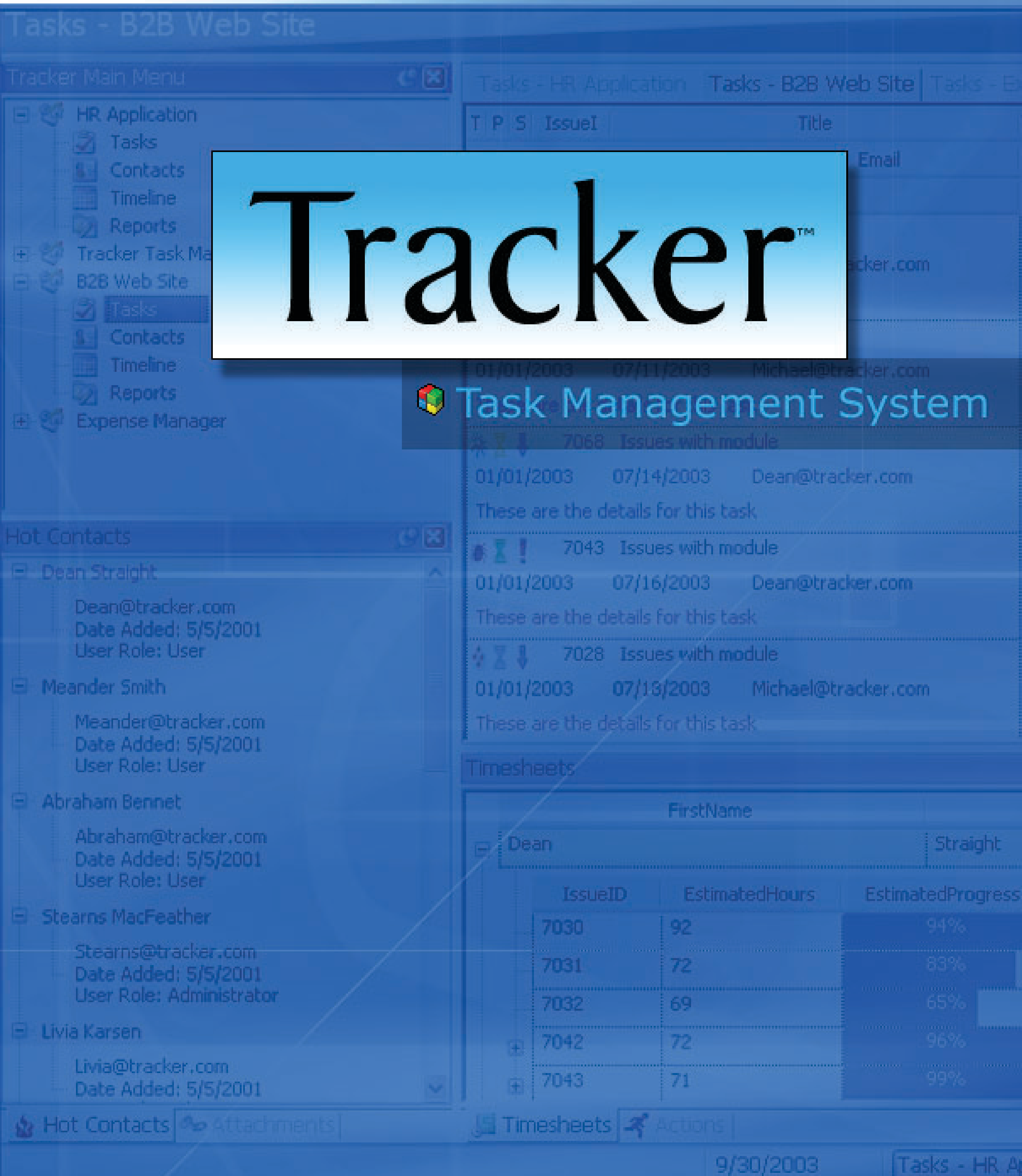
Infragistics™

Powering The Presentation Layer

Tracker™

Task Management System



01/01/2003	07/11/2003	Michael@tracker.com
7068	Issues with module	
01/01/2003	07/14/2003	Dean@tracker.com
These are the details for this task		
7043	Issues with module	
01/01/2003	07/16/2003	Dean@tracker.com
These are the details for this task		
7028	Issues with module	
01/01/2003	07/13/2003	Michael@tracker.com
These are the details for this task		

Dean Straight
Dean@tracker.com
Date Added: 5/5/2001
User Role: User

Meander Smith
Meander@tracker.com
Date Added: 5/5/2001
User Role: User

Abraham Bennet
Abraham@tracker.com
Date Added: 5/5/2001
User Role: User

Stearns MacFeather
Stearns@tracker.com
Date Added: 5/5/2001
User Role: Administrator

Livia Karsen
Livia@tracker.com
Date Added: 5/5/2001

Dean Straight

IssueID	EstimatedHours	EstimatedProgress
7030	92	94%
7031	72	83%
7032	69	65%
7042	72	96%
7043	71	99%

Tracker Task Management System e-Book

About Infragistics

Infragistics is the leading publisher of development tools for Microsoft® .NET®, COM and Java® platforms. Infragistics mission is powering the presentation layer through a complete framework of elements for all commercially viable environments and devices, while helping the professional developer maximize the benefits of Infragistics' toolsets by offering value beyond the product. www.infragistics.com

Visit DevCenter

Infragistics DevCenter, a new way to help empower the .NET developer community. Built entirely with Infragistics ASP.NET components, the DevCenter is an integrated developer community presenting comprehensive online resources on presentation layer development. With presentation layer feature articles from leading technology publications, tutorials and how-to's on Infragistics presentation layer elements, .NET and ASP.NET fully –documented reference applications, samples of .NET Windows Forms and ASP.NET Web Forms controls, access to news groups, white papers, presentation slides and more. The site is a comprehensive portal for learning about technology and furthering your existing understanding of developing applications using presentation layer tools.

