eBay Application Development

RAY RISCHPATER

Apress™

eBay Application Development

ISBN (pbk): 1-59059-301-4

Printed and bound in the United States of America 10987654321

Technical Reviewer: Lauren Darcey

Editorial Board: Steve Anglin, Dan Appleman, Gary Cornell, James Cox, Tony Davis, John Franklin, Chris Mills, Steve Rycroft, Dominic Shakeshaft, Julian Skinner, Jim Sumser, Karen Watterson, Gavin Wray, John Zukowski

Lead Editor: Jim Sumser

Assistant Publisher: Grace Wong

Project Manager: Kylie Johnston

Copy Editor: Ami Knox

Production Manager: Kari Brooks

Production Editor: Noemi Hollander

Proofreader: Thistle Hill Publishing Services, LLC

Compositor: Kinetic Publishing Services, LLC

Indexer: Kevin Broccoli

Artist: Kinetic Publishing Services, LLC

Cover Designer: Kurt Krames

Manufacturing Manager: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States: phone 1-800-SPRINGER, email orders@springer-ny.com, or visit http://www.springer-ny.com. Outside the United States: fax +49 6221 345229, email orders@springer.de, or visit http://www.springer.de.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, email info@apress.com, or visit http://www.apress.com.

The source code for this book is available to readers at http://www.apress.com in the Downloads section.

Contents at a Glance

Contents

Foreword

You hold in your hands one of the first books devoted solely to creating software applications that incorporate the eBay Marketplace. By applying the information in this book, you're joining a community of thousands of software developers who are taking the promise of distributed Web services and putting them into action to create and enhance real businesses today.

In this book, Ray Rischpater describes the methods that any software developer can use to automatically list items for sale on eBay, perform searches of eBay product listings, and more. The fact that Ray chose both Perl and C# for code examples illustrates how any language or platform can be used to create eBay applications. Whatever language or tools you choose, whichever platform you prefer, our XML-based API makes it easy to create customized eBay functionality today.

At the end of 2003, the eBay Marketplace comprised more than 94 million registered users around the world. During the last holiday shopping season, one out of every three people who were on the Internet came to eBay. Nearly 40 percent of items listed for sale on eBay.com are listed through the API. In 2004, the eBay API will service an estimated 10 billion calls from developers. Developers are using the eBay platform to build solutions that make trading easier for eBay buyers and sellers. The revolution of Web services is real, and eBay is proud to be part of that revolution.

The opportunities to build a business or accelerate an existing business with the eBay Marketplace have never been greater. We at eBay love to help developers who come up with innovative ideas to help users and improve the marketplace, and there's no better way to do that than creating a new application using the eBay API. We're looking forward to seeing what you come up with!

The eBay Developers Program Team
`http://developer.ebay.com`
February, 2004

About the Author

Ray Rischpater is a software engineer and writer who has focused on mobile computing since 1995. During that time, he has developed countless applications for Fortune 500 companies using handheld computers and wireless interfaces for enterprise and commercial deployment. He is the author of 8 books and 48 articles on mobile and wireless computing.

Ray Rischpater is a member of the American Radio Relay League as well as the Institute of Electrical and Electronics Engineers and is presently employed as a staff engineer at Rocket Mobile, Inc. When not writing books or software, Ray spends time with his family and provides emergency service using amateur radio in the San Lorenzo Valley in Northern California.

CHAPTER 3

Introducing the eBay SDK

In the last chapter, I took you on a whirlwind tour of the eBay SDK. You saw how the eBay SDK was divided into two sections: the data model representing abstractions of key data types used in eBay's business logic, and the API section encapsulating eBay API calls. You saw how your application accesses the Sandbox, and even explored your first eBay application using the Sandbox.

In this chapter, you build on that knowledge, getting the formal background on how the parts of the eBay SDK fit together, and which parts you need to use to craft specific bits of functionality for your application. I begin by giving you a detailed walkthrough of the eBay data model, building on the eBay-specific terminology you learned in the last chapter. You then see the API section of the eBay SDK in the same detail, and find out what happens under the hood when you use a member of the eBay SDK's API namespace. Finally, I conclude the chapter with another sample application that lets you explore eBay's tens of thousands of categories, a valuable tool you can use when crafting your own applications that lets users list items for sale on eBay.

Examining the eBay SDK Data Model

A key component of the eBay SDK is its *data model*—the interfaces and classes that encapsulate objects pertaining to everyday concepts in the world of eBay business. The data model includes fundamental objects representing accounts, feedback, items, sales, and users, along with a slew of other kinds of data your application can manipulate when working with the eBay service.

These objects are first-class citizens in the API, with their own methods and properties, just like the elements of the `eBay.SDK.API` namespace such as the `eBay.SDK.API.ValidateTestUserRegistrationCall` object you encountered in Chapter 2. By using these objects, you can avoid creating data structures or classes that encapsulate the same data as these do when using the eBay SDK. All of these objects include useful utility methods that stem from the `Object` class in the .NET Framework, including the following:

- The `Equals` method lets you test two objects for equivalency.
- The `GetHashCode` method lets you get a unique hash code for an object.
- The `GetType` method returns the item's type as a type object.
- The `Finalize` method performs necessary cleanup when the application is done using the object.
- The `ToString` method returns a `String` instance that describes the contents of the object.

These .NET-specific methods are not accessible from SDK COM components and Java packages.

Let's take a closer look at each of the namespaces that contain data model objects to see how they're organized.

Inside the eBay.SDK.Model Namespace

The `eBay.SDK.Model` namespace is a grab bag for data models that don't necessarily fit into one of the other namespaces. This namespace contains classes that represent eBay bids, postal addresses, and so on. Members of this namespace include the following:

- The `AccessRule` class, available via the `IAccessRule` interface, represents a single access rule to a specific API.
- The `Address` class, available via `IAddress`, encapsulates contact details such as one associated with a user.
- The `APIException` class, accessed via the `IAPIException` interface, which inherits from the .NET `Exception` class, represents a software exception. You will frequently use its `Message` property to get a text message describing the extension.
- The `Bid` class, accessed via the `IBid` interface, represents a buyer's bid to purchase an item.
- The `BidCollection` class, accessed via the `IBidCollection` interface, represents a collection of bids using `Bid` objects. To see how collections work, see the section "Managing Collections of Data Model Objects" later in this chapter.

- The `Category` class represents a category on the eBay service, accessed via the `ICategory` interface. You will encounter categories in detail in the section "Viewing eBay Categories: A Sample Application" at the end of this chapter.

- The `CategoryCollection` class, manipulated using the `ICategoryCollection` interface, represents a set of categories.

- The `CountryHelper` class provides a set of helper methods you use when converting between country names and two-character country codes such as *de* (Germany).

- The `CustomCategory` class, accessed via the `ICustomCategory` interface, represents a user-defined category.

- The `CustomCategoryCollection` class, accessed via the `ICustomCategoryCollection` interface, represents a collection of `CustomCategory` instances.

- The `ListingDesigner` class, accessed via the `IListingDesigner` interface, represents a custom listing format on the eBay service's Web site.

- The `LogoInfo` class, accessed via the `ILogoInfo` interface, encapsulates the return value from the `eBay.SDK.API.GetLogoURLCall.GetLogoURL` method, which returns a `String` containing the URL to the eBay logo that you can display in your application.

- The `SellerEvent` class, accessed via the `ISellerEvent` interface, encapsulates a record of the sales activities of an item up for auction.

