ to provide a memory location where the image can be stored and then displayed; in addition, we use some GDI+ methods for image operations.

The `System.Collections` namespace holds collections of objects, such as lists, queues, arrays, hash tables, and dictionaries. In addition, `System.ComponentModel` implements components, including licensing and design-time adaptations. For a rich set of Windows-based user interface features, we also include the `System.Windows.Forms` namespace. Last but not least, the `System.Data` namespace lets us access and manage data and data sources. For more information on the namespaces provided by the .NET Framework, please refer to the MSDN help.

Listing 5.1 Using Externally Defined Namespaces

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
```

The next section in the source file defines the namespace for the application, the basic application classes, and the methods. Listing 5.2 shows the source code that is created.

Listing 5.2 The Photo Editor Namespace

```
namespace Photo_Editor_Application
{
  /// <summary>
```

```
/// Summary description for Form1.
/// </summary>
public class Form1 : System.Windows.Forms.Form
{
  /// <summary>
  /// Required designer variable.
  /// </summary>
  private System.ComponentModel.Container components = null;

  public Form1()
  {
    //
    // Required for Windows Form Designer support
    //
    InitializeComponent();

    //
    // TODO: Add any constructor code after
    // InitializeComponent call
    //
  }

  /// <summary>
  /// Clean up any resources being used.
  /// </summary>
  protected override void Dispose( bool disposing )
  {
    if( disposing )
    {
      if (components != null)
      {
        components.Dispose();
      }
    }
    base.Dispose( disposing );
  }

  #region Windows Form Designer generated code
  /// <summary>
  /// Required method for Designer support - do not modify
  /// the contents of this method with the code editor.
  /// </summary>
  private void InitializeComponent()
```

```
    {
      this.components = new System.ComponentModel.Container();
      this.Size = new System.Drawing.Size(300,300);
      this.Text = "Form1";
    }
    #endregion

    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    [STAThread]
    static void Main()
    {
      Application.Run(new Form1());
    }
  }
}
```

First, we define the namespace `Photo_Editor_Application`. This is the namespace that refers to all the classes, types, and methods defined by the photo editor application. Next, class `Form1` is defined as `public` and is derived from `System.Windows.Forms.Form`. This class implements the application window, which is called `Form1`, at least for now. The first property defined in the `Form1` class is declared `private`, named `components`, and defined to be of type `System.ComponentModel.Container`. The value is set to `null` to indicate that no initialization has yet been done.

Next, we define the public constructor of `Form1`. The implementation first calls the `InitializeComponent` method created by Visual Studio Designer. This method takes care of the necessary Windows Forms property initialization. You should not modify the Designer-generated part of the code directly, but you can do so through the properties window of `Form1.cs[Design]`.

After the constructor, the dispose method is defined. The `Dispose()` method is used to free system resources if they are no longer needed. Even though Visual Studio.NET provides garbage collection for allocated memory, we must explicitly delete other system resources at the time they are no longer used. Resources other than memory should be disposed of in order to keep the resource footprint (memory, disk space, handles, and so on) as small as possible. In the example, the components are disposed of if they were allocated (not `null`); to do this, we call the base class's dispose method.

Next comes a block specified with the `#region` and `#endregion` keywords. This block allows the developer to write code that can be expanded or collapsed within the Visual Studio.NET development environment. In the example, the `#region-#endregion` block encloses the initializing method of the Designer-generated form, as described earlier, and this code should not be altered. You can collapse the code by pressing the "-" symbol, or expand it by selecting the "+" symbol next to the keyword. Usually the IDE provides the expand-collapse feature automatically for multiline comments, class definitions, and method definitions, to name only a few. The developer can define additional collapsible and expandable regions. The region statements can be nested. In that case, the `#endregion` matches the last defined `#region` statement that has not yet been matched.

The final part of the code defines the `static main` entry point for the application. The application is defined to be running in a single-threaded apartment (STA), and the main entry point then creates and runs an instance of `Form1`.

The next step is to change the output directory of the compiler to the `bin` and `bind` directories. As mentioned in Chapter 4, the `bin` directory holds all assemblies necessary to run the photo editor application in release configuration, whereas the `bind` directory holds the same files but compiled in debug configuration. To change the output directory, choose the project in the Solution Explorer window and then choose Project | Properties | Configuration Properties | Build | Output Path; change the path to the `bin` directory (for release configuration) and `bind` (for debug configuration).

Before you check the source files into the configuration management system, you need to make some additional changes. For easier readability, maintainability, and understanding, it is worthwhile to rename some of the generated source files to more meaningful names. We do this before checkin because renaming files already under configuration management is not always easy.

Therefore, we change the name of the application source file from `Form1.cs` to `PhotoEditor.cs`. We do this by right-clicking on the file name `Form1.cs` in the Solution Explorer window and then going to the Properties window below Solution Explorer and changing the name in the File Name field. After the name is changed, we adjust other properties of `PhotoEditor.cs[Design]`. We select the corresponding tab and click on the form. We change the Text field to `Photo Editor Application`, and change the (Name) field to `PhotoEditorForm`.

To finish the cosmetics, click on the photo editor form and choose View Code (by right-clicking on the form and choosing the option). Change `Form1()` to `PhotoEditorForm()`, as shown in Listing 5.3.

Listing 5.3 `PhotoEditorForm` Creation

```
static void Main()
  {
    Application.Run(new PhotoEditorForm());
  }
```

Before you put the project into the source control system, make sure that it compiles. Go to the Build menu in Visual Studio, and choose Build Project (or use the shortcut by pressing Ctrl+Shift+B). If there are any errors during compilation, they will be shown in the output window below the main window. Double-clicking on the error message will open the correct source file to the approximate position of the error.

5.6.2 Using Version Control

The Visual Source Safe source control system integrates smoothly with the Visual Studio.NET IDE. There are several ways to communicate with it. The following assumes that Visual Source Safe is installed on the computer (even though working with other source control systems may be quite similar). The first possibility is to use the File | Source Control menu, which is shown in Figure 5.9.

Another method of communicating with the version control system is to configure a toolbar menu. Go to View | Toolbars | Source Control. You'll get a toolbar (shown in Figure 5.10) that can be added to the toolbar section of Visual Studio.

There is yet another way to check the project into the source control system. Right-click on the project in Solution Explorer and choose Add Solution to Source Control.

In our sample project, we add the project to the source control system by using the latter method. The system asks for the login credentials before the Visual Source Safe dialog appears, as shown in Figure 5.11.

Now we choose the name and location for storing the project in the source control system. For this project, we take the defaults suggested by Source Safe (location root and project name Photo Editor Application), so

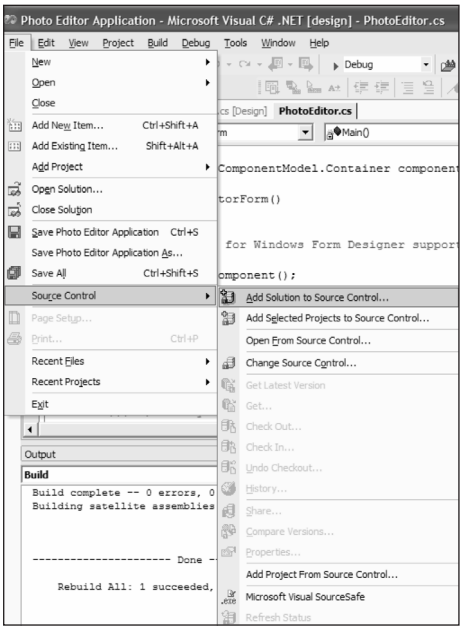

Figure 5.9 Visual Source Safe Menu

Figure 5.10 Visual Source Safe Toolbar

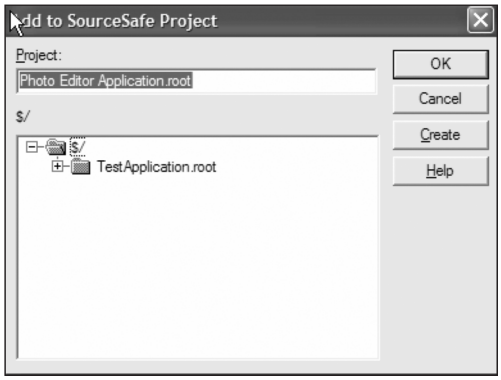

Figure 5.11 Source Safe Dialog

we just click OK. The system asks whether it should create the new project, and again we simply confirm by clicking OK. As you can see, all files in Visual Studio Explorer now have a little lock symbol next to them; this means that the files are read-only (checked in) under source control.

Before a developer can make any change to the files, he or she must check out the files to get write permissions to them. Checkout means that the developer will work on a private copy of the file. The developer works on the private copy, keeping it checked out, until the change is (partially) complete and compiled; then the file must be checked in. At checkin time the private file is added to the version control as the newest version of the file; the file becomes read-only again, and all developers on the project can see the the new file. Changes can be made only by the developer who currently has the file checked out. It is also possible to undo a checkout, thereby reverting to the original version of the file and discarding the changes made. (Nobody will ever know about the changes. Usually it is a good practice to save the changes under a different file name in a private file before undoing a checkout and losing the changes.)

Visual Source Safe provides many other useful tools, such as forcing the system to get the latest version of all files and comparing the changes in different file versions. Most of the functionality is self-explanatory. For more detailed information, you can consult the Visual Source Safe help files.

5.6.3 The Exception-Handling Application Block

The first "real" functionality that we will implement is exception management. As discussed earlier, exception handling in the photo editor application is based on the Microsoft Exception Management Application Block. You can download this application block from `http://msdn.microsoft.com/downloads/list/bda.asp`. You install the files by following the instructions on the screen. Alternatively, you can simply take the installed source files from the sample solutions provided on the CD under `Chapter5\src\Exception Management Application Block`.

Next, go to the installation directory of the application block and open the `Code` directory with a double-click in Windows Explorer. There are two versions of the exception management block installed. The first version is written in Visual Basic (`VB`), and the second version is implemented using C# (`CS`). We are interested in the C# version, so open the `CS` subdirectory and double-click on the Exception Management Application Block (CS) solution. This will open a solution with two projects: `Microsoft.`

`ApplicationBlocks.ExceptionManagement.Interfaces` and `Microsoft.ApplicationBlocks.ExceptionManagement` in Visual Studio.NET.

To use the application blocks, we need to build both projects. *But* before we actually build them, we first set the output path of both projects to the `bin` and `bind` directories of the photo editor project. To set the output path of the compiler-generated assemblies, right-click on the project in Solution Explorer. Choose Properties and go to Configuration Properties | Build | Outputs | Output Path. Make sure that the configuration of the selected project is set to Release. Choose the table entry for Output Path, and navigate to the photo editor `bin` directory. Then choose Apply. If the project is now being built in release mode, then the assemblies will be saved in the `bin` directory of the photo editor project. Do the same for the Debug configuration of both projects by specifying the `bind` directory as the output path.

After that, build the solution by going to the Build menu and choosing Build Solution (or by using one of the previously mentioned shortcuts). Then change the build configuration in the Active Solution Configuration list box and build this configuration as well. (Either use the menu bar or go to the Build menu, choose Configuration Management, and choose Debug or Release depending on the configuration you just built.) You can check the success of the build by checking the `bin` and `bind` directories for the created assemblies. You should be able to see `ExceptionManagement.dll` and `ExceptionManagement.Interfaces.dll` in both directories.

After the assemblies are built, we add references to `Microsoft.ApplicationBlock.ExceptionManagement.dll` and `Microsoft.ApplicationBlock.ExceptionManagement.Interfaces.dll` to the photo editor application project. To do this, you choose the menu item Project | Add Reference. Alternatively, you can right-click on the Photo Editor Application project in Solution Explorer and choose Add Reference. A dialog box opens. Choose Browse and navigate to the `bin` directory, where the assemblies for the Microsoft Exception Management Application Block reside. (By referencing the assemblies in the `bin` directory we are referencing the assembly that was built in release configuration. Alternatively, we could reference the debug version. But we do not intend to debug the application block, so we reference the release version.) Select the two assemblies and press OK.

Another Source Safe dialog box opens that lets you choose whether to check out the project file for editing. Because we will add references to the project, we need to update the file `Photo Editor Application.csproj` to reflect these changes. Click OK to check out the file. You can add a com-

ment to the history of the file before checkout. The rule for the photo editor application is to add checkin comments that explain the changes made. In Solution Explorer, you can now see the newly added references in the Reference section. In addition to the references, we need to add a `using` statement to the photo editor application to indicate the use of the externally defined functionality. Again, the source control will ask whether the `PhotoEditor.cs` file should be checked out; we acknowledge this by choosing checkout. The added code can be seen in Listing 5.4.

Listing 5.4 Using the Microsoft Exception Management Application Block

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

using Microsoft.ApplicationBlocks.ExceptionManagement;
```