Visit Infragistics DevCenter at: devcenter.infragistics.com

Infragistics
Corporate Headquarters

www.infragistics.com
800-231-8588
609-448-2000
sales@infragistics.com

Infragistics
Europe

www.infragistics.com
+44 (0) 800 298 9055
+44 (0) 20 8387 1474
sales-europe@infragistics.com

About this Book

This e-Book is a supplement to the Tracker reference application which can be downloaded from http://DevCenter.Infragistics.com/RefApps/Tracker/Tracker.aspx. This book provides technical insight to how and why the Tracker application created. Please use this as a learning tool, and feel free to distribute to all of your friends!

About the Author

Jason is a Senior Technology Evangelist for Infragistics. In addition, Jason is a Microsoft Certified Solutions Developer and a Microsoft .NET MVP. Jason is the author of "Teach Yourself Visual Studio .NET 2003 in 21 Days" from Sams, the co-author of the "Visual Basic .NET Bible" and the "C# Bible" from Wiley, and a contributor to "SQL Server 2000: The Complete Reference" from Osborne/McGraw-Hill and "ASP.NET @ Work: 10 Enterprise Applications" from Wiley. Jason is currently working on "Teach Yourself Visual C# .NET in 21 Days" from Sams, which is due out in the Q1 2004.

Acknowledgements

Writing the Tracker application and the e-Book would not have been possible without the help of others. I would like to thank very much from Infragistics, Brad McCabe, Tim Hitchings, Michael Saltzman, Ayoola Ogunsola, Adam Jaffe, Margaret Todd, Kim Plescia, and finally Dean Guida. I would thank Sharon Figel at Infragistics for her awesome graphics (especially the cool cover to the e-Book), and especially I would like to thank Sheri Nawrocki from Computer Ways who checked my writing, created the graphics used on the controls in the Tracker application, and put up with me during this whole process.

Questions, Comments and Updates

http://DevCenter.Infragistics.com/RefApps/Tracker/Tracker.aspx

Chapter
1

Chapter 1: Overview of the Tracker Application

The Tracker application is a Windows Forms application written using the .NET Framework. The purpose of this application is to demonstrate best practices for using Microsoft .NET technologies in conjunction with the advanced user interface capabilities of Infragistics controls for Windows Forms. This booklet will guide you through the creation of the Tracker application, explain key points in the design, and discuss how the user interface and back end infrastructure were created. The booklet is broken down into 10 chapters:

Chapter 1: Overview of the Tracker Application
Chapter 2: Understanding the Architecture Design and Goals
Chapter 3: Implementing the Tracker Database
Chapter 4: Implementing the Tracker Data Access Layer
Chapter 5: Implementing the Tracker Business Logic Layer
Chapter 6: Integrating the Microsoft Application Blocks in Tracker
Chapter 7: Designing the Tracker User Interface
Chapter 8: Implementing the Tracker User Interface
Chapter 9: Enhancing the Tracker User Interface
Chapter 10: Where to go from here

By reading each chapter, you will understand why the end product looks and acts like it does, how .NET fits in, and how the Infragistics controls are used. You will also get an understanding of how to modify and extend the application to fit your needs. During the process of each chapter, screen shots of the actual application will be used, so you can easily reference back to what each topic is covering.

Today, you are going to get an overview of what Tracker is, and what it can do for you, and the tools and technologies you need to move forward with modifying the solution. You are going to learn about:

- An overview of the Tracker Application
- Tracker Usage Scenarios
- Technologies used in Tracker
- Setting up Tracker
- Summary for Today and What is Next

Overview of Tracker

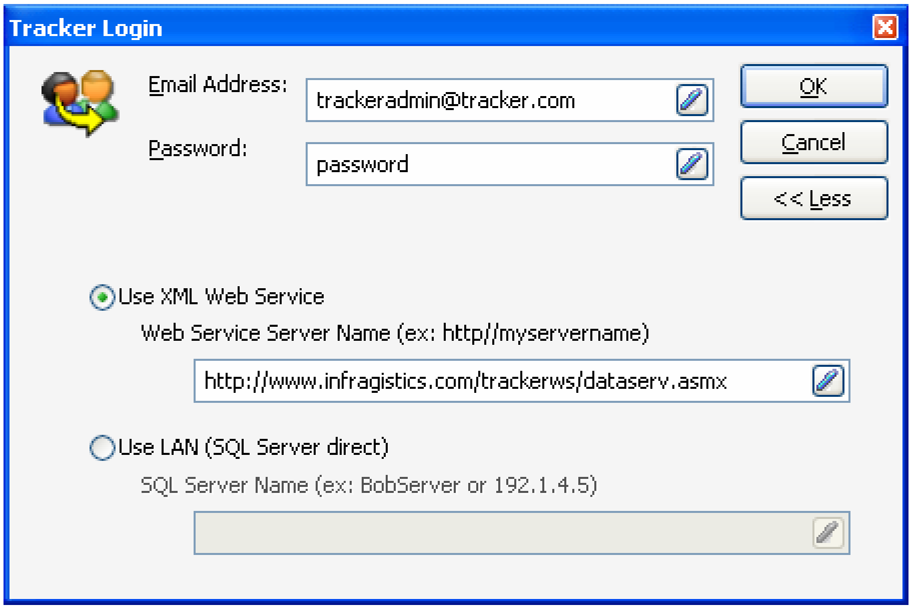
Tracker is a Windows Forms application that helps project managers or developers manage three main aspects of daily software development tasks:

- Create tasks for a project (Tasks can be generic tasks, bugs, defects, etc…)
- Track time for tasks (Time is a schedule and time per task, which in turn could be used for billing)
- Track resources, such as contacts, documents and schedule

The Tracker application does this through the use of Project Workspaces (workspaces). Workspaces contain modules that implement the flow of developing a software project. Within a workspace, you may have resources that Tracker manages for you on a workspace by workspace basis. The concept for what Tracker can do is a combination of several actual bug and defect tracking applications, as well as the TaskVision Windows Forms reference application that can be found at http://www.windowsforms.net.