- The `StringCollection` class, accessed via the `IStringCollection` interface, provides a collection of `String` instances.

- The `TimeLeft` class, accessed via the `ITimeLeft` interface, represents an arbitrary time interval in days, hours, minutes, and seconds, used for things like noting how long until an auction closes.

- The `UUid` class, accessed via the `IUuid` interface, represents a unique ID used by the eBay interfaces when performing item additions to avoid repeatedly adding the same item.

You will encounter examples with these objects throughout this and the next three chapters.

Inside the eBay.SDK.Model.Account Namespace

The `eBay.SDK.Model.Account` namespace contains classes that represent managing an eBay seller account. The core element of this namespace, the `eBay.SDK.Model.Account.Account` class, represents a seller account and provides a property, `Activities`, that contains a record of each account transaction that occurs for that account. The `eBay.SDK.Model.Account` namespace has the following additional classes:

- The `AccountActivity` class encapsulates a specific account transaction, representing it with properties including `Balance`, containing the balance as of a specific transaction; `Credit` and `Debit`, which indicate whether the transaction incurred a credit or a debit; and `Id`, which contains the unique ID of the specific `AccountActivity` within the eBay service. You use the `AccountActivity` class via the `IAccountActivity` interface.

- The `AccountActivityCollection` class lets you manage collections of `AccountActivity` objects. You use the `AccountActivityCollection` class via the `IAccountActivityCollection` class.

- The `AccountInvoiceView` class represents an account invoice with the properties `EmailAddress`, containing the e-mail to which invoices are sent; `InvoiceBalance`, the balance of the account when the invoice was generated; and `InvoiceDate`, the date at which the invoice was generated in Greenwich mean time (GMT). You use this class via the `IAccountInvoiceView` class.

- The `AccountPaymentMethodImpl` class represents how an account holder pays an account via its `Type` property; this property indicates the type of the payment using `AccountPaymentEnum`, which indicates how the account is paid. This class also includes properties such as `CreditCard`, which contain the actual account number of the payment method. You use this class via the `IAccountPaymentMethod` interface.

- The `AccountPeriodView` class represents a view into account activity over a specific billing period. Properties such as `State` (described by the `AccountStateEnum`), `CurrentBalance`, `LastInvoiceAmount`, `PaymentMethod`, and `LastAmountPaid` make this object resemble an account summary you'd receive from your favorite credit card company. You use this class via the `IAccountPeriodView` interface.

- The `AdditionalAccount` class contains the balance and ID of an account related to a principal `Account` instance for a user. You use this class via the `IAdditionalAccount` interface. The `AdditionalAccount` nodes represent historical data related to accounts that the user held with a country of residency other than the current one. eBay users can have one active account.

- The `AdditionalAccountCollection` represents a collection of `AdditionalAccount` objects. You use this class via the `IAdditionalAccountCollection` interface.

- The `CreditCardImpl` class represents the properties of a credit card used for payment, including its account number and expiration date. You use this class via the `ICreditCard` class.

- The `DateRangeImpl` class implants a range between two dates (date and time), denoted by its `BeginDate` and `EndDate` properties. You use this class via the `IDateRange` class.

- The `DirectDebitImpl` class represents a direct deposit payment method.

- The `EBayDirectPay` class represents a payment made to eBay via a direct payment such as a check or money order. You use this class via the `IEBayDirectPay` interface.

- The `InvoiceViewSettings` class represents the settings that you can apply to an `AccountInvoiceView` class via its `InvoiceMonth` and `InvoiceYear` properties. You use this class via the `IInvoiceViewSettings` interface.

- The `PeriodViewSettings` class represents the settings that you can apply to an `AccountPeriodView` class via its `DateRange` and `ViewRecord` properties. You use this class via the `IPeriodViewSettings` class and the helper enumeration `PeriodViewRangeEnum`, which lets you define the range for a period view's settings.

NOTE *When working with dates provided by eBay, bear in mind that all dates returned by eBay are in Greenwich mean time. You'll need to convert those to the time zone where your application is running!*

You will see more about using these in Chapter 4.

CAUTION *Don't confuse the eBay notion of an account with the notion of a user account. While eBay users have accounts in the Web sense, in the parlance of the eBay SDK, an* account *is the entity that tracks transactions, like your checking account, and you manipulate it using the classes in the* `eBay.SDK.Model.Account` *namespace. All of the stuff pertaining to a user is referred to by the name* user, *and you can find its functionality in the* `eBay.SDK.Model.User` *namespace.*

Inside the eBay.SDK.Model.Attributes.Motors and eBay.SDK.Model.Attributes Namespaces

A relatively new addition to the eBay service is the eBay Motors service, where you can list and sell your car, motorcycle, or boat to interested buyers, or find the dream vehicle you've been looking for. Its use and cost structure is similar to that of eBay, but given the very nature of the merchandise, it's a separate service, and the SDK has additional support for it.

The `eBay.SDK.Model.Attributes.Motors` namespace contains the bulk of this support, which has the following classes and interfaces:

- The `Car` class, accessed via the `ICar` interface, represents a car for sale, inheriting from `eBay.SDK.Model.Attributes.Motors.Vehicle`.

- The `CarOptions` class, accessed via the `ICarOptions` interface, represents the options a specific car may have, such as a cassette player, CD player, and whether or not the car is a convertible.

- The `CarPowerOptions` class, accessed via the `ICarPowerOptions` interface, describes the common powered control options a vehicle may have via its properties including `AirConditioning`, `CruiseControl`, and `PowerSeats`.

- The `CarSafetyFeatures` class, accessed via the `ICarSafetyFeatures` interface, describes safety features for a vehicle including whether or not it has antilock brakes, a driver-side airbag, or a passenger-side airbag, indicated by an enumeration.

- The `Motorcycle` class, accessed via the `IMotorcycle` interface, represents a motorcycle for sale. The `Motorcycle` class is a subclass of the `Vehicle` class.

- The `Vehicle` class, accessed via the `IVehicle` interface, represents features to all vehicles, including cars, motorcycles, and boats.

The eBay.SDK.Model.Attributes namespace contains the Attributes class, responsible for maintaining a motor item and the notion of whether it's a car or a motorcycle.

Inside the eBay.SDK.Model.Feedback Namespace

The eBay.SDK.Model.Feedback namespace contains the classes and interfaces that represent the feedback that buyers and sellers leave for each other. The notion of feedback is an important one in the eBay service, because it provides all users with peer-based accountability and a measurement of buyer and seller reliability. This namespace contains the following classes:

- The Feedback class and its IFeedback interface encapsulate a single feedback item for a user.
- The FeedbackCollection class and its IFeedbackCollection interface represent a collection of Feedback objects.

You will see more about using these in Chapter 4.

Inside the eBay.SDK.Model.Item Namespace

The eBay.SDK.Model.Item namespace, like its siblings the eBay.SDK.Model.Account and eBay.SDK.Model.User namespaces, contain fundamental elements you'll use when interacting with the eBay SDK for listing and searching for items. This namespace contains the following classes:

- The Fees class and its IFees interface represent the fees a user is charged when listing an item on eBay for sale, or the fees associated with a completed auction.
- The GiftSettings class and its IGiftSettings interface represent the buyer's choices for how a purchase is presented to the recipient when purchasing an item through an eBay store.
- The Item class and its IItem interface are at the heart of eBay's representation of items that are traded in eBay auctions. These classes encapsulate the description, bid status, and other information about an item for auction through its properties.
- The ItemCollection class and its IItemCollection represent a collection of Item instances.