Now that the photo editor application is aware of the exception management class, the classes and methods provided by it can be used. The provided default publisher logs the exceptions to the event log. This is not the behavior we intend for the photo editor application, so we must create a custom publisher. Before continuing, make sure that all changes are saved and the project compiles.

5.6.4 Customized Exception Handling

You create custom publishers by implementing the `IExceptionPublisher Interface`, which is defined in the `Microsoft.ApplicationBlocks. ExceptionManagement.Interfaces.dll` assembly. To keep the exception publisher code separate from the application code, add a new file to the photo editor application project. To add a file, right-click on Photo Editor Application in Solution Explorer and choose Add | New Item (see Figure 5.12).

Select Class in the dialog box and type the name `ExceptionPublisher.cs` for the file. Click on Open; this will open a new file with the added name selected. The file contains the `using system` statement and shows that it is

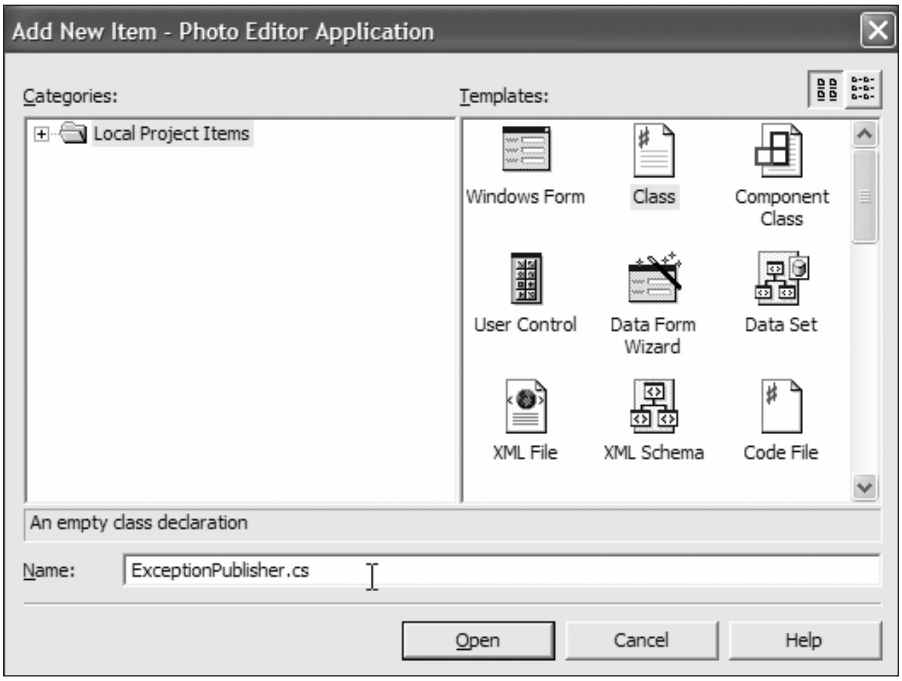

Figure 5.12 Adding the `ExceptionPublisher.cs` File

part of the `Photo_Editor_Application` namespace. In addition, a class definition for class `ExceptionPublisher` and its constructor are provided. The new file is automatically added to Visual Source Safe and is marked as checked out (marked with a red check mark next to the file name in Solution Explorer).

Next, we add the `using` statement for the Exception Management Application Block in the same way as was shown for the `PhotoEditor.cs` file. In addition, we add the `using System.Windows.Forms` statement for Windows Forms support. Then we must derive the `ExceptionPublisher` class from `IExceptionPublisher`. Therefore, we change the class definition to `public class ExceptionPublisher : IExceptionPublisher`. When we're finished typing, a stub for the `IExceptionPublisher` interface can be added automatically by pressing the Tab key (otherwise, we simply type it in as shown in Listing 5.5). The finished `Publish` method is shown in Listing 5.5. It also shows the three arguments the `Publish` method takes.

Listing 5.5 `Publish` Interface Implementation with XML Documentation

```
#region IExceptionPublisher Members
    /// <summary>
    /// Custom Publisher, displays error message on screen.
    /// </summary>
    /// <param name="exception">Exception, containing meaningful
    /// error message</param>
    /// <param name="additionalInfo">Provides additional info
    /// about the exception</param>
    /// <param name="configSettings">Describes the config
    /// settings defined in the app.config file</param>
    public void Publish(Exception exception,
      System.Collections.Specialized.NameValueCollection
      additionalInfo,
      System.Collections.Specialized.NameValueCollection
      configSettings)
    {
      // TODO:  Add ExceptionPublisher.Publish implementation
      string caption = "Photo Editor";
      DialogResult result;

      // Displays the MessageBox.

      result = MessageBox.Show( exception.Message, caption,
        MessageBoxButtons.OK,
        MessageBoxIcon.Error, MessageBoxDefaultButton.Button1,
        MessageBoxOptions.RightAlign);
    }
#endregion
```

To generate nicely formatted documentation from the source code, we add the XML description as specified in Chapter 4. You generate the documentation from the code by selecting the Tools menu in Visual Studio and choosing Build Comment Web Pages. Then select the radio button Build for the entire solution, and specify Save Web Pages in the `doc` directory of your project (in the sample solution this is `Chapter5\doc`) and click OK. The documentation is generated, and an Explorer window opens that shows the generated solution's comment Web pages. Click on the namespace to navigate down the program documentation tree and open the next lower level.

According to the requirements, exceptions are to be published via a message box displaying meaningful error messages. This leads us to the next implementation step, which is to implement the message box. The easiest way to display a message box is to use a Windows message box. The `System.Windows.Forms` namespace, which we have already added to the `using` statement section, provides the functionality to display a simple message box. Several overloaded types of message boxes are supported by the .NET Framework. The one that we use here is as follows:

```
MessageBox.Show( text, caption, buttons, icon, defaultButton, options);
```

The message box used in this example takes six parameters, which are explained in Table 5.3. For other variants of the message box, please refer to the MSDN help.

To use the customized exception publisher, the final step is to provide a configuration file. The configuration file makes the exception application block aware of the custom publisher that is to be used (this is similar to registration of the custom publisher). To add an application configuration file, right-click on Photo Editor Application in Solution Explorer and choose

Table 5.3 `MessageBox.Show` Parameters

Parameter Type	Name	Description
`string`	`text`	Text to be displayed in the dialog box, which is the error message in this case.
`string`	`caption`	Text displayed in message box title.
`MessageBoxButtons`	`buttons`	The buttons to be displayed in the dialog window. For the error message dialog window, this is just the OK button.
`MessageBoxIcon`	`icon`	This defines the icon displayed in the message box. For an error message the displayed icon is a red circle with a white x inside.
`MessageBoxDefaultButton`	`defaultButton`	The default button. In this case there is only one button displayed, so `button1` is the only and default button.
`MessageBoxOptions`	`options`	The text in the message box is right-aligned.