A typical session when logged into Tracker might be:

Logging in to Tracker: Tracker can be configured in one of two ways for data access. You can use an XML Web Service or you can access SQL Server directly. The login screen allows you to choose where the data source for Tracker is.



Email Address: trackeradmin@tracker.com

Password: password

● Use XML Web Service

Web Service Server Name (ex: http//myservername)

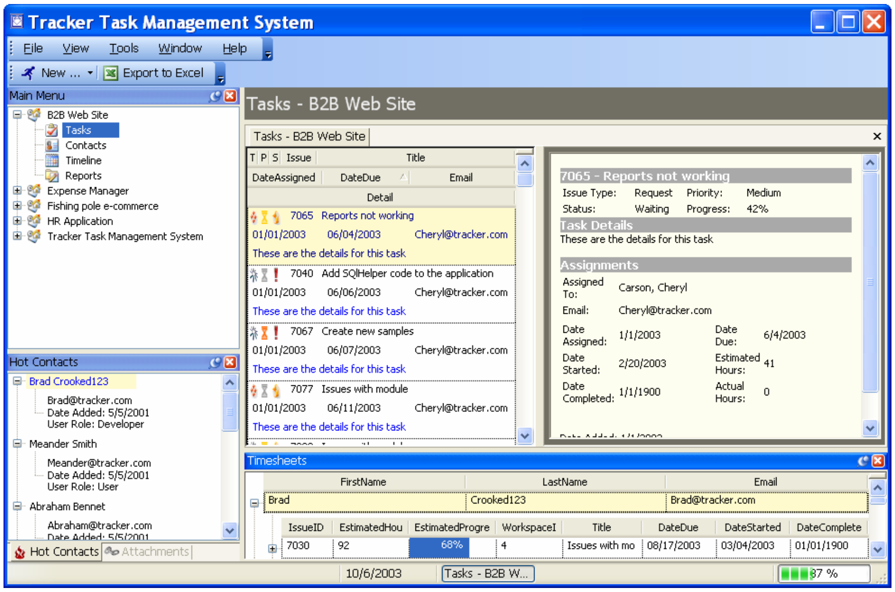
http://www.infragistics.com/trackerws/dataserv.asmx

○ Use LAN (SQL Server direct)

SQL Server Name (ex: BobServer or 192.1.4.5)

Figure 1.1: The Tracker login screen

Tasks View: All of the Tasks for the selected workspace are listed. A task can be marked as a bug, defect, suggestion, request or task. Using the context menu on the grid, or the View main menu, you can filter the display of tasks that are listed. If you look at the process of developing a piece of software, the details of an item would start as a task, then move to testing, could be a defect, may be referred back as a bug or marked as completed. Each task that you enter tracks the type of task, the status of the task, who is assigned the task, and the percentage complete of the task. Figure 1.2 shows the Task view of Tracker.



Tasks - B2B Web Site

- B2B Web Site
  - Tasks
  - Contacts
  - Timeline
  - Reports
- Expense Manager
- Fishing pole e-commerce
- HR Application
- Tracker Task Management System

T P S Issue	Title
DateAssigned / DateDue / Email	
Detail	
7065	Reports not working
01/01/2003 06/04/2003	Cheryl@tracker.com
These are the details for this task	
7040	Add SQlHelper code to the application
01/01/2003 06/06/2003	Cheryl@tracker.com
These are the details for this task	
7067	Create new samples
01/01/2003 06/07/2003	Cheryl@tracker.com
These are the details for this task	
7077	Issues with module
01/01/2003 06/11/2003	Cheryl@tracker.com
These are the details for this task	

7065 - Reports not working

Issue Type: Request Priority: Medium
Status: Waiting Progress: 42%

Task Details

These are the details for this task

Assignments

Assigned To: Carson, Cheryl
Email: Cheryl@tracker.com
Date Assigned: 1/1/2003 Date Due: 6/4/2003
Date Started: 2/20/2003 Estimated Hours: 41
Date Completed: 1/1/1900 Actual Hours: 0

Brad Crooked123
Brad@tracker.com
Date Added: 5/5/2001
User Role: Developer

Meander Smith
Meander@tracker.com
Date Added: 5/5/2001
User Role: User

Abraham Bennet
Abraham@tracker.com

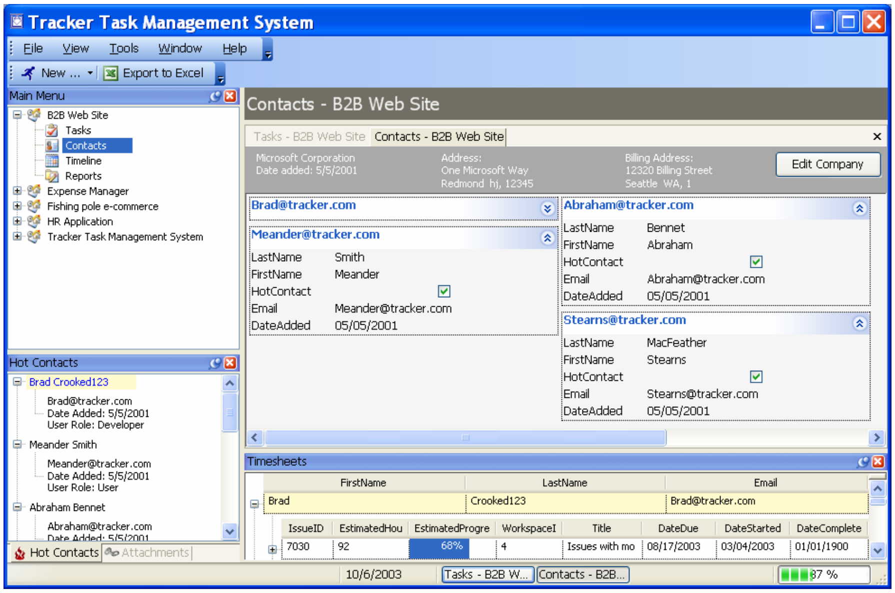
FirstName	LastName	Email
Brad	Crooked123	Brad@tracker.com