- The `ItemFound` class and its `IItemFound` interface represent an item found by searching the eBay service for items.

- The `ItemFoundCollection` class and its `IItemFoundCollection` interface represent a collection of `ItemFound` objects.

- The `ItemStatus` class and its `IItemStatus` interface provide specific information about an item for auction, such as whether it's an adult-oriented item, whether the reserve is met, and so forth.

- The `ListingFeatures` class and its `IListingFeatures` interface provide details about the appearance of an item's listing, such as typesetting options on the item's Web page on the eBay service. You can serialize a `ListingFeatures` object using the `ListingFeaturesSerializer` class and its `IListingFeaturesSerializer` interface.

- The `PaymentTerms` class and its `IPaymentTerms` interface represent how a buyer may pay for an item listed for auction. You can serialize a `PaymentTerms` object using the `PaymentTermsSerializer` class and its `IPaymentTermsSerializer` interface.

- The `ShippingRegions` class and its `IShippingRegions` interface represent which regions (continents) a user will ship a specific item to. It's organized as a set of Boolean properties naming specific regions. You can serialize a `ShippingRegions` object using the `ShippingRegionsSerializer` and its `IShippingRegionsSerializer` interface.

- The `ShippingSettings` class and its `IShippingSettings` interface represent how the seller will ship an item and to where the item can be shipped (using a `ShippingRegions` object), along with whether the buyer or the seller will pay shipping charges.

You will see how many classes and interfaces in this namespace are used in Chapter 5.

Inside the eBay.SDK.Model.Sale Namespace

The `eBay.SDK.Model.Sale` namespace contains the classes and interfaces that represent an actual item sale (and not the listing, which is managed through the classes in the `eBay.SDK.Model.Item` namespace). This namespace contains the following:

- The `CheckoutData` class and its `ICheckoutData` interface provide a container for the details of the checkout session that occurs as buyer and seller settle an auction by items and sales.

- The `CheckoutDetailsData` class and its `ICheckoutDetailsData` interface provide a container for fine-grained details about a checkout transaction, such as additional shipping costs or the seller's return policy and special instructions.

- The `CheckoutStatusData` class and its `ICheckoutStatusData` interface provide a container for the current state of the checkout process.

- The `Sale` class and its `ISale` interface provide the representation of an actual item sale on the eBay service. These classes contain properties that let you determine the nature of a sale, including the amount paid, current price, and number of items purchased.

- The `SaleCollection` class and its `ISaleCollection` interface provide a container for collections of `Sale` objects.

You will see how to use the members of this namespace in Chapter 5.

Inside the eBay.SDK.Model.User Namespace

The `eBay.SDK.Model.User` namespace contains everything needed to represent a user of the eBay service. Both buyers and sellers are users, although only those users that sell items have accounts (which track the money they owe eBay for the use of its auction services). Of course, the `User` class and its `IUser` interface are at the heart of this namespace, but it also contains a few other classes:

- The `SchedulingLimit` class and its `ISchedulingLimit` interface represent the maximum number of `Item` listings a user may schedule, and how long in advance an item can be scheduled.

- The `User` class and its `IUser` interface represent everything that makes up an eBay user, including her e-mail address, cumulative feedback score, registration time and date, unique ID, and so forth. Many of its properties are instances of other classes.

- The `UserCollection` class and its `IUserCollection` interface, as you can guess by now, lets you manage a collection of `User` objects.

Managing Collections of Data Model Objects

The .NET Framework provides a type-safe mechanism by which to manage collections of like-typed objects via classes in the `System.Collections` namespace and the mother of all collections, the `ICollection` interface, which defines enumerators, size, and synchronization methods for all collections. The SDK architects at eBay wisely chose to use this model for managing collections of objects via classes such as `UserCollection`, `SaleCollection`, `CategoryCollection`, and so on.

The interface to a collection is quite simple: Each collection can provide you with a count of the number of items in the collection and an *enumerator,* an object that you can use to fetch sequentially each item in the collection. (In some component models, the enumerator is known by the term *cursor.*) The collections in the eBay SDK share the following methods in their interfaces:

- The `Add` method lets you add an object to the collection.
- The `AddRange` method lets you add (concatenate) one collection to the end of the other.
- The `Clear` method lets you remove all items from the collection.
- The `Contains` method lets you determine if a collection contains a specific element.
- The `CopyTo` method lets you copy the contents of the collection to an array.
- The `GetEnumerator` method lets you get an enumerator so that you can iterate over all objects in the collection.
- The `IndexOf` method lets you obtain the index of a specific entry in the collection.
- The `Insert` method lets you insert (add) an item between two items in the collection.
- The `ItemAt` method lets you obtain an item at a specific index of the collection.
- The `ItemCount` method lets you determine the number of items in the collection, just like the `Count` property.
- The `Remove` method lets you remove an item from the collection.
- The `RemoveAt` method lets you remove an item at a specified index position within the collection.

While you can certainly use the `ItemCount` method and a traditional `for` or `while` loop to iterate across all elements in a collection, it's often easier to do so using an *enumerator,* which lets you visit each item in the collection. In C#, you can access the items directly using a `foreach` loop through code such as that shown in Listing 3-1.

Listing 3-1. Using an Enumerator in C#

```
1: ICategoryCollection categories;
2: ...
3: foreach( eBay.SDK.Model.ICategory category in categories )
4: {
5:   Console.WriteLine( category.CategoryName );
6: }
```

Here, the C# compiler uses the base class interface for collections to fetch each `Category` object within the `categories` collection in line 3, and the loop prints the name of that category using `Console.WriteLine` on line 5. Listing 3-2 shows the corresponding Perl code.

Listing 3-2. Using an Enumerator in Perl

```
1: $categoryEnumerator = $categories->GetEnumerator();
2:
3: while( $categoryEnumerator->MoveNext () )
4: {
5:   $category = $categoryEnumerator->Current();
6:   print $category->{ 'CategoryName' }, "\n";
7: }
```

This code does exactly the same thing, but you need to use the `while` loop on line 3 and directly access the collection's enumerator via the method on line 1. On initialization, the enumerator's notion of which element to return when calling its `Current` method is an imaginary element *before* the first element, so the first time you call `MoveNext`, the `Current` method will return the first element, and so on. The loop terminates when `MoveNext` returns false, indicating that it has visited every item in the collection.

I show you how to use collections, specifically `CategoryCollection`, in the sample application in the section "Viewing eBay Categories: A Sample Application" at the end of this chapter.

Examining the eBay SDK API Interfaces

While you can create some pretty nifty applications using just the classes and interfaces from the data models you saw in the last section, the real value of the

eBay SDK is in the `eBay.SDK.API` namespace, where there's an interface defined for each eBay API call.

In turn, each interface has a corresponding API call class that implements the eBay API call. Typically, the implementation of this class contains code to create the appropriate XML class, and then other classes such as the `eBay.API.ApiSession` class and the `eBay.API.APICall` class contain the logic necessary to make the network transaction to issue the request, receive the response, and parse the resulting XML. For each eBay API call, the class is named *apiName*`Call`, such as `eBay.SDK.API.ValidateTestUserRegistrationCall`, with its interface `eBay.SDK.API.ValidateTestUserRegistration`.