Add | New item. In the dialog window that opens, choose Application Configuration File and press Open. A configuration file is added to the solution. Change the configuration file to correspond with Listing 5.6.

Listing 5.6 `App.config`: The Application Configuration File

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <section name="exceptionManagement"
      type="Microsoft.ApplicationBlocks.ExceptionManagement.
       ExceptionManagerSectionHandler,
       Microsoft.ApplicationBlocks.ExceptionManagement" />
  </configSections>

  <exceptionManagement mode="on">
    <publisher assembly="Photo Editor
    Application"type=
     "Photo_Editor_Application.ExceptionPublisher"
      fileName="c:\PhotoEditorExceptionLog.xml"/>
  </exceptionManagement>
</configuration>
```

As you can see, the configuration file provides information regarding the customized publisher. The configuration entry `<publisher assembly=...` defines the name of the assembly in which the customized exception publisher is defined. The `type="..."` defines the namespace and the method name for the publisher, whereas the file name specifies the log file name to which exceptions are logged in case the defined publisher cannot be found. If an exception occurs, the Exception Manager Application Block will now know about the customized exception publisher and will call the specified exception publisher.

Make sure that the project compiles, and check in all the changes by choosing the Pending Checkins tab below the main window. When you check in a file, usually it is good practice to provide a meaningful comment. The comment for the checkin at this point might read, "Added custom exception handling using the Microsoft Exception Manager Application Block. Exceptions are published in a window on the screen." After typing the comment, choose Check In. A dialog opens if the files really should be checked in. Click on OK, and all the changes are available in the repository, visible to everybody on the team.

Now the exceptions can be used in the photo editor application. All code that could possibly throw an exception should be put in a `try-catch-finally` block:

```
try
{
  // Some code here
    *
    *
    *
  //in case a problem is found, an exception can be thrown
  throw(new Exception("Some information here"));
}
catch(Exception exception)
{
    ExceptionManager.Publish(exception);
}
finally
{
  // Code here will always be executed, whether
  //there was an exception thrown or not.
}
```

The example also shows how an exception is thrown within a method and how it provides additional information as a string. C# defines a `finally` block in addition to the `catch` block. The `finally` statement can be put after the `catch` block. The code in the block is always executed, either after the `catch` is finished or if the `try` block is finished.

An Important Note on Code Conventions From this point on, it is your task to provide the `try-catch` statements during the implementation of the code, even if it is not explicitly mentioned in the descriptions of the implementation. In addition, for the remainder of this book it is assumed that you will add the XML comments according to the coding guidelines while implementing the functionalities.

It's also up to you to save and check in files whenever you achieve an intermediate result. It is good practice to check in only code that is compiling and that has the added XML documentation.

The sample solutions that are provided on the accompanying CD contain comments and exception code.

5.6.5 Loading an Image

After the implementation of the custom exception publisher is completed, we start to implement the basic functionality of the photo editor application: loading an image. The .NET Framework provides standard file dialogs for loading and for saving files. The standard implementation of the Windows File dialog provides a window that enables the user to browse, select, name, and filter files. Because this functionality corresponds to what was specified in the requirements, the photo editor application uses the .NET-provided standard file dialogs. To start the implementation, we click on the `PhotoEditor.cs[Design]` tab and choose the Photo Editor Application form.

Next, we set the WindowState in the Layout section of the properties to Maximized. This sets the main window to maximized when the application is started. Compile and run the solution to see the described effect. After that, we add the File menu to the form. Go to the Toolbox, choose MainMenu, and drop the main menu on the form. (The Toolbox is usually shown on the left border of the screen. If the Toolbox is not visible, go to the View menu and select Toolbox, or simply press Ctrl+Alt+X.)

This action adds an empty menu bar to the form. Rename the menu to `MainMenu` in the properties window. To do that, first make sure that `main Menu1` is selected. Then go to the menu bar and type `&File` where the text Type Here is shown, and press Return. This will add a top-level menu item called File. The prefix `&` indicates that the shortcut to access the menu is the Alt+F key combination. After that, change the name of the menu in the properties section to `FileMenu`. Add a submenu item for the Open File menu item, as shown in Figure 5.13.

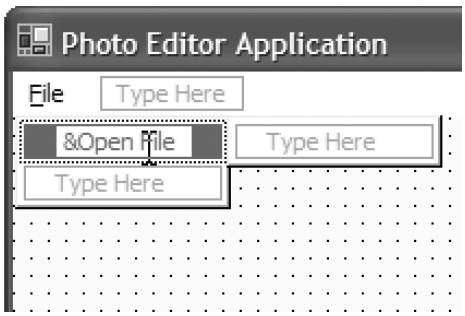

Figure 5.13 Creating the File Menu

Change the name in the property section of the submenu item to `Open File`. To add functionality to the menu, double-click on the OpenFile menu item. The `PhotoEditor.cs` source file will be opened, and the Designer adds a stub implementation of the `OpenFile_Click` event handler. The Designer also adds code to make the system aware of the function that handles the event of the specified type. Listing 5.7 shows the generated code that initializes some properties of the OpenFile menu item and registers the event handler method.

Listing 5.7 Adding an Event Handler Method

```
//
// OpenFile
//
this.OpenFile.Index = 0;
this.OpenFile.Text = "&Open File";
this.OpenFile.Click +=
    new System.EventHandler(this.OpenFile_Click);
```

To load an image, we must add a new field to the `PhotoEditorForm` class. To add new fields to a class using the Visual Studio class wizard, click on the Class View tab, right-click on the class, and select Add | Add Field. In the dialog window that opens, you specify the properties of the field. To create the field, specify `private` as the field access, `OpenFileDialog` as the field type, and `loadFileDialog` as the field name. Alternatively, simply add the field to the class manually by adding the following line to the class:

```
private OpenFileDialog loadFileDialog;
```

At initialization time the `loadFileDialog` object is created and memory is allocated for it. To do that, add the following line to the constructor `PhotoEditorForm()`:

```
//
// TODO: Add any constructor code after InitializeComponent call
//
loadFileDialog = new OpenFileDialog();
```

Finally, you implement the `loadFileDialog` in the `OpenFile_Click()` method, as shown in Listing 5.8.

Listing 5.8 The File Open Dialog

```
loadFileDialog.Filter = " jpg files (*.jpg)|*.jpg| gif files
(*.gif)|*.gif| bmp files (*.bmp)|*.bmp| All files (*.*)|*.*";
      loadFileDialog.ShowDialog();
      loadedImage = new Bitmap(loadFileDialog.FileName);
      this.Invalidate();
```

First, you define a file filter whose task it is to show only files of certain file types within the chosen directory. Files of other types (or file extensions) than the ones specified are not shown in the dialog window. This filter operation is provided by the `OpenFileDialog` class, which defines a `Filter` property that can be used to filter files by their types. In our example, the files that are of interest are image files of various types. The supported types are images with the extensions `.jpg`, `.gif`, and `.bmp`. In addition, we want to show all files in a directory when the file name is specified as "*.*".

After the file filter is defined, the standard Windows File Open dialog box is shown by calling its method `ShowDialog()`. A file dialog window appears that enables the user to browse directories and select a file. After the user has selected a file and clicked the Open button, the selected file name can be extracted from the file dialog object using the `FileName()` method. The .NET Framework provides converters to load and save the most commonly used image types. For all image types supported by the photo editor application, converters are provided by the .NET Framework. Thus, we need no customized functionality to work with the various image types. For supported formats and available conversion types, refer to the MSDN help pages.

To work with the loaded image, we must create a `Bitmap` object and allocate memory for it. We do this by calling the `Bitmap` constructor with the file name as a parameter and assigning the image to `loadedImage`. The loaded image field is not yet defined. Therefore, we add the following line to the `PhotoEditorForm` class:

```
private Bitmap loadedImage;
```

As you can see, the application uses a bitmap type image to work with rather than using the type under which the image was actually stored. It is at this point that Windows' automatic image type conversion saves a lot of work. To show the loaded image on the screen, we next force a refresh,

invalidating the current Windows form. Invalidating the window (or parts of it) sends a paint message to the appropriate window: either the control or the child window. As a result, the part of the screen that has been invalidated is redrawn using the `Paint` event handler method.

Before an image is loaded by the user, a default image will be shown. The image provided is `Hawaii.jpg` and should be located in the `bin` and `bind` directories if the debug version is run. To initialize the bitmap, simply add a line to the `PhotoEditorForm` constructor:

```
loadedImage = new Bitmap(@"Hawaii.JPG");
```

The last step tells Windows that custom drawing is needed for this form. To do that, you overridee the `OnPaint` method. To implement that method, you can either add the event handler manually by typing or create a stub implementation using Visual Studio.NET. To automatically create a stub, click on PhotoEditorForm in the `PhotoEditor.cs[Design]` view, and go to the properties section. If you press the yellow lightning symbol underneath the Solution Explorer window, you will see a tab with all the events listed. Double-click on the `Paint` event. This will create a stub implementation. Then implement the code as shown in Listing 5.9. The implementation reveals that the `Graphics` class is used to save the image in memory, which is then displayed on the screen.

Listing 5.9 Overriding the `OnPaint` Method