IssueID	EstimatedHou	EstimatedProgre	WorkspaceI	Title	DateDue	DateStarted	DateComplete
7030	92	68%	4	Issues with mo	08/17/2003	03/04/2003	01/01/1900

10/6/2003 Tasks - B2B W... 87 %

Figure 1.2: The Tracker Tasks view.

Contact Manager: The Contact Manager allows you to create a top level company, and then add additional people that represent that company. This is different from Outlook, where everyone is a contact. In Tracker, each Workspace is linked to a company, each company has users.



Contacts - B2B Web Site

Tasks - B2B Web Site | Contacts - B2B Web Site

Microsoft Corporation
Date added: 5/5/2001

Address:
One Microsoft Way
Redmond hj, 12345

Billing Address:
12320 Billing Street
Seattle WA, 1

Brad@tracker.com

Meander@tracker.com

LastName	Smith
FirstName	Meander
HotContact	☑
Email	Meander@tracker.com
DateAdded	05/05/2001

Abraham@tracker.com

LastName	Bennet
FirstName	Abraham
HotContact	☑
Email	Abraham@tracker.com
DateAdded	05/05/2001

Stearns@tracker.com

LastName	MacFeather
FirstName	Stearns
HotContact	☑
Email	Stearns@tracker.com
DateAdded	05/05/2001

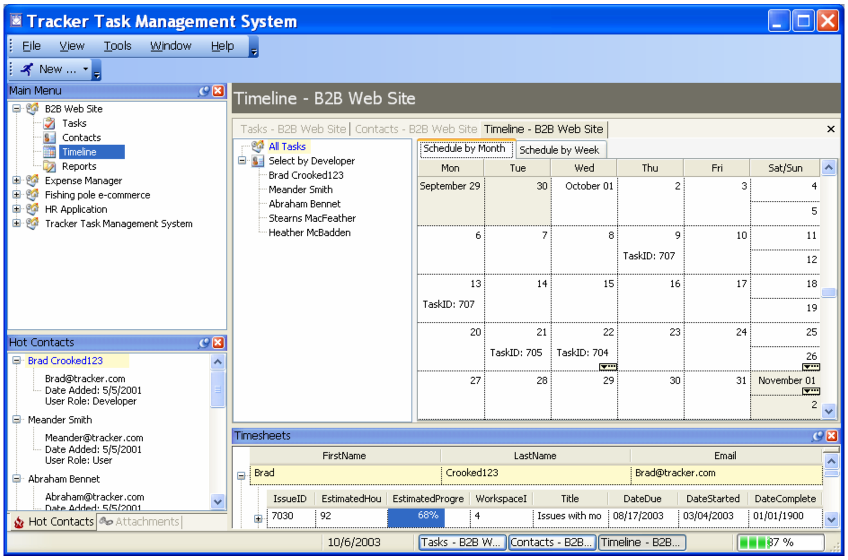
FirstName	LastName	Email
Brad	Crooked123	Brad@tracker.com

IssueID	EstimatedHou	EstimatedProgre	WorkspaceI	Title	DateDue	DateStarted	DateComplete
7030	92	68%	4	Issues with mo	08/17/2003	03/04/2003	01/01/1900

10/6/2003 Tasks - B2B W... Contacts - B2B... 87 %

Figure 1.3: The Tracker Contacts view.

Schedule and Timeline: The Schedule and Timeline module allows you to link tasks that are created by a user or manager to the developers that are working on them. This is not a full blown Microsoft Project like charting application, rather a clean interface that allows you to track what resources are being used and when.



Brad Crooked123
Brad@tracker.com
Date Added: 5/5/2001
User Role: Developer

Meander Smith
Meander@tracker.com
Date Added: 5/5/2001
User Role: User

Abraham Bennet
Abraham@tracker.com
Date Added: 5/5/2001

Mon	Tue	Wed	Thu	Fri	Sat/Sun
September 29	30	October 01	2	3	4 5
6	7	8	9 TaskID: 707	10	11 12
13 TaskID: 707	14	15	16	17	18 19
20	21 TaskID: 705	22 TaskID: 704	23	24	25 26
27	28	29	30	31	November 01 2