The superclass of all `eBay.SDK.API` classes, `eBay.API.APICall`, contains several properties that all classes of the `eBay.API` namespace inherit. To use any eBay API call, you must set these properties as follows:

- The `ErrorLevel` property must contain an integer indicating the level of verbosity for resulting error messages. These integers are defined by the `ErrorLevelEnum` enumeration. If no value is set, the default value specified by the API session in the `IApiSession` object is used.

- The `DetailLevel` property specifies which fields are returned by the eBay API call. It's not used by all functions, but lets you pick which properties should be returned by an API call.

- The `SiteId` property specifies the eBay site to which the API call pertains, such as sites in the United States or Germany, or another site. Like `ErrorLevel`, the default value specified in the API session is used if you do not set this property.

- The `Timeout` property specifies how long the API request may go unanswered before it is cancelled and an exception generated. Like `ErrorLevel`, the default value specified in the API session is used if you do not set this property.

- The `ApiCallSession` property must contain a valid `APISession` instance that has been initialized with the URL of the eBay service and your developer, application, and certification keys.

- The `Verb` property specifies the verb (e.g., `AddItem`, `GetSellerList`) of the eBay API call that you are going to make.

- The `Response` property will be set after each successful API call for you to examine the original XML message returned from the eBay API server.

- `OnPostRequestXml` is the event that you can handle to get notified when the underlying constructed XML is posted to eBay. The `OnReceiveResponseXml` event is the one to notify you when an response XML is received from eBay.

Each of the API calls in the namespace has a method that's the same name as the API itself. This method—eBay calls it the *master method*—makes the actual API call to eBay and returns an appropriate object or object collection with the response, or raises an exception in the event of an error. For example, calling `ValidateTestUserRegistration.ValidateTestUserRegistration` issues the underlying eBay API call `ValidateTestUserRegistration` to the eBay service.

To supply arguments to an `eBay.SDK.API` call, you set the arguments via the API call's properties. For example, `AddItemCall`, which adds a new item to eBay to create a new auction for an item, has the property `ItemToAdd`, which you must set to the item you wish to add to the eBay service.

Unlike the various data models, there's no subdivision of the `eBay.SDK.API` namespace. There are 27 API calls in the SDK; Table 3-1 summarizes the ones you're likely to use most often.

Table 3-1. The eBay.API.SDK Calls

Call	Interface	Purpose
`AddItemCall`	`IAddItem`	Adds an `Item` for auction to the eBay service
`AddToItemDescriptionCall`	`IAddToItemDescription`	Adds information about a listed `Item`
`GetAccountCall`	`IGetAccount`	Returns the account of an eBay user
`GetAPIAccessRulesCall`	`IGetAPIAccessRules`	Returns the rules by which your application can use the eBay API, including details such as the number of API requests you may make over a unit of time
`GetBidderListCall`	`IGetBidderList`	Returns information about the bidders for an item in an auction
`GetCategoriesCall`	`IGetCategories`	Returns a collection of categories given a parent category
`GetEbayOfficialTimeCall`	`IGetEbayOfficialTime`	Returns the official time at eBay, which presides over all auctions
`GetFeedbackCall`	`IGetFeedback`	Returns feedback regarding a user

Table 3-1. The eBay.API.SDK Calls

Call	Interface	Purpose
GetHighBiddersCall	IGetHighBidders	Returns the list of all high bidders for a Dutch auction
GetItemCall	IGetItemCall	Returns the specified item
GetItemTransactionsCall	IGetItemTransaction	Returns the transactions for a specific item
GetLogoURLCall	IGetLogoURL	Returns the URL of the eBay application logo
GetSearchResultsCall	IGetSearchResults	Returns the results of searching for an item by various criteria
GetSellerEventsCall	IGetSellerEvents	Returns price updates and sale information from eBay
GetSellerListCall	IGetSellerList	Returns the list of items for sale on eBay, bounded by the API's search criteria
GetSellerTransactionsCall	IGetSellerTransactions	Returns the list of transactions for a seller
GetStoreDetailsCall	IGetStoreDetails	Returns the details pertaining to a specific eBay store
GetUserCall	IGetUser	Returns information about a specific eBay user
GetWatchListCall	IGetWatchList	Returns the items on an eBay user's watch list
LeaveFeedbackCall	ILeaveFeedback	Leaves feedback for an eBay user
RelistItemCall	IRelistItem	Relists an existing item on eBay
ReviseItemCall	IReviseItem	Revises information about an item on eBay
ValidateTestUserRegistrationCall	IValidateTestUserRegistration	Validates a test user, just as if the user were a real seller and had entered a credit card number
VerifyAddItemCall	IVerifyAddItem	Verifies that an item was successfully added to the eBay service

WARNING *To prevent your application from overloading eBay's application servers, there are a number of restrictions on how you can use these APIs, such as repeatedly using* `GetCategoriesCall` *to fetch individual categories, or using* `GetItemCall` *to get an item's highest bidder. To get a better understanding of these restrictions, consult Chapter 9 in this book, or see the documentation at the eBay Web site.*

Debugging eBay Applications

With the advent of today's modern interactive programming environments, debugging isn't nearly the unadulterated pit of despair and misery it used to be. Languages like C, C++, C#, and Perl all have interactive debuggers; depending on your preferences and budget, they can be simple command-line affairs like the famous GNU GDB application, or the amazingly powerful Microsoft Visual Studio debugger.

Debugging integration touch points—such as the ones between your application and the eBay SDK—can often be a challenge, because you don't have access to what goes on under the hood of something like the eBay SDK. As a result, if you make an incorrect assumption about how the environment operates, you can introduce a defect, and spend valuable time working out just what caused the defect in the first place. To make your life easier, here are a couple of things you can do to monitor the behavior of your eBay application when things go wrong.

Logging eBay SDK XML for the eBay API

The `eBay.SDK.API.ApiSession` class has the ability to log the XML requests and responses generated when executing eBay API calls. While these logs are of the most use when you're trying to debug eBay API applications (such as those I discuss in Chapters 8 and 9) and want to see how the eBay SDK uses the same eBay API calls, it can also shed light on other failures, such as when your application simply doesn't get a response from the eBay server, or when you're getting a drastically different response than you expect.

Listing 3-3 has one such eBay SDK XML log, here intentionally pruned to save space in the book.

Listing 3-3. An eBay SDK XML Log

```
1: [7/22/2003 00:37:38 AM]
2: [Request][https://api.sandbox.ebay.com/ws/api.dll]
3: <?xml version="1.0" encoding="utf-8"?>
4: <request>
```

```
 5:   <RequestUserId>******</RequestUserId>
 6:   <RequestPassword>******</RequestPassword>
 7:   <DetailLevel>1</DetailLevel>
 8:   <ErrorLevel>1</ErrorLevel>
 9:   <SiteId>0</SiteId>
10:   <Verb>GetCategories</Verb>
11:   <CategoryParent>0</CategoryParent>
12: </request>
13:
14: [7/22/2003 00:37:39 AM]
15: [Response]
16: <?xml version="1.0" encoding="utf-8"?>
17: <eBay>
18:   <EBayTime>2003-07-22 00:37:39</EBayTime>
19:   <Categories>
20:     <Category>
21:       <CategoryId>0</CategoryId>
22:       <CategoryLevel>1</CategoryLevel>
23:       <CategoryName><![CDATA[]]></CategoryName>
24:       <CategoryParentId>0</CategoryParentId>
25:       <IsExpired>0</IsExpired>
26:       <IsVirtual>0</IsVirtual>
27:       <LeafCategory>0</LeafCategory>
28:     </Category>
29:     <CategoryCount>1</CategoryCount>
30:     <UpdateGMTTime>2003-07-22 00:37:39</UpdateGMTTime>
31:     <Version>26</Version>
32:   </Categories>
33: </eBay>
```

As you can see, the log file simply keeps track of every request (marked by the keyword `[Request]` followed by the server URL) and response. For security, the log obscures the eBay user account name and password in the request (lines 5–6). You can see how each `eBay.SDK.API` call property maps to a specific XML entity, so checking the log is a good way to find out if you're forgetting an obvious property.