```
/// <summary>
/// Custom paint method.
/// </summary>
protected override void OnPaint(PaintEventArgs e)
{
  try
  {
    Graphics deviceContext = e.Graphics;
    deviceContext.DrawImage(loadedImage, 0,0);
  }
  catch(Exception exception)
  {
   ExceptionManager.Publish(exception);
  }
}
```

Compile and run the project. The default image will be displayed as the background image when the application is first started. To display another image, go to File | Open File. The Open File dialog will be shown, and you can browse the file system. Select an image, and it will be displayed in the application window.

5.6.6 Scrolling and Basic Image Operations

Loading a large image or resizing the application window shows a drawback of the current implementation: If the image is larger than the application window, only part of the image is shown. A better behavior for a Windows application is to show scrollbars if the image is larger than the window. Because the photo editor should be a well-behaved application, we need to add scrolling.

Depending on the needs of an application, there are several ways to add scrolling capabilities. Table 5.4 gives a short overview of the various techniques, describing how they are implemented and when they should be used.

Table 5.4 Comparison of Scrolling Techniques

Scrolling Technique	Description	When to Use
Using a scrollable control	These are controls directly or indirectly derived from `System.Windows.Forms.ScrollableControl`. They support scrolling, provided by the .NET Framework. For example, `TextBox`, `ListBox`, and the `Form` class itself support scrolling.	Use this scrolling technique when there is no need to draw in the control with GDI+, the control is composed of this custom control and other controls, and the virtual space is limited.
Placing a non-scrollable control in the `Panel` control	An instance of the `Picture` control is created, and the `Picture` control is placed in a `Panel` control. You then create a new image in which you can draw (possibly using GDI+). Then the background image of the custom control is set to the new image (including the graphics you were drawing).	Use this technique if you want to be able to draw into the image with GDI+, the custom control is not composed of other custom controls, and the virtual space is limited.

(*continued*)

Table 5.4 (*Cont.*)

Scrolling Technique	Description	When to Use
Using the `UserControl` class with child controls	Derive the control from the `UserControl` class, build an image, draw into the image, and set the `BackgroundImage` property of this control to the build image.	Use this technique if you need to draw into the limited virtual space (possibly using GDI+) and you're using child controls (constituent controls) of the custom controls.
Smooth scrolling	Create a custom control that derives from the `User` control, add vertical and horizontal scrollbars as child (constituent) controls, and write a `Paint` event handler that draws the image incrementally according to the scrollbar's position. In addition, the background color of the part of the control that is not covered by the image can be drawn in a defined background color.	This technique creates a polished and professional appearance. The image scrolls smoothly to the desired position rather than jumping to the new position, as it does when you use the build in scrolling.

For the photo editor application, we choose smooth scrolling to give the application a professional and polished appearance. This also lets us support graphics drawn with GDI+, something that is necessary for the supported graphics and text overlays described in the later chapters.

5.6.7 Refactoring of the Current Code

Before we begin implementing scrolling, we must do some refactoring of the existing project code. Even though the implemented code works and shows a quick result, we need to adapt it to meet the approved design and to give us the flexibility to accommodate the implementation of functional requirements to be added in later iterations.

According to the design specification, the `Picture` class should be split out as an independent class. This makes perfect sense, because the image and all its properties should be self-contained and separate from the GUI code. To add a new class, go to Solution Explorer, right-click on Photo Edi-

tor Application, and select Add | Add Class. Type `Picture` for the name of the class to be created, and press Open. After that, add a reference to the `System.Drawing` namespace to the new file by adding the following statement:

```
using System.Drawing;
```

The reference is needed to access the type `Bitmap`, which is used to store the image.

To implement the functionality to load an image, to the `Picture` class we add a `public` method named `LoadImage` with the return type `void` and no parameters. Switch to the class view and right-click on the `Picture` class and Add | Add Method. Or simply add the following line:

```
public void LoadImage() {...}
```

We also add a field for storing the image data. Add the following line to the `Picture` class:

```
private Bitmap loadedImage;
```

In C#, properties (also called *accessors*) are used to access `private` field data. To add the property, right-click on the `Picture` class (in the Class View tab of Solution Explorer) and select Add | Property. This will display a dialog box as shown in Figure 5.14.

Enter the data as it is shown in Figure 5.14, and press Finish. The property functionality with public accessor methods is generated automatically and is added to the class. To actually return the loaded image when the `get` method is called, change the return value from `null` to `loadedImage`. For now, the `set` part of the property is not used. In theory you could set the image by assigning the loaded image to the provided value:

```
loadedImage = value;
```

The next step is to add to the `PhotoEditorForm` class a `public` field that is of type `Picture` and is named `PictureObject`. This field will hold a reference to an instance of a `Picture` class object. The `PictureObject` then needs to be initialized with an allocated `Picture` class instance. This is done in the constructor of the `PhotoEditorForm`. Add the following line to the `PhotoEditorForm()` constructor:

```
PictureObject = new Picture();
```

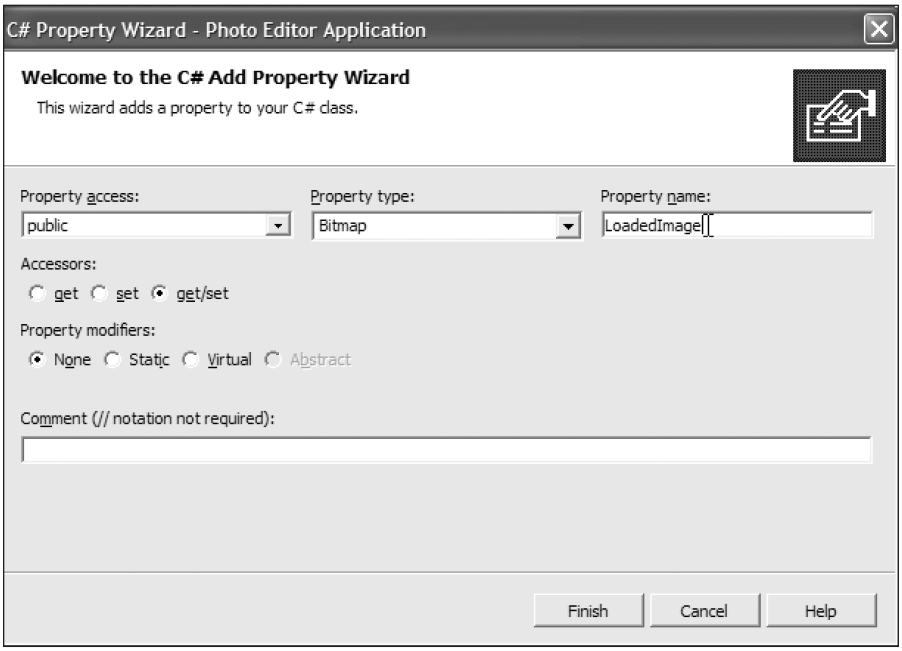

Figure 5.14 Adding the `loadedImage` Property

The previously added lines in the `PhototEditorForm` constructor need to be deleted because the `OpenFileDialog` and the `loadedImage` functionalities are now responsibilities of the `Picture` class. Therefore, we delete the following lines in the `PhotoEditorForm` constructor:

```
loadFileDialog = new OpenFileDialog();
loadedImage = new Bitmap(@"Hawaii.JPG");
```

Next, in the `OnPaint` event handler method, the `PictureObject.LoadedImage` should be painted instead of the default image. We implement this by changing the `DrawImage` call in the `OnPaint` event handler to look like this:

```
deviceContext.DrawImage(PictureObject.LoadedImage, 0,0);
```

To load an image via the `Picture` class, we change the `OpenFile_Click` event handler method to call the `LoadImage` method of the `Picture` class by adding this line:

```
PictureObject.LoadImage();
```

From the same method, we move the code for opening the file dialog window to the `LoadImage` method of the `Picture` class. Basically this is all the code except the following line:

```
this.Invalidate();
```

Now add the following statement to the `Picture.cs` file:

```
using System.Windows.Forms;
```

Then add a field for `OpenFileDialog` to the `Picture` class by adding this line:

```
private OpenFileDialog loadFileDialog;
```

Now it is time to load the default image at startup and to create an instance of `OpenFileDialog` in the `Picture` class constructor. Therefore, we add the following two lines to the `Picture` class constructor:

```
string defaultImage =
    PhotoEditorForm.GetApplicationDirectory + @"\Hawaii.jpg";
loadFileDialog = new OpenFileDialog();
```

You can see in this code that we have introduced an additional method to the `PhotoEditorForm` with the name `GetApplicationDirectory`. When you call this method, the path to the directory in which the photo editor application was started is returned. This is necessary in case a user starts the application from a different directory via the command line (in that case, if we would search the current directory for the default image we would search in the directory the command line shows and not the directory where the application was started). To make this work, add the following lines to the `PhotoEditorForm` class:

```
// Get the directory in which the application was started.
// Note: In C# you can initialize a member at definition time.
// The compiler will take care of initializing the member in the
// constructor of the corresponding class.
private static string applicationDirectory = Application.StartupPath;
```