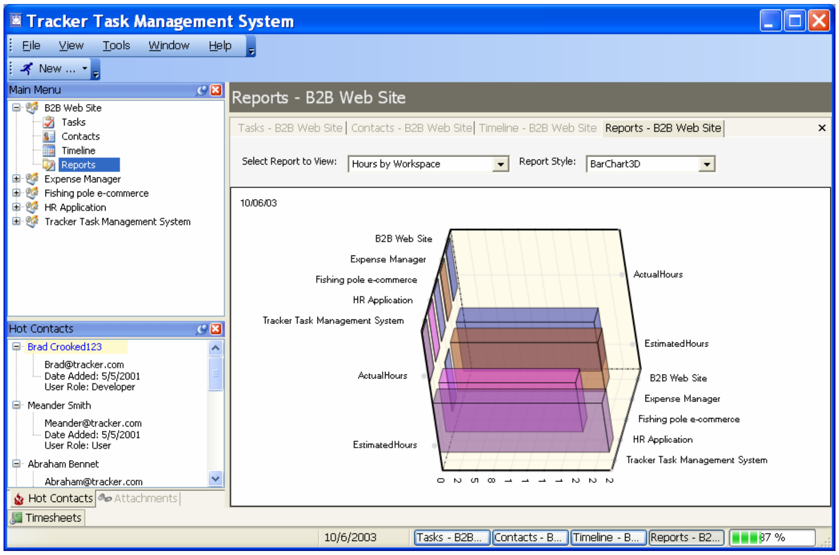
FirstName	LastName	Email
Brad	Crooked123	Brad@tracker.com

IssueID	EstimatedHou	EstimatedProgre	WorkspaceI	Title	DateDue	DateStarted	DateComplete
7030	92	68%	4	Issues with mo	08/17/2003	03/04/2003	01/01/1900

10/6/2003

Figure 1.4: The Tracker Timeline view.

Reports: Using the advanced charting capabilities of the NetAdvantage 2004 Volume 1 toolset, various reports can be viewed using the Report menu item.



Select Report to View: Hours by Workspace

Report Style: BarChart3D

Brad Crooked123
Brad@tracker.com
Date Added: 5/5/2001
User Role: Developer

Meander Smith
Meander@tracker.com
Date Added: 5/5/2001
User Role: User

Abraham Bennet
Abraham@tracker.com

10/6/2003

Figure 1.5: The Tracker Reports view

Tracker Usage Scenarios

To better understand how the Tracker might be used, I have outlined three possible usage scenarios.

- Project managers (scenario 1)
- Developers (scenario 1, 2)
- Independent Software Consultants, one-man business or small team of developers who work together on projects. (scenario 3)

Tracking tasks, managing bugs, keeping a schedule, having a central location for documents, and overall organization is always a good thing; Tracker will help you keep these things organized.

***Note**: As you will learn, the usage scenarios do not 100% represent what Tracker can do. During the development phase, you should always go back and update the usage scenarios which represent your applications. This will give anyone using the application a complete picture of what it can and cannot do.*

Scenario 1: Project Manager

ACME Tools is a manufacturing shop that makes custom parts for The Clapper. The Clapper aids common folk in turning on and off electronic devices by "clapping" their hands.

Bob, the project manager for ACME Tools, is in charge of three developers, who normally slack off. Since ACME is a family run business which has been around for 80 years, employee loyalty and family ties always seemed more important than actual production from the developers he manages. Currently Bob is charged with writing a custom application for a large customer of ACME, a nation-wide cable television channel that sells products via 900 numbers in the wee hours of the morning.

Bob wants to use this project, since it is on a tight timeline and budget, as a metric for how good his developers actually are. He turns to Tracker to help him. With Tracker, Bob can manage just what he needs to make this project a success. Once Bob installed Tracker, he created a new *Workspace*, and called it "HSN". After the Workspace was created, Bob entered all of his available resources, which were his three developers, and gave them *Developer* roles. He also entered into the system how much his resources cost him per hour, which can help Bob ask for more money in the budget next year.

In the *Developer* role, Bob knows his guys can log into Tracker, enter the Workspace, and enter time durations for tasks based on the project's requirements. Bob entered all of the tasks, but his guys can view all of the tasks for this project, as well as the percentage of the completed tasks based on the complete list of outstanding tasks. Bob also let's his guys enter in the estimated time in hours to complete a task. Bob could do this, but he knows that he should be a good manager and empower the guys to make decent estimates on time requirements.

Bob hopes this will be a motivating factor to get the job done quicker. One nice item of Tracker is that once a task is created, a complete history of the task is maintained. This includes who is assigned the task, related documents associated with the task, hourly and daily entries on when the tasks were worked on, and when a tasks was completed. Once a task is completed, he can mark it for testing and Q&A. A tester can then defer the task back to the developer with a list of defects for the task. Bob can even be notified via e-mail at any point of the process. This keeps Bob up to snuff on what is going on.

While all this is going on, Bob can get an overall picture of the project status via the Workspaces' *Start Page*, which is a bird's eye view of what is going on. He can even view and print out 3-Dimensional charts from Tracker and give it to the owner's of ACME. They like this a lot.

After three months, 389 tasks, 2605 bugs, and 72,000 man hours, Bob delivered the custom software to HSN 2 months late and $50,000.00 over budget. Using Tracker, Bob realized maybe the problem was him, and not the developers. Bob now works as a lawn-mower maintenance man for the local gas station.

Scenario 2: Developer

Kenny is a developer for ACME. Kenny's boss Bob is useless. He has no clue about software, how to manage programmers, or even about ACME's flagship product, The Clapper.
On the average day, Kenny figures he works about 3 hours. Though he punches in at 8:45am everyday, and punches out at 5:00pm everyday, Kenny spends most of his time trying to figure out how to avoid work.

Kenny's great uncle help start ACME back in 1918, so Kenny is in pretty good shape. He could never get fired from this job. Bob, Kenny's boss, just implemented a new software package called Tracker, which Bob thinks will help account for where all the time goes in a day. Kenny actually likes the product, he is trying to get is A+ Certification, and he thinks this can help him find more time during the day to study.
Each day, Kenny logs into Tracker, and gets a view of the tasks he has been assigned. The view also has a nice slider bar that has a percentage of each task being done. When Kenny clicks on a task, more details about the task are displayed to the right of the task in a big window. From here, Kenny can view all of the supporting documents he has uploaded to the server regarding this task, and why it is taking him so long to complete.