Getting the `ApiSession` object to generate a log file is easy:

1. First, set its `LogCallXml` property to `true`.

2. Next, set its `Log` property to a newly created `eBay.SDK.LogFile` instance.

3. Open the newly created log file using its `Open` method. If the specified file doesn't exist, it will be created.

Listing 3-4 recaps these steps in a snippet of C#, while Listing 3-5 does the same in Perl.

Listing 3-4. Enabling the eBay SDK Log in C#

```
1: IApiSession apiSession;
2:
3: apiSession = new ApiSession();
4: apiSession.LogCallXml = true;
5: apiSession.Log = new eBay.SDK.LogFile( );
6: apiSession.Log.Open( "c:\\tmp\\ebaylog.txt" );
7: apiSession.Url = "https://api.sandbox.ebay.com/ws/api.dll";
```

Listing 3-5. Enabling the eBay SDK Log in Perl

```
1: $apiSession = Win32::OLE->new("eBay.SDK.API.ApiSession");
2:
3: $apiSession->{'LogCallXml'} = True;
4: $apiSession->{'Log'} =
5:   Win32::OLE->new("eBay.SDK.LogFile");
6: $apiSession->{'Log'} ->Open("c:\\tmp\\ebaylog.txt");
7: $apiSession->{'Url'} =
8:     "https://api.sandbox.ebay.com/ws/api.dll";
```

Both of these snippets are pretty self-explanatory. After creating the `ApiSession` object (line 3 of Listing 3-4, line 1 of Listing 3-5), the code sets the `LogCallXml` property (line 4 of Listing 3-4, line 3 of Listing 3-5). Next, it sets the `Log` property to a newly created log file, and sets the name of the log file (lines 5–6 of Listing 3-4, lines 4–6 of Listing 3-5). Finally, the code sets the session's URL to use the eBay Sandbox.

Monitoring Network Activity

Another tool you can use (admittedly crude by today's standards of single-stepping through source code, breakpoints, watchpoints, and viewing or changing variables on the fly) is monitoring your application's use of the network when executing your application. This is akin to the activities of our forefathers (or, alas, some of *us!*) that used a computer's CPU load or the status of front-panel lights to get a feel for what our application was doing as it executed.

The easiest tool to use is the Microsoft Windows Task Manager; the Networking tab shows a graph of network activity. For it to be very meaningful, of course, you should shut down other applications that may use your computer's network adapters, including e-mail clients, Web browsers, and

even instant messaging utilities. By single-stepping through your code, you can see if long pauses are actually due to eBay SDK network requests, and correlate the behavior with the network activity you see in the network activity graph.

Using Network Use to Indicate Application Defects

An anecdote here may well prove the point. When debugging the sample application that appears in the next section, I mistakenly asked eBay to send me *all* categories, rather than just the categories that were subcategories of a specific parent. I was doing this on the weekend from home over a dial-up line. After watching it start and apparently never return a couple of times, I decided that our connection was just being flaky, and launched the application and went to town to buy groceries. To my chagrin, it was *still* downloading something (I couldn't tell what, this being the domain of the eBay SDK) when I returned. I wasn't even sure it was downloading, until I brought up the Microsoft Windows Task Manager and clicked the Networking tab, and was met with a graph frighteningly like the one you see in Figure 3-1. As soon as I saw the graph, I realized my mistake: I was downloading all categories! A minute's worth of code changes later, and my mistake was fixed, my application ran considerably more smoothly, and the Windows Task Manager was showing me the graph in Figure 3-2.

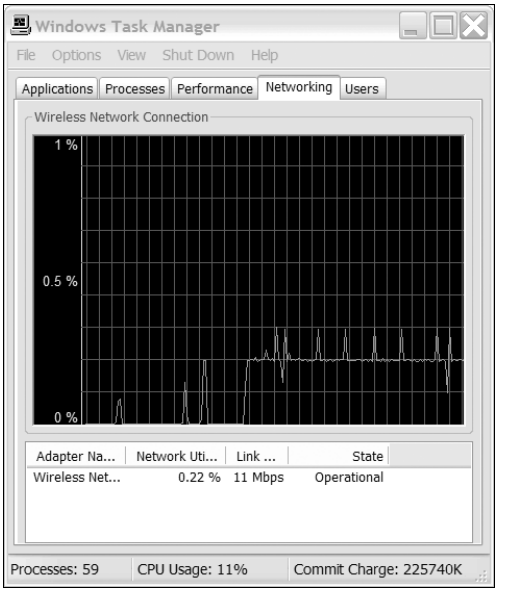

Figure 3-1. Network activity vs. time with defective application

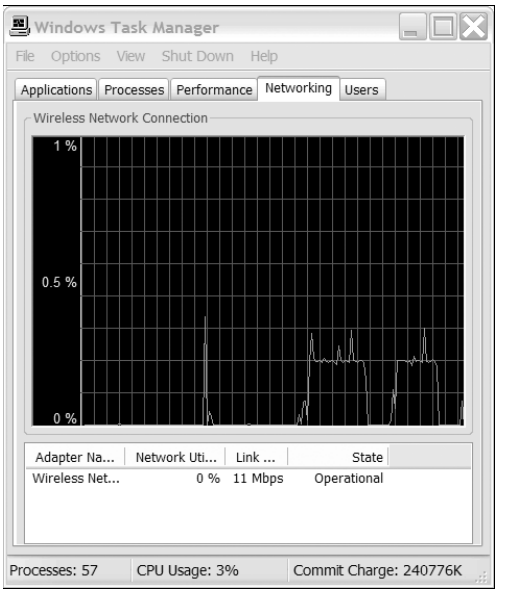

Figure 3-2. Network activity vs. time with corrected application

For more challenging bugs, or when you're using the eBay API directly, bypassing the eBay SDK, you may need to call out the heavy artillery: a network monitor. Tools like UNIX's tcpdump and the Microsoft Windows–friendly WinDump (available from `http://windump.polito.it/`) let you monitor the exact contents of a TCP/IP connection, and are invaluable tools when debugging *any* networking application.

Viewing eBay Categories: A Sample Application

This chapter's sample application is Categories, an application that lets you browse eBay's categories using a hierarchical list. Figure 3-3 shows a screen shot of Categories in action.