```
/// <summary>
/// Accessor to the Application directory.
/// </summary>
public static string GetApplicationDirectory
{
  get
  {
     return applicationDirectory;
  }
}
```

To complete the refactoring, we delete the fields for `loadFileDialog` and `loadedImage` from the `PhotoEditorForm` class. Check the implementation by compiling and running the application.

Even though the changes made in this section are not very large, they show that refactoring can take a substantial amount of time. We think that this time is well invested if developers consistently try to improve the existing code. This does not mean that the implementation should necessarily become more complex by refactoring, but if developers identify possibilities to improve existing code with regard to maintainability and extensibility (if needed), the refactoring should be implemented (in fact, refactoring should simplify the code by making it easier for other developers to read and understand). We also strongly recommend that you do refactoring in very small steps. Refactoring can be a powerful tool if done consistently over the life cycle of a project. On the other hand, refactoring can become a nightmare if it is not done consistently throughout the whole life cycle of the project and if it is not done in small chunks with thorough testing in place.

5.6.8 Creating a Custom Control for Smooth Scrolling

Even though Visual Studio provides a wide variety of controls and wizards, sometimes it is necessary to develop controls with different, customized behavior. For the scrolling in this project, we want to provide smooth scrolling, something that the Visual Studio controls do not provide. Therefore, we'll develop a custom control. Before we start, it's a good idea to check whether a control that satisfies the needs of the project has already been developed and is available on the Internet.

As mentioned earlier, we want to implement the smooth scrolling control to give our application a professional feel when users scroll an image. The advantage of developing a custom control for this is that we can use the

control in other applications if needed. Another advantage is certainly that implementation of a custom control is a common task in application development, and we want to show how to develop and use a custom control to extend the features provided by Visual Studio.

For the implementation details of `CustomScrollableControl`, please refer to the sample solution on the CD. The project can be found in the `Chapter5\src\Photo Editor` directory. Instead of showing all the implementation details here, we explain the functionalities implemented and the necessary properties without the implementation details. You can implement the custom control based on the description of the functionality, or simply read through the text while checking the source code for the implementation details. If you try to implement the functionality, the Web comment report, which is available in the `doc` directory, might be helpful. It shows all members of the `CustomScrollableControl` class along with comments explaining the functionality.

To create custom controls, add a new C# project to the photo editor solution. The type of the project is a Windows control library, and the name of the control is `CustomScrollableControl`. After the project and its files are generated, change the name of the generated file and the class name from `UserControl1` to `CustomScrollableControl` and change the output directories to `bin` and `bind`.

Do It Yourself Try to implement the `CustomScrollableControl` feature. Use the description here, the comment Web pages in the `doc` directory, and the sample solution to guide you through the implementation.

5.6.9 Implementation of `CustomScrollableControl`

All fields are defined as `private`, and accessor methods are provided for fields that need to be accessed by other classes.

The `Image scrollingImage` field is used to store the image that is displayed in the control. The `get` property for this field returns the image; the `set` method sets the scrollable image and calls a method to adjust the scrollbars.

Another field needed is a point that specifies the viewport coordinates. `Point viewportCoords` represents the coordinates of the image relative to the control. The *viewport* defines the coordinates of the image relative

to the window. If the image is not scrolled, the pixel at position (0/0) of the picture is shown at position (0/0) of the control. If the image is scrolled by 100 pixels in y direction, the picture position (0/100) is shown at the (0/0) position of the custom control.

`Rectangle ScrollingImageArea` is a convenience accessor that returns a rectangle whose size is measured from the origin of the control to the x-coordinate of the vertical scrollbar and the y-coordinate of the horizontal scrollbar. This is equivalent to the area of the control that is available for drawing the image and is defined as the client area minus the area that is taken by the scrollbars.

The base functionality of this control is to scroll through an image smoothly. Therefore, we add vertical and horizontal scrollbars to the `Custom ScrollableControl` form. The scrollbars can be dragged from the toolbox onto the form. The scrollbars are positioned and docked in the form to the right and the bottom of the form, as in other Windows applications.

`private void drawImage` is a helper method that is used to calculate and draw the correct portion of the bitmap in the custom control. This method is called directly by the scrollbars whenever a change is detected. The method clips the region of the image to the area of the image that is visible in the control and draws the image. To clip the region, we use a `Rectangle` method that is defined in the `Windows.System.Drawing` namespace.

We customize the `Paint` event handler using `private void CustomScrollableControl_Paint` so that we can repaint parts or the entire image (in case, for example, the image is restored after the control was minimized). The GDI+ drawing surface, provided as a parameter to `PaintEventArgs`, is stored in a local variable called `graphics`. Then a `solidBrush` is created to fill the client area with a solid color. Next, we check whether the `scrollingImage` exists. If it does not, then there is no image and the complete client area is filled by the solid brush.

After that, a local variable of type `Rectangle` is created. The rectangle to the right of the image and left of the scrollbar is calculated and stored in the local variable `rect`. If the calculated rectangle is not empty, this area will be filled with the solid brush. After that, we do the same thing for the area below the image and above the horizontal scrollbar. Then the small rectangle in the lower-right corner is calculated and filled with the solid brush.

The `private void adjustScrollBars` method dimensions and sets certain properties for the scrollbars. This method does not take any parameters. A constant field is defined that is used to calculate the number of incremental steps for each scroll request. Then we check whether an image

exists. If it does, the minimum and maximum values of the scrollbars are set to 0 and the width or height of the image. In addition, we define the behavior of small and large changes to the scrollbars. The actual values of the scrollbars are set to the corresponding value of the viewport (meaning the coordinates of the upper-left corner in the actual image).

The `private void scroll` method is the heart of the custom scrolling functionality. This is the code that actually does the smooth scrolling. This method handles the scrolling from the given previous position to the current position of the scrollbars. To achieve a smooth scrolling effect, the viewport is incrementally changed until it is in the new end position. In between the incremental steps, the method is sleeping for a short time to simulate the effect of a sliding image. Therefore, two constants are defined. The first constant is used for the time period during which the control sleeps before displaying the next image position relative to the viewport, and the second is a divisor for the difference calculation of the previous and the current position of the scrollbar.

We also create a local variable that holds the drawing context; this variable is checked to see whether the previous value of the scrollbar is the same as the current value. If it is not, we must apply horizontal scrolling. A Boolean local variable indicates that the direction the scrollbar was moved, and the defined integer divides the absolute change that was made into the smaller, incremental steps. The incremental steps are then checked to see whether they are smaller than 1. If they are, then the value is set to 1 for scrolling up, or to –1 for scrolling down. Following that, the loop in which the incremental scrolling over the image is executed.

Then some checks are added to make sure that scrolling is stopped if the image is shown according to the scrollbar position and that the stepping did not go too far (if it did, the values are set to the final position). Before the image is drawn at its new position (with respect to the control), the control sleeps for a specified amount of time. Then the image is drawn, and the next image position is calculated and displayed. This continues until the image is shown in its final position, in which case a `break` statement is executed what makes the program jump out of the `while` loop.

`private void hScrollBar_Scroll` and `private void vScrollBar_Scroll` are the event handlers for the scrollbars. The `Scroll` event is triggered whenever the user clicks on the scrollbar and changes its position. The parameters that are passed to the event handler methods are references to the sender's `object` and `ScrollEventArgs`. The `ScrollEventArgs` object provides information on the scroll type. If the user clicks on the small

arrows of the scrollbar, either a `ScrollEventType.SmallDecrement` or a `ScrollEventType.SmallIncrement` is provided. If the user clicks inside the scrollbar or drags the scrollbar, a `ScrollEventType.LargeDecrement` or `ScrollEventType.LargeIncrement` type is provided. The event handlers for the scrollbars usually contain a `switch` statement, depending on the scroll type. In the case of `customScrollableControl`, the event handler stores the previous position of the viewport in a local variable before it assigns the current position to the viewport. Then the scrolling method is called with the previous position and the new position.

`private void CustomScrollableControl_Resize` implements the event handler for the `resize` event. The `resize` event is triggered whenever the control window is resized. In that case, the scrollbar position must be recalculated and the viewport may have to be updated. The implementation checks to see whether there is an image, and, if there is, the new `viewportCoords` are calculated. The `Math.Min` method is used to return the value of the smaller number that was provided. The `Math.Max` method is used to return the maximum value of the provided parameters.

Build the custom control from the sample solution so that it can be used in Visual Studio.NET.

5.6.10 Configuring the Custom Control for Use in Visual Studio.NET

It's easy to configure Visual Studio.NET to use the custom control. Just go to the Tools menu. Choose Add/Remove Toolbox Items, and browse to the directory in which the control was built (navigate to the `bind` directory if the debug version is used; otherwise, go to the `bin` directory, assuming we set the output path correctly) and choose the control, as shown in Figure 5.15. The control is then shown in the Toolbox. Like any other control, it can be used by dragging it from the Toolbox onto the form.

To implement the custom scrolling in the photo editor application, add the control to the form `PhotoEditorForm[Design]`. Position it as shown in the requirements and dock it to the top, left, and right of the form (by using the `Dock` property) and rename it `customScrollableControl`. Then double-click on the new control to create the `Load` event handler.

The next step is to draw the image to the custom control instead of the Form. In order to do that, remove the following line from the `Paint` event handler:

```
deviceContext.DrawImage(PictureObject.LoadedImage, 0,0);
```

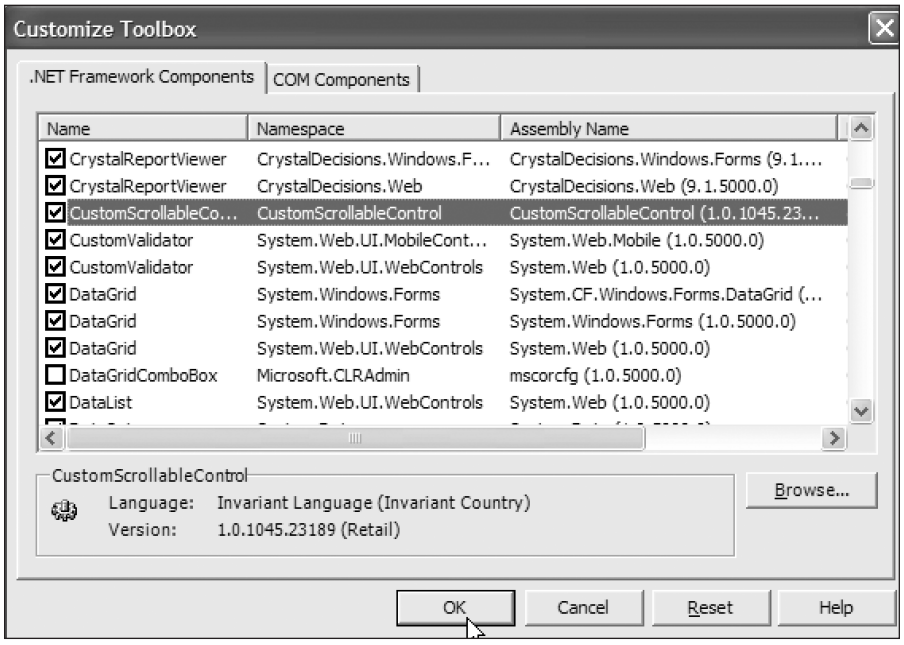

Figure 5.15 Adding a Custom Control to the Toolbox

Change the `OpenFile_Click` event handler method to display the image in the custom control, and invalidate the control to force a paint event. This will show the image within the new control. Listing 5.10 shows the new event handler.

Listing 5.10 The New `OpenFile_Click` Event Handler