On Friday at noon, Kenny completed Task #118. He marked the task as Completed, and entered in the number of hours he worked on the task that day. Each day, Kenny can enter in the number of hours he works on a task. When he clicks the Save button on the Task Details form, an email is sent to Bob, which notifies him that the task was completed. From this point, Kenny knows that Bob will assign the task to a tester. When the tester checks the functionality, the task may be sent right back to Kenny, along with a list of defects. Each defect is on the Tests-Debugging node of the Workspace Tasks screen, and when Kenny fixes the defects for a task, he can mark them as Fixed, Not Fixable, Not a Bug, or Suspended.
For the tester, each task contains how the functionality is to be performed, and this cannot be edited by Kenny, since he is not a Manager role, he cannot fool the tester. Sometimes Kenny needs to modify a dependency on getting a task complete, such as waiting for more details from the customer, or from Bob, to actually finish the task. So he just marks a defect as Suspended, or a Task as Pending More Details, with a few notes on what the problem is.

Eventually, all Kenny cares about is that his assigned tasks are all *closed*. This means they have been coded, tested and verified, all of the defects are marked as Fixed or Not a Bug, and finally closed out by Bob. Kenny likes to switch to Calendar view to see what he has done each day. It is real easy for him to print out a daily accounting of his time, and all of the tasks he has complete.

Kenny likes Tracker, it does help him realize that his great uncle is probably disappointed in how he is helping destroy the empire he helped build. After much thought, Kenny now works full time for a moving company. Tracker made him realize writing code is way too hard, and that he is not that good at it anyway.

Scenario 3: Independent Software Consultant

Jason owns his own software consulting business. Through the years, he has built up about 15 customers. Each customer has multiple applications that Jason has written throughout the years. Jason does everything from writing applications, training developers, writing books about developing software, and writing sample applications for big companies to use as technology references.

Jason has always been fairly organized, but for years, he has been in search of the perfect software that will help him manage his many projects, and the various tasks and documents that go along with projects. Keeping track of where all the time goes in a day is always an issue, so Jason has written a custom application just to keep track of his billable hours, but that is not enough. Jason has used Groove (http://www.groove.net), Outlook, and other custom applications. But being a software developer himself, he is very picky about a nice UI and decent usability. Poking around the web, Jason found Tracker.

Tracker is the software Jason has been looking for his whole life to help him track his life. Plus, it has an awesome user interface, which fits in with other Windows XP applications, like Outlook. After Jason installed Tracker, he immediately created a new entry for each of his customers, and their pertinent information, in the Contacts section of Tracker. Once he did this, he created a Workspace for each of his current projects. Each Workspace represents a current project, and since he entered his Customers already, he could easily associate a Workspace with a Customer.

After Jason did this, he went to his file cabinet. Each project he works on has a color-coded folder. In this folder, he keeps track of all documents for the project. This documentation includes project plan, schedule, notes, screen shots, and other things associated with a project. Jason entered all of the items for *Project X* as tasks in the Project X Workspace.

As Jason entered the Tasks, he gave each task a priority, a time estimate, and a category, such as UI, Database, Code, etc. He also scanned in supporting documents for the tasks, and attached them to the task. The coolest feature in Tracker is that Jason can dictate whether to store the document in SQL Server or on the file system in a special folder, or just simply link to the documents existing location. Once Jason entered all of the tasks, he switched over to the Calendar. From this screen, Jason can drag each task item to a day or days in the Calendar; sort of creating a schedule for the project. Even though he has already committed to a deadline, he knows the next project he will use Tracker to actually estimate the time and cost for the project.

Since Jason usually works alone or with one or two other people, and he depends heavily on his customers to be the beta testers, he sets them up as the Customer role in Tracker. Since Tracker is written in .NET, it can actually run over the internet. Jason just installs the Tracker user interface application on the customer's computer, and they can access certain parts of the Tracker system. This is done via XML Web Services, using the WS-Security from the Web Services Enhancements toolkit. This makes Tracker very secure.

When a customer logs into Tracker from their office, they can view the status of project, and can test the tasks that Jason has marked as Completed. Once the customer tests out the tasks, they can add bug reports to the task, and even get a printout of project status with completed tasks, closed tasks, and fixable bugs. Jason can also leave notes for the customer to read about tasks, so they know what to test in particular. What Jason likes most is that the customer can only view certain pieces of information. They do not have full access to the screens in the program. Once they log in, they only see their workspaces.

Jason feels a lot better ever since using Tracker, he has better control over his time, and his customers have never felt more involved. He has a cool application that he can use to track projects, bugs, documents and time. Since the source-code for Tracker is available online, Jason is thinking about adding a new Billing module to Tracker, so he can print and email invoices to his customers.

Technologies used in Tracker

With the great tools from Microsoft, as well as Infragistics, key infrastructure aspects of the Tracker application are simply plugged into the Tracker application and configured for use. The Application Blocks found at the Microsoft Architecture Center at http://msdn.microsoft.com are used in Tracker, as well as the user interface elements from Infragistics.

The application blocks are provided by Microsoft as Best Practice implementations of .NET Technologies. The following application blocks are used in the Tracker application:

- Data Access Application Block

- Exception Management Application Block
- Application Updater Application Block

If you have not downloaded the Application Blocks, you can do so at the Microsoft .NET Architecture Center at: http://msdn.microsoft.com/architecture/.

Microsoft provides several more application blocks not used, and there were reasons for why some were implemented and some were not. In Chapter 6, *Integrating the Microsoft Application Blocks in Tracker*, you will learn how each application block was used and why.