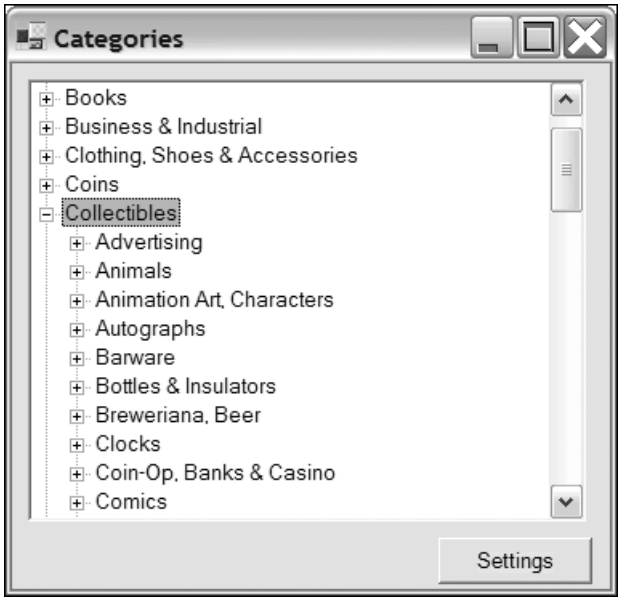

Figure 3-3. The Categories application

This application is easy to use; after launching, it downloads the top-level categories and lets you investigate subcategories by clicking the cross next to a given category. (Leaves are denoted as list items without a cross, just like in a standard hierarchical view.) Clicking the Settings button lets you enter your eBay developer, application, and certificate keys, along with a test user's account name and password. This data persists between sessions (and between the various sample applications in this book).

In doing so, the application demonstrates how to use the `eBay.SDK.API.GetCategoriesCall`, along with the `eBay.SDK.Model.ICategory` and

eBay.SDK.ModelICategoryCollection interfaces. In the discussion that follows, I present the C# implementation of Categories. Instead of a Perl implementation that performs these operations, in Chapter 6 you learn how to write the same application in Perl that uses the eBay Integration Library to minimize the load on eBay servers that your application creates.

The application's functionality can be divided up into three broad sections:

- The sample application framework, responsible for establishing the ApiSession and establishing its keys, target URL, and debugging log. The sample application also contains a secondary dialog box for entering the application keys, test user account name, and test user password. For a walkthrough of the sample application framework, see the Appendix.

- The gathering of the root categories, performed in a separate thread once the application starts to populate the top level of the list category. This is assisted by the function GetCategories, which uses the eBay.SDK.API.GetCategoriesCall method to obtain the a list of categories.

- The gathering of the categories below a specific list entry.

WARNING *In the interests of simplicity, this sample code is written to show you how to use these APIs, but does not show you the* best *way to use them. Repeatedly invoking* GetCategoriesCall.GetCategories *in your application is expensive, and presents a load to the eBay service. If your application has an upper ceiling of eBay API calls it can make, you may find that in the field your application can reach this upper limit using an algorithm such as the one presented here. Production applications should cache the results of common operations such as* GetCategoriesCall.GetCategories*, and use the cached results instead of repeating the same call over and over again. A good way to do this is to harness the power of the eBay Integration Library, which I discuss in Chapter 6.*

Starting the Categories Application

The Categories application is a single monolithic class, the CategorySample class within the com.lothlorien.ebaysdkbook namespace. Listing 3-6 shows the skeleton of the application, including its members and methods, but not the actual method implementations.

Listing 3-6. The Skeleton of the Categories Application

```
1: using System;
2: using System.Threading;
3: using System.IO;
```

```
 4: using System.Windows.Forms;
 5: using eBay.SDK;
 6: using eBay.SDK.API;
 7: using eBay.SDK.Model;
 8:
 9: namespace com.lothlorien.ebaysdkbook
10: {
11:   public class CategorySample : System.Windows.Forms.Form
12:   {
13:     // eBay Session
14:     IApiSession apiSession;
15:     // UI components
16:     private System.Windows.Forms.Button config;
17:     private System.Windows.Forms.Label status;
18:     private ConfigDialog configDialog;
19:     private System.ComponentModel.Container components = null;
20:
21:     // This application's fields
22:     private System.Windows.Forms.TreeView tree;
23:
24:     private delegate void
25:         DelegateAddRootNodes( CategoryCollection categories );
26:     private DelegateAddRootNodes AddRootNodesFunc;
27:
28:     public CategorySample()
29:     {
30:       // Inits components.
31:       // Creates a new thread to load root nodes.
32:       // Start the new thread.
33:     }
34:
35:     private void LoadRootNodes()
36:     {
37:       // Entry point for the thread to load the
38:       // root nodes and invoke the main thread
39:       // to redraw.
40:     }
41:
42:     private CategoryCollection RootNodes()
43:     {
44:       // Load the root nodes from eBay.
45:
46:     }
47:
```

```
48:     private void AddRootNodes( CategoryCollection categories )
49:     {
50:       // Add the root nodes _and_ their children
51:       // to the tree.
52:     }
53:
54:     private CategoryCollection GetCategories( int parent,
55:                                                int level )
56:     {
57:       // Issue the API call.
58:     }
59:
60:     private void tree_BeforeExpand( object sender,
61:                                     TreeViewCancelEventArgs e)
62:     {
63:       // Populate a branch when clicked.
64:     }
65:
66:     private void InitializeEBayComponent( )
67:     {
68:       // Create the API session.
69:       // Set its log file.
70:       // Set the API session's access credentials.
71:     }
72:
73:     protected override void Dispose( bool disposing )
74:     {
75:       // Dispose the components and close the logfile.
76:     }
77:
78:     #region Windows Form Designer generated code
79:     private void InitializeComponent()
80:     {
81:       // Support the IDE.
82:     }
83:     #endregion
84:
85:     [STAThread]
86:     static void Main()
87:     {
88:       // Call application's Run.
89:     }
90:
```

```
 91:     private void config_Click(object sender, System.EventArgs e)
 92:     {
 93:       // Show the configuration dialog box.
 94:     }
 95:
 96:     private void LoadKeys()
 97:     {
 98:       // Load the configuration keys and test user.
 99:     }
100:   }
101: }
```

The application begins with the usual `using` directives provided by the Microsoft Visual Studio .NET environment, along with those the application needs for the eBay SDK (lines 5–7).

The application has a handful of member variables, including the `apiSession` the application uses (line 14), the configuration button labeled "Settings" (line 16), and a simple text-based status bar (line 17), as well as the configuration form on line 18 (which I discuss in detail in the Appendix, as it has only C#-relevant code). Visual Studio .NET uses the final UI variable, `components`, for its nefarious purposes (line 19).This application uses a single `TreeView`, declared on line 22. Finally, the application declares a delegate to support invoking a `TreeView` UI method from the thread that initializes the root categories on lines 24–26.

The `TreeView`, as you may recall, is a control that lets you store a list of `TreeNode` objects, which each in turn may contain additional node lists. As the user clicks nodes, the `TreeView` sends events that manage the opening and closing of each list item to reveal the sublists below the selected node. The Category application uses this feature to download subcategories of a given selected category.

The constructor, `CategorySample`, initializes the Visual Studio .NET–generated code and then executes code that loads the various keys and test user information from a configuration file. Next, it creates and starts a new thread responsible for downloading the root categories from eBay while the application UI starts up. This gives you the opportunity to immediately see the application, even though it must perform a network transaction with eBay before it can do anything useful.

This work to load the root nodes of the tree is performed by the thread in the `LoadRootNodes` method, which first queries eBay for the root nodes, and then adds them to the `TreeView` tree. This is helped along by the helper function `GetCategories`, which actually performs the eBay SDK API request to obtain a collection of categories given a specific parent category. Once this occurs, the thread exits.

With the tree initialized, you can click a specific category to view the subcategories in that category, which generates the .NET Framework `BeforeExpand`

event. The application registers to receive this event in its constructor, so that when you click an item, it is invoked. It uses the `GetCategories` helper to obtain the categories under the category you choose, and adds them to the `tree`.