```
/// <summary>
/// Opens a file dialog window so the user can select an
/// image to be loaded
/// </summary>
/// <param name="sender">A reference to the object calling
///   this method</param>
/// <param name="e">The event arguments provided by the
///   event handler</param>
/// <requirements>F:editor_load_and_save</requirements>
private void OpenFile_Click(object sender,
```

```
            System.EventArgs e)
    {
      try
      {
        PictureObject.LoadImage();
        if(PictureObject.LoadedImage == null)
          throw(new Exception("Error, image could not be
                loaded"));
        DisplayImage();
      }
      catch(Exception exception)
      {
        ExceptionManager.Publish(exception);
      }
    }
```

In addition, a new `DisplayImage` method is introduced in this example. This method is implemented in the `PhotoEditorForm` class.

Its implementation sets the scrolling image of the custom control to the currently loaded image of the `Picture` class instance, and it invalidates the custom control. The implementation of the `DisplayImage` method is as follows:

```
public void DisplayImage();
customScrollableControl.ScrollingImage = PictureObject.LoadedImage;
customScrollableControl.Invalidate();
```

The application, in its first cut, shows the default image when first loaded. To get the same result with the custom control, we call the newly implemented `DisplayImage()` method from within the `customScrollable Control_Load` event handler (which was generated by Visual Studio.NET automatically by the double-click on `customScrollableControl` in the `PhotoEditor.cs[Design]` tab).

This completes the implementation of smooth scrolling. It is now time to test the implemented functionality. Running the application and loading an image shows that the scrolling works smoothly, but the image flickers when the scrollbar is moved. We can prevent this by setting a style property in the `CustomScrollableControl` constructor. As a result of the change, the background is not drawn in the background color before the image is drawn. The following line is used in the constructor to accomplish this:

```
this.SetStyle (ControlStyles.Opaque, true);
```

After this change, the control works without any noticeable flicker. The next step is to provide the tab control containing the buttons for the basic image operations.

5.6.11 Cropping an Image

Cropping an image means to cut a particular region out of the image, keep the cropped portion, and delete the rest. For the photo editor, users will crop their images to get them to a standard size and upload them to the online shop. In a later version, we might provide interactive graphics for this task, but for the first version we present a dialog box where users specify the new size of the image.

To implement the crop dialog box, choose `PhotoEditorForm` and go to the Toolbox to drag a `TabControl` onto the form. In the properties of the tab control, select the following properties:

Properties of TabControl

(Name)	tabControl
Dock	Bottom

A tab control is used to group related image-processing functionality. Later in the book we will add other tab controls. Now rename the tab control to `tabControl`. Add a tab to the control by selecting and then right-clicking on the `tabControl`. Then select `Add Tab` to add the actual tab. Set its properties as follows:

Properties of Tab

(Name)	basicImageOperations
Text	Basic Image Operations

When the photo editor application is now run, the screen layout should correspond to the GUI shown in the `photo_editor` requirements, except that the buttons are still missing. To add a button for the cropping functionality, drag a button to the tab. Change the displayed text on the button to `Crop Image`, and change the name of it to `cropImageButton`. Double-click on the button to add the event handler for the click event. We use a dialog box to collect the size information for the crop rectangle. To let users open a dialog box when the crop button is pressed, you must add a new form to

the photo editor application. You do this by right-clicking the Photo Editor Application project in Solution Explorer and selecting Add | Add Windows Form. Name the new form `CropDialog`. Then drag two text boxes onto the form and change their properties as follows:

Properties of TextBox1 (Left) and TextBox2 (Right)

(Name)	imageWidth	imageHeight
AcceptsReturn	True	True
AcceptsTab	True	True
Text	Image Width	ImageHeight

In addition, add two buttons; change their names to `OKBtn` and `CancelBtn` (and change the text to be displayed on the button accordingly). Also add two labels next to the text boxes that explain what the text box input is—for example, "Enter new image height." The Form should now look approximately like Figure 5.16.

After adding all the design-related properties, we add the event handlers. A double-click on the image width text box will add an `imageWidth_TextChanged` event handler. Before implementing the event handler, add to the `CropDialog` class two `private integer` variables called `tempWidth` and `tempHeight` for the width and height. The event handlers are then implemented. To extract the new width from the text box entry, the `imageWidth` object provides a property, `Text`, that represents the text in the text box. This text is converted to an integer value and stored in the `tempWidth` variable. The functionality is shown in Listing 5.11.

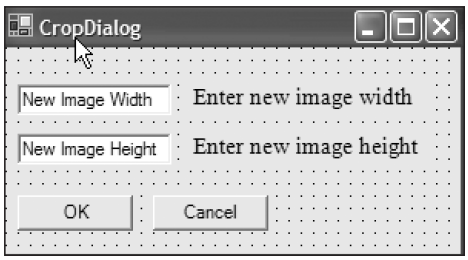

Figure 5.16 The Crop Dialog Box

Listing 5.11 The `TextChanged` Event Handler for the Width

```
private void imageWidth_TextChanged(object sender,
              System.EventArgs e)
    {
      tempWidth = Convert.ToInt32(imageWidth.Text);
    }
```

Do It Yourself Implement the height text box event handler method using Listing 5.11 as a template.

Next, we implement the OK and Cancel buttons. We start with the implementation of the Cancel button. To add the event handler for the button, double-click on the Cancel button in the `CropDialog.cs[Design]` view. If the Cancel button is selected, no cropping action is done and the `PhotoEditor` form should be invalidated (to force a repaint). In addition, the `CropDialog` box can be disposed of to indicate to the garbage collector that the memory is no longer being used. Therefore, we change the constructor of `CropDialog` to accept one argument of type `object` named `sender`. This object will be a reference to the calling `PhotoEditorForm` object. The `sender` object is then cast in the constructor to a `PhotoEditorForm` object and is stored in a local `private` variable of type `PhotoEditorForm` named `editorForm`. The code to be added to the constructor, after creating the `private` field, is as follows:

```
editorForm = (PhotoEditorForm)sender;
```

The `CancelBtn_Click` event handler is then implemented as shown in Listing 5.12.

Listing 5.12 The `CropDialog` Cancel Button

```
private void CancelBtn_Click(object sender, System.EventArgs e)
    {
      editorForm.Invalidate();
      this.Dispose();
    }
```

After that, we implement the OK button click event handler for the Crop dialog box. When the OK button is pressed, the stored values for width and height are sent to `PictureObject.CropImage`. The `Picture` object is then responsible for cropping the loaded image to the specified size. Therefore, we add the event handler by double-clicking on the OK button and adding the following lines to the event handler:

```
editorForm.PictureObject.CropImage(tempWidth, tempHeight);
editorForm.DisplayImage();
this.Dispose();
```

This will crop the image, assuming that a `CropImage` method is provided by the `Picture` class. Therefore, we must add the `CropImage` method to the `Picture` class in the next step.

Add a new `public void` method `CropImage` to the `Picture` class. This method takes two integer variables (the width and the height). Now that we have defined the signature, let's take care of the implementation. The `CropImage` method should check whether the provided parameters specify a region within the loaded image and whether the parameters are actually larger than zero. After that, the clip region needs to be calculated as a rectangle. We then clone the current image by applying the calculated cropping rectangle, and we store a copy of the cropped image in a temporary bitmap called `croppedImage`. The `loadedImage` is then set to the cropped image and is displayed. The complete implementation of the `CropImage` method is shown in Listing 5.13.

Listing 5.13 The `CropImage` Method

```
/// <summary>
/// Method called from CropDialog. The current
/// shown image is cropped to the size provided
/// in the parameters. The cropping is done
/// with a rectangle whose center is put on
/// the center of the image.
/// </summary>
/// <requirement>F:image_crop</requirement>
/// <param name="newHeight>height of the cropped
/// image</param>
/// <param name="newWidth">width of the cropped
/// image</param>
public void CropImage(int newWidth, int newHeight)
```

```
{
  // Check that cropping region is
  // actually within the image and that
  // the values provided are positive
  if((newWidth < loadedImage.Size.Width) &&
    (newHeight < loadedImage.Size.Height) &&
    (newHeight > 0) && (newWidth > 0))
  {
    int xmin = (loadedImage.Size.Width / 2) -
      (newWidth / 2);
    int xdim = newWidth;
    int ymin = (loadedImage.Size.Height / 2) -
      (newHeight / 2);
    int ydim =  newHeight;
    Rectangle rectangle = new Rectangle(xmin, ymin,
                  xdim, ydim);
    if(rectangle.IsEmpty)
    {
      throw(new Exception("Error, CropImage failed to
              allocate clipping rectangle"));
    }
    Bitmap croppedImage = loadedImage.Clone(rectangle,
           System.Drawing.Imaging.PixelFormat.DontCare);
    Bitmap oldImage = loadedImage;
    loadedImage = new Bitmap(croppedImage,
              rectangle.Size);
    if(loadedImage == null)
    {
      throw(new Exception("Error, Image memory allocation
                  failed"));
    }
  }
}
```

The last step is to implement the `cropImage` button click event handler. First, create the event handler. Then in the event handler create a new `CropDialog` object and show the dialog on the screen by adding the following lines:

```
CropDialog openDialog = new CropDialog(this);
openDialog.Show();
```

This completes the implementation of the cropping functionality. Compile and run the project to see whether it works. If you added the XML code documentation while implementing the functionality, then you can also generate the comment Web pages and you will have a nice description of the newly added functionality.

5.6.12 Rotate and Flip an Image

Now we show how to implement the rotate and flip functionality. To support this, we add new buttons to the tab card of `PhotoEditor.cs[Design]`. Add the buttons to match the screen layout shown in Figure 5.17. Then rename them and change the text of the buttons.

To add the event handler method for horizontal flipping, double-click on the corresponding button. Before adding the event handler, though, you should consider that the flipping and rotating of an image are basic image operations and therefore they should be handled by the `Picture` class. Therefore, add to the `Picture` class a `public void` method called `RotateFlipImage` that takes a parameter of type `RotateFlipType` that is named `rotateFlip`. (The `Bitmap` class actually supports rotating and flipping and provides an enumeration for various rotation and flip values.) Implement the event handlers for the button click events by passing the correct `RotateFlipType` to the `RotateFlipImage` method of the `Picture` class. (For information about `RotateFlipType`, see the MSDN documentation or the sample solution.) Then the `RotateFlipImage` method can be implemented as shown in Listing 5.14.

Listing 5.14 The `RotateFlipImage` Method

```
public void RotateFlipImage(RotateFlipType rotateFlip)
    {
      loadedImage.RotateFlip(rotateFlip);
    }
```

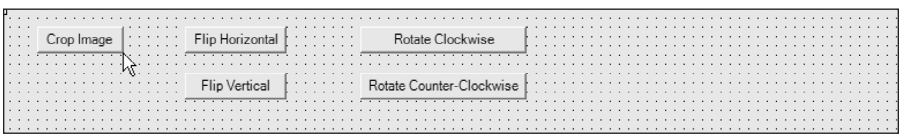

Figure 5.17 The Photo Editor Form Buttons

Do It Yourself Implement the functionality for vertical flipping, rotating clockwise, and rotating counterclockwise in the same way the horizontal flipping functionality was added. Use the enumeration members of `RotateFlipType`.

5.6.13 Save an Image

Before testing is started, we need to implement the last feature that is missing in the implementation. We have not implemented the functionality to save an image on disk. This task is very similar to the implementation of image loading. Instead of `OpenFileDialog`, we use `SaveFialeDialog`; and instead of loading the image, we save the image under the user-selected name.

Go to the `[Design]` view of `PhotoEditor.cs` and add a new menu item in the main menu, underneath the Open File entry. Name the menu item `saveMenu`, and change the text to `&Save as ...`. Add the event handler for the click event. The save file functionality is also implemented in the `Picture` class. Therefore, add a `public` method called `SaveFile()` to the `Picture` class, and call it from the event handler of the Save button. To make `SaveFileDialog` work, add a `private` member variable to the `Picture` class of type `SaveFileDialog`, and in the constructor create the object for it. The `SaveFile` method then can be implemented as shown in Listing 5.15.

Listing 5.15 The `SaveFile` Method

```
public void SaveImage()
    {
      saveFileDialog.Filter = " jpg files (*.jpg)|*.jpg|
        gif files (*.gif)|*.gif| bmp files (*.bmp)|*.bmp|
        All files (*.*)|*.*";
      saveFileDialog.ShowDialog();
      loadedImage.Save(saveFileDialog.FileName);
      saveFileDialog.Dispose();
    }
```

After this change has been made, compile and run the application to see the result. The images can now be loaded, rotated, flipped, cropped,

and saved. This is all the functionality that is needed for the project in the elaboration phase. The next task is to write unit tests to validate the implemented functionality.

Do It Yourself The buttons provided are not very nice-looking. You can make the program more appealing by adding bitmaps that show the user what the result of a requested image transformation will be. (Customize the rudimentary buttons provided, or develop custom bitmaps that can be loaded onto the buttons.) Also, we recommend that you change and play around with the implementation to see what improvements you can make.

Figure 5.18 shows the working application.

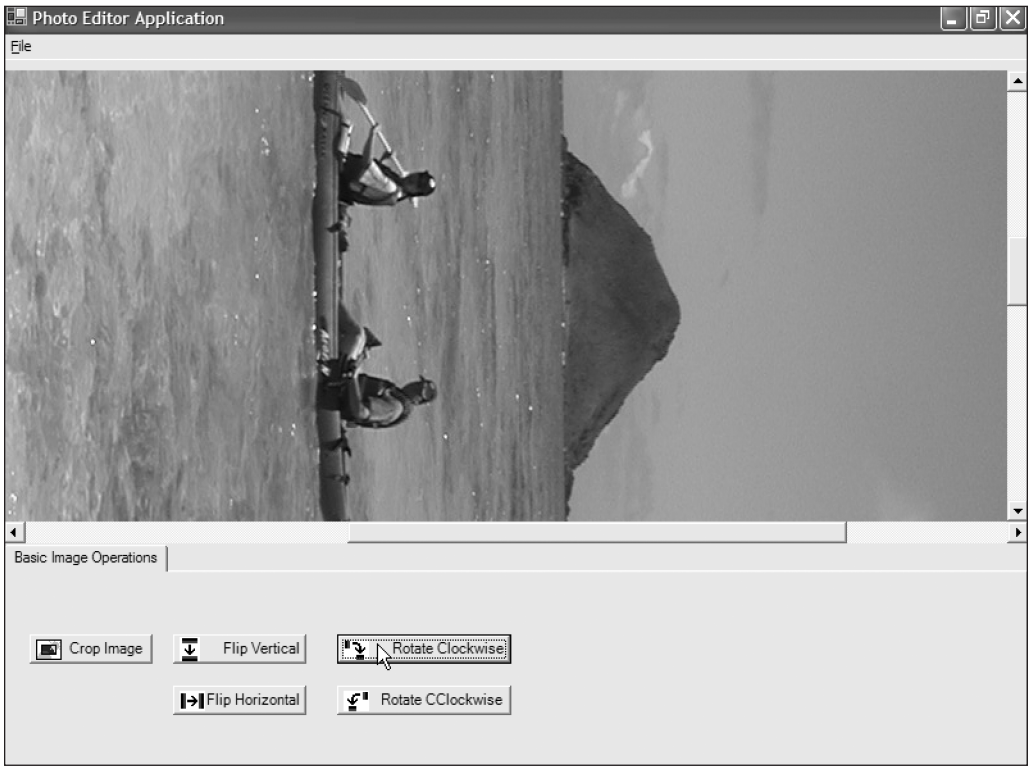

Figure 5.18 The Working Photo Editor Application

5.7 Unit Tests

The goal for the unit tests is to verify that all the required functionality is actually implemented and that the various modules are working in isolation. For the photo editor application, most of the required functionality is implemented in the `Picture` class. The only exception is the GUI. It is cumbersome to test the GUI automatically, and it is more an integration test rather then a unit test task. Therefore, the `F:photo_editor` key is not explicitly tested at the unit test level.

To test the load and save functionality, it would be an advantage to have overloaded methods where the test could specify the file name to be loaded or saved. With the current implementation of the `Picture` class, the file dialog would open every time the load functionality is called, making the automated tests very difficult. You can use a commercial capture replay tool to exercise these kinds of actions. Tools such as WinRunner do a good job on those things. But as mentioned, the easier solution (and probably the nicer solution as well) is to add two overloaded methods to the `Picture` class. The one overloaded method is the load method, which accepts a parameter of type `string`; the other is the save method, which also accepts a `string` parameter. The overloaded methods then load or save the picture under the provided file name. The implementation can be seen in Listing 5.16.

Listing 5.16 The Overloaded Save and Load Methods of the `Picture` class

```
/// <summary>
/// Opens the file with the provided name.
/// </summary>
/// <param name="fileName">name of the file to be opened</param>
/// <requirement>F:image_load_and_save</requirement>
public void LoadImage(string fileName)
{
  Bitmap oldImage;
  oldImage = loadedImage;
  loadedImage = new Bitmap(fileName);
  if(loadedImage == null)
  {
  throw(new Exception("Error, LoadImage with file name
    failed"));
```

```
  }
  oldImage.Dispose();

}
/// <summary>
/// Saves the current image
/// with the provided fileName.
/// </summary>
/// <param name="fileName">Name under which the image is
  saved</param>
public void SaveImage(string fileName)
{
  loadedImage.Save(fileName);
}
```

These two hooks can now be used by the test program to actually load and save an image without having to go through a file load and save dialog box. Very often, such hooks are implemented for testing. This is another reason to include the test team (if there is a separate test team available) in the project planning from the beginning. In this way, hooks for testing can be discussed in the planning phase of the project and implemented as features during development. In the photo editor project, this not as important because the development team also does the testing. Nevertheless, testing is incorporated in the project planning from the beginning.

5.7.1 The NUnit Test Framework

To test the `Picture` class functionality, an *automated test framework* would be very helpful. The framework we are looking for should be capable of running specified tests automatically, and it should show the result on the screen. Ideally it would have a GUI we could use to run selected tests, or it would run the tests from the command line (to let us run the tests every night using a script). It also should be easy to use so that we don't have to spend much time in training and setting up the test framework.

Luckily, such a test framework exists. Called NUnit, this framework is an automated test framework that is implemented for many programming languages in a similar form (for example, JUnit for Java, CppUnit for C++, and many more). The unit test framework was first developed for Smalltalk by the extreme programming group around Kent Beck. Later, the framework was ported to many languages and is now also available for Microsoft

.NET as NUnit. The framework can be downloaded from `http://www.nunit.org`. A quick-start guide and other documentation can be found at the site.

5.7.2 Unit Test Implementation

After downloading and installing the NUnit framework, we develop the tests (you will also find an installation of the NUnit test framework in the `src` directory). First, add a new class to the photo editor application project with the name `UnitTest`.

Add a reference to `NUnit.framework.dll` (which can be found in the `src\NUnit20\bin` directory), and add a `using` statement for `NUnit.Framework`. To use the classes provided by the `Picture` class, also add a `using` statement for the `System.Drawing` namespace that defines `RotateFlipType`. To tell the framework that the class `UnitTest` contains unit test methods that are executable by NUnit, we add an attribute called `[TestFixture]` before the class definition. Before implementing the actual test methods, we implement the setup and tear-down methods of the tests by inheriting two attributes from the unit test framework. The `SetUp` method is called before a test method is called, and the `TearDown` method is called after the test method is executed. The attributes used to indicate these methods are `[SetUp]` and `[TearDown]`.

For the unit tests of the `Picture` class, the `SetUp` method creates a new `Picture` object and writes to the standard output that a new test has started. In the `TearDown` method, the image is disposed of and an indication of the test result (passed or failed) is written to the standard output. Listing 5.17 shows the `SetUp` and `TearDown` implementation.

Listing 5.17 The Unit Test `SetUp` and `TearDown` Methods