Tracker Security Model

Security is always a big deal when writing applications, and Tracker is no different. Since XML Web Services are used, I decided to implement the WS-Security implementation which is part of the Web Services Enhancements Toolkit. Using WS-Security, the plumbing of a robust security model is built in, so there was no need to customize how authentication is handled.

The Tracker application can be configured not to use Web Services, so other security models can easily be implemented. In Chapter 2: *Understanding the Architecture Design and Goals*, I will explain how WS-Security is used, give you some resources to articles and books that will help you decide what security model to use in your own applications.

Programming Languages Used in Tracker

The initial release of the Tracker application is written in Visual Basic .NET using Visual Studio .NET 2003 and the .NET Framework 1.1. In order to maintain compatibility with users of Visual Studio .NET 2002, there are no language specific features used in Tracker that will break backward compatibility with .NET Framework 1.0 or Visual Studio .NET 2002.

Tracker Data Store

Tracker was designed using SQL Server 2000 Developer Edition. This means that any version of SQL Server 2000, including MSDE, can be used for Tracker. There are no specific SQL Server 2000 functions used, so the Tracker database can also be used with SQL Server 7.0. In Chapter 4, *Implementing the Tracker Data Access Layer*, you will get a better understanding of how the data access layer is implemented, and how you could port the Tracker application to Oracle.

Key Concepts used in Tracker

In order to showcase what can be done with Windows Forms in the .NET Framework; Tracker implements some other important aspects of writing robust Windows applications. Considering the Application Blocks from Microsoft are being used, as well as the WS-Security implementation, some other key concepts that Tracker showcases are:

- XML Web Services
- Complete separation of user interface, data access and business logic
- Writing a scalable .NET application that does not use ADO.NET Datasets for data manipulation
- Security and Authentication
- Professional, Clean User Interface using Infragistics controls
- Printing
- Multiple threads in a Windows Forms application

Setting up Tracker

Tracker was built using Visual Studio .NET 2003, .NET Framework 1.1, Windows Forms and ASP.NET 1.1.

You have several options when installing Tracker.

- Install source code, and work with Tracker in Visual Studio .NET
- Install client only, and interact with the public web service at Infragistics
- Install the Tracker web service and the Tracker client on your machines and test Tracker on your own client and server.

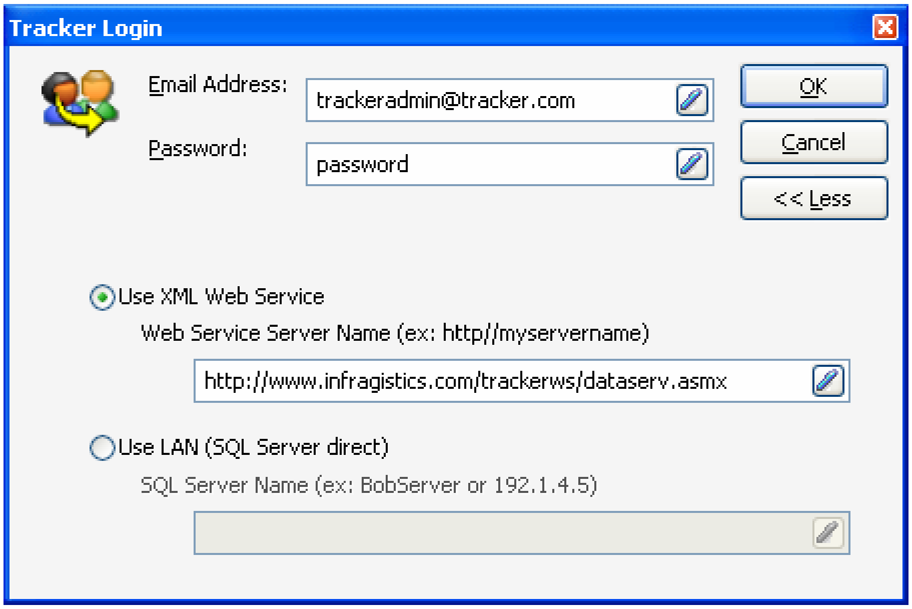
Each installer for Tracker has an option to install the database. If you want to install Tracker without installing the database, simply leave the database options blank when prompted during setup.

Client only Install

To install the client only to interact with the Tracker web service at Infragistics, you only need to install the client installer from this URL:

http://devcenter.infragistics.com/RefApps/Tracker/Tracker.Aspx

During the installation, you have the option of installing the Tracker database. You do not need to do this, since you will be accessing the database via the web service on the Infragistics servers. There will be a Tracker icon installed on your desktop, and the default login to the Infragistics web service will be filled in on the Tracker login screen, as the following picture demonstrates.


Tracker Login

Email Address: trackeradmin@tracker.com

Password: password

● Use XML Web Service

Web Service Server Name (ex: http//myservername)

http://www.infragistics.com/trackerws/dataserv.asmx

○ Use LAN (SQL Server direct)

SQL Server Name (ex: BobServer or 192.1.4.5)

If you want to install Tracker and work with SQL Server directly on your own computer, you can install the Tracker database during setup, and when you attempt to log into Tracker, change the data source location option to Use LAN, and type the name of your SQL Server, and the client will access your local data source.

Client and XML Web Service Install

To install the client and web service to work with Tracker on your own computers, install the Tracker Web Service installer and the Tracker client installer from this URL:

http://devcenter.infragistics.com/RefApps/Tracker/Tracker.Aspx

After you have the client and web service installed, you can change the connection in the Use XML Web Service option on the Tracker login screen to:

http://yourservername/trackerws/dataserv.asmx

Getting the Source Code Working with Visual Studio .NET

In order to get Tracker up and running on your development machine, and for you to effectively works with the projects in Visual Studio .NET 2003, you should install the following components. Throughout the e-Book, you will learn what role each component plays. I strongly recommend that you also read the help files for each of the application blocks. This e-Book is not a replacement for those help files.