The remainder of the methods are application helper methods. The `InitializeComponent` method (lines 78–83) is created by the Visual Studio .NET application builder when you create the initial form. In a similar vein, the `InitalizeEbayComponent` (lines 66–71) initializes the `apiSession` after loading the user keys with `LoadKeys` (on lines 96–99) and starts logging eBay messages. The `Dispose` method (lines 73–76) closes the eBay log file and disposes the components. The `Main` method is the application's entry point; it simply invokes the `Application` class's `Run` method to start the application. Finally, the `config_Click` method simply invokes the configuration form, which you can see in detail in theAppendix.

Obtaining Top-Level Tree Entries

The application begins by initializing its components, the eBay components, and fetching the root-level categories for the tree view. Listing 3-7 shows the relevant methods of the application (here removed from the namespace and class declarations for brevity).

Listing 3-7. Obtaining Top-Level Tree Entries at Application Start

```
 1: public CategorySample()
 2: {
 3:   Thread loadRootNodesThread;
 4:
 5:   // Init our components, including the eBay one.
 6:   InitializeComponent();
 7:   InitializeEBayComponent();
 8:   // Set up the tree's event handlers.
 9:   this.tree.BeforeExpand +=
10:     new TreeViewCancelEventHandler( this.tree_BeforeExpand );
11:
12:   AddRootNodesFunc =
13:       new DelegateAddRootNodes( AddRootNodes );
14:   loadRootNodesThread =
15:       new Thread( new ThreadStart( LoadRootNodes ) );
16:   loadRootNodesThread.Name = "eBay Categories Thread";
17:   loadRootNodesThread.Start();
18:
19: }
20:
```

```
21: private void LoadRootNodes()
22: {
23:   CategoryCollection categories;
24:   categories = RootNodes();
25:   this.Invoke( AddRootNodesFunc,
26:           new object[] { categories } );
27: }
28:
29: private CategoryCollection RootNodes()
30: {
31:   Cursor.Current = Cursors.WaitCursor;
32:
33:   return GetCategories( 0, 2 );
34: }
35:
36: private void AddRootNodes( CategoryCollection categories )
37: {
38:   TreeNode node;
39:   if ( categories != null )
40:   {
41:     tree.BeginUpdate();
42:     foreach( Category category in categories )
43:     {
44:       if ( category.CategoryName != "" &&
45:          category.CategoryId == category.CategoryParentId )
46:       {
47:         // This is a top-level node.
48:         node = new TreeNode( category.CategoryName );
49:         node.Tag = category.CategoryId;
50:         tree.Nodes.Add( node );
51:         foreach( Category subcategory in categories )
52:         {
53:           if ( subcategory.CategoryName != "" &&
54:        subcategory.CategoryId != subcategory.CategoryParentId &&
55:        subcategory.CategoryParentId == category.CategoryId )
56:           {
57:             // This node is a child of the current parent node.
58:             TreeNode child =
59:               new TreeNode( subcategory.CategoryName );
60:             child.Tag = subcategory.CategoryId;
61:             node.Nodes.Add( child );
62:           }
63:         }
```

```
64:         }
65:       }
66:       tree.EndUpdate();
67:       status.Text = "";
68:       status.Refresh();
69:       Cursor.Current = Cursors.Default;
70:     }
71: }
```

This work begins in the constructor, `CategorySample` (lines 1–19), where you declare the thread that will load the root nodes `loadRootNodesThread` on line 3. Next, on lines 5–7, the constructor initializes first the Windows GUI components and then the eBay components using the helper functions `InitializeComponent` and `InitializeEBayComponent`. After that, the application adds the event handler for the `tree` control's `BeforeExpand` method on lines 9–10, and creates a delegate for the `AddRootNodes` function that the `loadRootNodesThread` will use to invoke the `AddRootNodes` function and update the `tree` control on lines 12–13. Finally, the method creates the `loadRootNodesThread` (lines 14–15), specifying the entry point as the `LoadRootNodes` function, and then starts the thread on line 17 after setting the thread's name to something meaningful for the display of running tasks in the Microsoft Windows Task Manager. Once the thread starts, the constructor exits, and the system invokes the `Main` method, starting the application and its event handler.

Meanwhile, while the application is accepting threads, the `LoadRootNodes` function (lines 21–27) is running in its own thread. This function first calls the `RootNodes` function, which returns a `CategoryCollection` of the root categories, and then uses the `Form` method `Invoke` to signal to the application that it should execute the `AddRootNodes` function on the main application thread.

> **NOTE** *You can* only *update controls from the main thread of an application. Consequently, if you look to multiple threads to maintain the interactivity of your application, you must create a delegate for any user interface update methods and use* `Invoke` *to ensure that they run on the main thread. If you forget to do this, your control update methods will generate an exception.*

The `RootNodes` method, on lines 29–33, uses the `GetCategories` helper to obtain the categories one and two levels deep below the category `0`, the root category after first setting the cursor to `Cursor.WaitCursor` to inform the user that a blocking operation is taking place. You need to get both the root-level categories *and* their children so that the tree view will know which nodes have children, correctly

differentiating parent nodes with a cross from leaf nodes, which have no cross and cannot be expanded.

The `AddRootNodes` method (lines 36–71), called on the main thread, actually does the work of manipulating the child nodes of the `tree` and inserting each category. The code's a little tricky, so let's take it a line at a time to understand what it does.

The code begins by declaring an empty `TreeNode` node to contain a newly created node from a category name on line 38. Then, if the incoming category list, `categories`, is not empty, it locks `tree` using its `BeginUpdate` method on line 41, to ensure that it will not process user events while the update occurs.

Lines 42–65 are the meat of this function, examining in turn each of the categories the application received from eBay. Each entry in the `categories` collection is one of two kinds of entries:

1. If the entry has a name and its unique ID in its `CategoryId` property is the same as the ID in its `CategoryParentId`, it's a root node and should be shown at the top of the list. If it's a root node, it should be inserted in the `Nodes` property of the `tree`.

2. Otherwise, it's a child category of a root node, and should be inserted in a specific `Nodes` property of a node contained in the `tree`'s `Nodes` property.

The comparison on lines 44–45 determines whether or not the specific category `category` is a root node. If it is (that is, if it has a nonempty name and its ID is its parent's ID), a new `TreeNode` with its text set to the category's `CategoryName` property is created on line 48. Next, on line 49, the node's `Tag` property (a category reserved by the `TreeNode` class for use by the application) receives the `CategoryId` of the category for that node, so that when you click the entry corresponding to that `TreeNode`, the application can fetch its subcategories.

The inner loop, on lines 51–64, tests each category in the `categories` loop to determine which of the returned categories are children of the category just added to the `tree`. You need to do this here, because there's no other good way to denote which tree nodes have children in the user interface without actually loading those children into the appropriate tree node. It's easy to detect a subcategory of a category—its `CategoryParentId` property is equal to its parent's `CategoryId` property. Lines 53–55 test this for a category selected by the `foreach` loop on line 51, also including two additional tests. The first, on line 53, is simple defensive programming, ensuring that no category without a name appears on the user interface. The second, on line 54, ensures that no top-level category is accidentally treated as a subcategory. (Failing to include the test on line 54 would make each category a subcategory of itself.) If a subcategory passes these tests, a new `TreeNode`, `child`, is created, and the `Tag` property assigned appropriately on line 60 before adding the new `child` to the `node`.