```
namespace UnitTest
{
  [SetUp] public void Init()
    {
      Console.WriteLine("********* New Test-case: *******");
      Console.WriteLine("photo_editor");
      TestImage = new Photo_Editor_Application.Picture();
    }

    private Photo_Editor_Application.Picture TestImage;
    private bool amIPassed = false;
```

```
    [TearDown] public void Destroy()
    {
      TestImage.LoadedImage.Dispose();
      if(amIPassed)
      {
        Console.WriteLine("=> PASSED");
      }
      else
      {
        Console.WriteLine("%%%%% Failed");
      }
    }
}
```

For the file load test, we load the default image and check whether the loaded image has the correct dimensions. The save file test crops the image and saves it under a new file name. Then the regular default image is loaded again (to make sure the `Picture` object has been changed and has the original dimensions again) before the saved image is loaded. The saved image is checked to see whether its dimensions are the same as the cropping image information that was provided before the image was saved. The XML tag `<requirement> </requirement>` identifies which requirement key is tested in this test. To indicate that a method is a test method, the attribute `[Test]` is used before the method definition. The implementation can be seen in Listing 5.18.

Listing 5.18 Image Load and Save Test

```
/// <summary>
/// Test for F:image_load_and_save.
/// Windows standard dialogs are used, so really
/// this tests only whether the default image loaded has the
/// dimensions
/// that are expected!
/// </summary>
/// <requirement>F:image_load_and_save</requirement>
[Test] public void LoadandSaveImageTest()
{
  const string fileName = "Test.jpg";

  Console.WriteLine("image_load_and_save");
```

```
  Assertion.AssertEquals("Load Image, width",
    TestImage.LoadedImage.Width, 2048);
  Assertion.AssertEquals("Load Image, height",
    TestImage.LoadedImage.Height, 1536);

  TestImage.CropImage(200, 400);
  TestImage.SaveImage(fileName);

  TestImage.LoadImage("Hawaii.jpg");

  TestImage.LoadImage(fileName);

  Assertion.AssertEquals("Load Image, width",
    TestImage.LoadedImage.Width, 200);
  Assertion.AssertEquals("Load Image, height",
    TestImage.LoadedImage.Height, 400);
  amIPassed = true;
}
```

We perform the actual test case by calling an `Assertion` method with the pass/fail criteria. If the test case fails, then the test is failed and the rest of the tests within this method are not executed even if there are more test cases defined. That is why the flag `amIPassed` can be set to `true` in case the end of the method is actually reached.

Do It Yourself Implement test cases for image cropping, rotating, and flipping in the same way it was done for loading and saving. Check the unit test project into Source Safe. A sample solution is provided with the source code on the accompanying CD.

Now that the test cases have been implemented it is time to run them to see whether the developed program actually does what it is expected to do. We can run the test either from the command line or via the NUnit GUI. In this chapter only the GUI is shown.

To start the tests, go to the Start menu; then go to Programs | NUnitV2.0 | NUnit-GUI. The NUnit GUI will open. Choose File | Open and navigate to the directory where the unit tests were built. The test tree will be shown, and the tests can be run by pressing the Run button. The result will look like Figure 5.19.

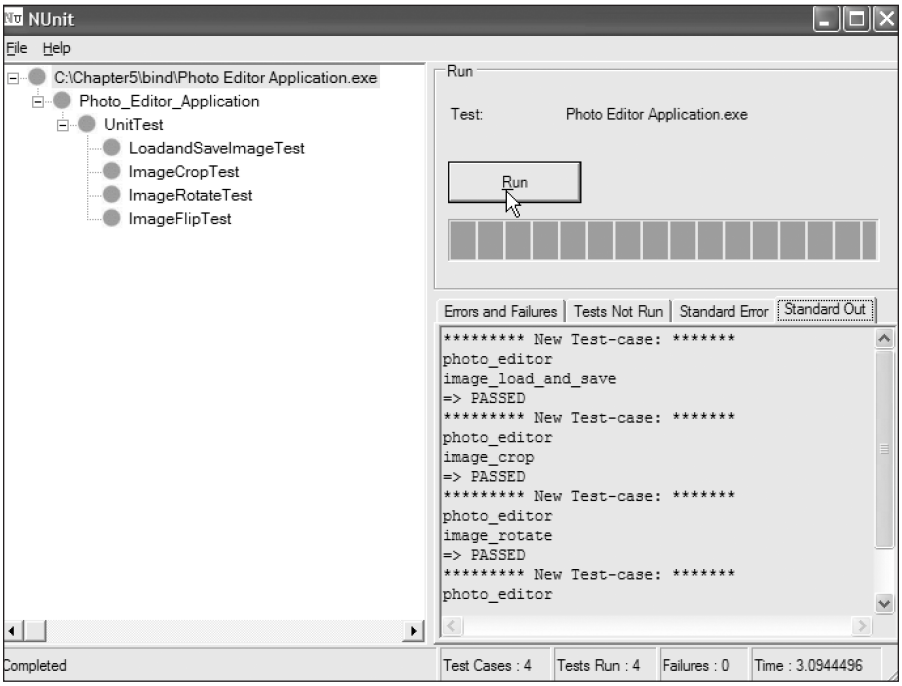

Figure 5.19 The NUnit GUI

The GUI gives a nice overview of the test result. If the progress bar is green, it means that all test cases passed. Otherwise, the bar will be red.

The Standard Out tab shows the results that were logged during the test run. If there are errors, information about the failed cases can be found in the Errors and Failures tab.

5.8 Conclusion

During the elaboration phase of our sample project, we design, develop, and implement a functional architectural baseline. We run the implementation and perform initial unit tests. We also update the project planning.

What remains is to produce the documentation from the source code. To provide a user's manual in the form of HTML pages, we produce the comment Web pages by going to the Tools menu and selecting the option

Build Comment Web Pages. In the displayed message box, we specify the destination where the files will be saved to (in our case, the documentation is stored in the `doc` folder of the project). Sample comment Web pages can be found in the `doc` folder in the sample solution. In addition, we produce the XML documentation, which is used for tracking (because it shows the requirement keys that have been implemented).

To produce the XML files, go to Solution Explorer, right-click on Photo Editor Application, and choose Properties. In the Configuration Properties dialog, specify the file name to be used as the XML documentation file. The XML file will be generated with every build, and the file will be saved in the build directory (`bin` or `bind` in the case of the photo editor project). The compiler will now create warnings if public members of a class do not specify an XML-style comment.

In addition to the documentation, we apply a label to the software and the documents (at least it should be done for the manually generated documents such as the project plan, the requirements document, and so on). To add a label in Visual Source Safe, open the Source Safe application and select the project. In the File menu choose Label. A window opens where you can specify the label. The label will be something like "Version 0.1.0." This complies with the Microsoft .NET standard, which uses three numbers to identify a specific version. Because the version produced is the first intermediate result, it is labeled with 0 (zero) for the main release, 1 for the intermediate release, and 0 for the minor releases. In addition, the `AssemblyInfo.cs` file of the Photo Editor Application should be adjusted to correspond to the version in the label before checkin.

5.8.1 Review

We must review the status of the project to decide whether the project will be continued. If the project is continued and an agreement with the customer is signed, then we check whether the project is ready to proceed to the construction phase. To decide whether the project is ready, we assess whether we have met the goals for the five core workflows:

- Requirements: Refine the requirements and system scope.
- Analysis: Analyze the requirements by describing what the system does.
- Design: Develop a stable architecture using UML.
- Implementation: Implement the architectural baseline.
- Test: Test the implemented architectural baseline.

It can be seen that the project meets all goals that were set for this phase and iteration. Therefore, the project is ready to move on to the next phase.

To get customer feedback early on, we deliver the project to the customer as intermediate V 0.1.0. It is crucial to deliver intermediate results that are functional but not yet the final product. In this way, the customer has a better understanding of the progress, and any changes that might be requested by the customer can be discussed and implemented as early as the intermediate project is delivered instead of at the end of the project. Especially with GUI development, this is very important. Another advantage of intermediate deliveries is that errors may be found by the customer and communicated to the development team while the product is still in development.

5.9 References for Further Reading

UML

Jim Arlow and Ila Neustadt, *UML and the Unified Process* (London, England: Addison-Wesley, 2002)

Martin Fowler, *UML Distilled* (Reading, MA: Addison-Wesley, 1999)

.NET Programming

`www.dotnetexperts.com/resources/`

`www.gotdotnet.com`

`www.msdn.microsoft.com`

The NUnit Test Framework

`http://sourceforge.net/projects/nunit/`

`http:www.nunit.org`

`http:/www.xprogramming.com/software.htm`