The remainder of the function, lines 66–69, undid the user interface setup that took place earlier. Line 66 unlocks the `tree`, so that it will again accept user input. Lines 67–68 clear the status line, which originally read "Downloading..." to give you some indication of what the application was doing. Finally, line 69 returns the cursor to its default, resetting it from the hourglass first set on line 31 of `RootNodes`.

Obtaining Categories

Obtaining a category list is easy—far simpler than the gymnastics you performed with the `tree` control in the last section. Listing 3-8 shows how the application gets categories using the `GetCategories` method.

Listing 3-8. The GetCategories Method

```
 1: private CategoryCollection GetCategories( int parent,
 2:                                           int level )
 3: {
 4:   CategoryCollection categories = null;
 5:   GetCategoriesCall getCategoriesCall;
 6:
 7:   // Set up the API we'll use.
 8:   getCategoriesCall =
 9:     new GetCategoriesCall( apiSession );
10:   getCategoriesCall.ErrorLevel =
11:     ErrorLevelEnum.BothShortAndLongErrorStrings;
12:   getCategoriesCall.CategoryParent = parent;
13:   getCategoriesCall.DetailLevel = 1;
14:   getCategoriesCall.LevelLimit = level;
15:   try
16:   {
17:     categories = getCategoriesCall.GetCategories();
18:   }
19:   catch( Exception e )
20:   {
21:     MessageBox.Show( this,
22:               "Exception - GetCategories call failed: " +
23:               e.Message, "Error" );
24:   }
25:   return categories;
26: }
```

This method is straightforward; the only real logic in the routine is the error handling required to keep the application running in the event that making the `GetCategories` API call goes awry. The routine takes two arguments: `parent`, which determines the parent category of interest; and `level`, which specifies how deeply the eBay service should traverse the category hierarchy in generating the response. Lines 4–5 create first the return value `categories` and then the `getCategoriesCall` variable that will contain a freshly created `eBay.SDK.API.GetCategoriesCall` object, created on lines 8–9.

The remainder of the initialization, on lines 10–14, sets the key properties for the API invocation:

- Lines 10–11 set the desired level of verbosity for error strings. When in doubt, I always choose the most verbose error strings, especially early in application development.

- Line 12 specifies the parent of the categories to be returned using the request's `CategoryParent` method. In return, the API will return the named category and its children, and its children's children, up to the level limit indicated by the request's `LevelLimit` parameter.

- Line 13 specifies the `DetailLevel` property, which defines the default level of detail, indicating that in the response all fields should be populated.

- Line 14 specifies the `LevelLimit` property, indicating how many levels below the given category should be retrieved and returned in the collection.

The `try/catch` block on lines 15–24 first attempts to invoke the master method for the `GetCategoriesCall`, `GetCategories`. This can fail for a myriad of reasons: an incorrect key, user account login or password, network congestion, and so forth. Consequently, it's imperative that you wrap each call to an eBay API with `try/catch` (or test the results of the OLE last error if you're using a language like Perl) to ensure that the request was a success, and that you control your application's behavior in case of an error.

> **WARNING** *Failing to correctly handle API failures may cause your application to fail eBay certification.*

This event handler is simple, but accomplishes the job. In the event of any error (indicated by an `Exception` or its subclass, `eBay.SDK.SDKException`), the application displays a message box showing the text from the exception (lines 21–23).

Obtaining Tree Branch Entries

The only other action the application must perform is managing the selection of a tree item, which may require that the item expand and download new categories. When a tree item is selected, if it can expand, the control issues a `BeforeExpand` event, which the application registered to receive during its constructor. When the application receives this event, it invokes `tree_BeforeExpand`, shown in Listing 3-9.

Listing 3-9. Expanding a Tree Item

```
private void tree_BeforeExpand( object sender,
                  TreeViewCancelEventArgs e)
{
  CategoryCollection subcategories;
  int baseLevel = -1;

  Cursor.Current = Cursors.WaitCursor;
  status.Text = "Downloading...";
  status.Refresh();
  subcategories = GetCategories( (int)e.Node.Tag, 3 );

  tree.BeginUpdate();
  foreach( Category subcategory in subcategories )
  {
    // Find the appropriate node to contain this subcategory.
    if ( baseLevel == -1 )
        baseLevel = subcategory.CategoryLevel;
    if ( subcategory.CategoryLevel == baseLevel + 2 )
    {
      foreach( TreeNode child in e.Node.Nodes )
      {
        if ( (int)child.Tag == subcategory.CategoryParentId )
        {
          TreeNode newChild =
            new TreeNode( subcategory.CategoryName );
          newChild.Tag = subcategory.CategoryId;
          child.Nodes.Add( newChild );
        }
      }
    }
  }
  tree.EndUpdate();
```

```
33:   status.Text = "";
34:   status.Refresh();
35:   Cursor.Current = Cursors.Default;
36: }
```

This code is very similar to the `AddRootNodes` routine you just saw. It begins by declaring a `CategoryCollection`, `subcategories` on line 4, which will contain those categories that descend from the item you selected. Next, it creates an integer, `baseLevel`, used to determine which subcategories fall below the selected category. This is necessary because although the `tree` provides a hierarchical data structure that mirrors the eBay category hierarchy, the eBay `CategoryCollection` model represents this information using a flat collection of `Category` items, each with a `CategoryLevel`.

Lines 7–9 prep the user interface for another network transaction, first selecting `Cursor.WaitCursor` and then updating the text on the status line. Line 10 requests the category selected, its children, grandchildren, and great-grandchildren. While the selected category and its children are already known (the selected category was fetched previously, and its children were fetched at the same time to determine whether this item was a leaf or a parent), there's no good way to use this information without additional data structures, so the routine just requests it again anyway.

Once the request completes, I lock the `tree` on line 12. Line 13 iterates through the resulting subcategories. Lines 16–17 use a bit of skullduggery to avoid the category node that was selected, and lines 18–30 add the grandchildren to the appropriate node using the same logic you first saw in `AddRootNodes`, ensuring that each node is correctly treated as a parent node or a leaf node.

Key Points

In this chapter, you learned the following key points regarding the eBay SDK:

- The lion's share of the eBay SDK are elements of the data model, which let you easily interact with the software analogues of real-world concepts such as users, their accounts, and the items they buy and sell.
- The `eBay.SDK.API` namespace contains classes and interfaces for each of the eBay APIs that the eBay SDK supports.
- The eBay SDK provides collections that inherit from the .NET collection interfaces to let you manage groups of identically typed components without resorting to using arrays. Moreover, collections *can* be used as arrays when needed.

- When using an `eBay.SDK.API` interface to execute an eBay API request, you must set its `ApiSession` property to an allocated and initialized `eBay.API.ApiSession` object, which contains the eBay developer, application, and certification key for your application, along with a user account ID and password.

- When you use an `eBay.SDK.API` interface to execute an eBay API request, the arguments for the request are stored as properties set by the interface, and the API itself is invoked using the interface's master method. In turn, this method returns the results to you, typically as an object defined in one of the namespaces contained by the `eBay.SDK.Model` namespace.

- You can use the eBay SDK log of your application's use of eBay SDK APIs to debug your application by setting the `ApiSession`'s `LogCallXml` property to `true` and setting its `Log` property to an initialized instance of `eBay.SDK.LogFile`.

- The eBay category system is hierarchical. You can traverse this hierarchy by using the `GetCategoriesCall` and supplying both the parent ID of the topmost-level category to get, along with a depth indicating how many levels of subcategories to return.