Microsoft Web Services Enhancements (WSE) 1.0 SP1 found at
http://www.microsoft.com/downloads/details.aspx?familyid=06255a94-2635-4d29-a90c-28b282993a41&displaylang=en

WSE Settings Tool for Visual Studio .NET found at
http://www.microsoft.com/downloads/details.aspx?FamilyId=E1924D29-E82D-4D9A-A945-3F074CE63C8B&displaylang=en

Data Access Application Block
http://www.microsoft.com/downloads/details.aspx?FamilyId=F63D1F0A-9877-4A7B-88EC-0426B48DF275&displaylang=en

Exception Management Application Block
http://www.microsoft.com/downloads/details.aspx?displaylang=en&FamilyID=8CA8EB6E-6F4A-43DF-ADEB-8F22CA173E02

Updater Application Block
http://www.microsoft.com/downloads/details.aspx?FamilyId=C6C17F3A-D957-4B17-9B97-296FB4927C30&displaylang=en

Tracker source code
http://devcenter.infragistics.com/RefApps/Tracker/Tracker.Aspx

Tracker Web Service
http://devcenter.infragistics.com/RefApps/Tracker/Tracker.Aspx

Summary for Today and What is Next

You should now have a good idea of what Tracker is, why it was written, and what it can do for you. You should also have Tracker installed and configured on your machine by now. The next chapter covers the architecture of the Tracker application, and gets into the real nuts and bolts of how Tracker is designed.

Chapter
2

Chapter 2: Architecture and Design Goals

This chapter covers the architecture and design goals of the Tracker Application. When you are done reading this chapter, you will have an insight of how the Tracker application is architected, how the solution is setup in Visual Studio .NET, and what the goals were when designing the application. This will include the various layers of the application, as well as the design goals of the user interface. This chapter builds on the previous chapter, Overview of the Tracker Application, in what Tracker can actually do, and why the application was created in this manner.

The Tracker architecture is similar to several other enterprise type samples you can find on the Internet. The problem with Windows Forms that I see is there are not enough sample applications on the web that use something different than a direct data bind to a Dataset coming from either an XML Web Service or SQL Server. With Tracker, I wanted to take the same approach used by Enterprise applications like .NET Pet Shop or FMStocks, which have a clearly defined layered approach to the application architecture, without using the Dataset object.

This added more complexity to the application, since without Datasets, you lose a lot of "built-in" functionality, but it was worth the effort. The biggest complaint of using Datasets in an enterprise application is the overhead they incur, especially when used with XML Web Services. I somewhat agree and somewhat disagree with this notion, but I think it would be useful to a lot of developers (hopefully you feel the same way), to see a pretty cool Windows Forms application with a different, non-Dataset approach. Of course, using Windows Forms, you obviously want to take advantage of features such as data binding. Data binding helps you get things done fast, and even though Datasets are not used in this application, the .NET Framework fully supports data binding using objects such as ArrayList.

Understanding the Architecture

To get an understanding of Tracker, let's first look at the actual Solution file in Visual Studio .NET, project by project, and list a description of each project and why it is important to the application.

Visual Studio .NET Solution file: **Infragistics.Tracker**

Projects within the solution:

AppSettings: This project contains 1 class, *AppSettings*, which holds the serialized application settings for an individual user. The following code will give you an idea of what this class looks like. (Only a portion of the code is shown)

```
<Serializable()> _
Public Class AppSettings
        Public LastName As String
        Public FirstName As String
        Public EmailAddress As String
        Public TypeOfConnection As String
        Public LANLocation As String
        Public WSLocation As String
        Public Password As String
        Sub New ( ByVal FirstName As String , ByVal LastName As String , _
                ByVal EmailAddress As String , ByVal TypeOfConnection As String , _
                ByVal LANLocation As String , ByVal WSLocation As String , _
                              ByVal Password As String )
            Me .FirstName = FirstName
            Me .LastName = LastName
            Me .EmailAddress = EmailAddress
            Me .TypeOfConnection = TypeOfConnection
            Me .LANLocation = LANLocation
            Me .WSLocation = WSLocation
            Me .Password = Password
        End Sub
    End Class
```

Tracker: The main Windows Forms application. This project references the *Tracker.BusinessLogic* application and the *Tracker.Info* application.

Tracker. BusinessLogic: Contains the business logic methods for the Tracker application. The business logic accepts and returns the business entity objects from the *Tracker.Info* project to and from the data access layer and the client application. The following snippet gives you a glimpse of how this class is designed.

```
Namespace Tracker.BusinessLogic

    <Serializable()> _
    Public Class Company
        Inherits App
        Function GetCompany( ByVal companyID As Integer ) _
            As Tracker.Info.CompanyInfo()

            If GetConnectionString Then
                Select Case TypeOfConnection
                    Case "WS"
                         Dim proxy As TrackerWS.DataServ1Wse = GetProxy()
                        Try
                            Return proxy.GetCompany(companyID)
                        Catch ex As Exception
                            Throw New ApplicationException(Exceptions.HandleError(ex))
                        End Try
                    Case "LAN"
                        Dim x As New Tracker.DataAccess.CompanyDB
                        Return x.GetCompany(companyID)
                End Select
            End If
        End Function
```

Tracker. DataAccess: Contains the code which interacts with SQL Server. The methods in the data access layer accept and return the objects from the *Tracker.Info* project to and from the business logic layer. The following code shows the *GetCompany* method from the *CompanyDB* class in the *Tracker.DataAccess* project that is called from the *GetCompany* method shown above.

```
Namespace Tracker.DataAccess
    <Serializable()> _
    Public Class CompanyDB
        Public Function GetCompany( ByVal companyID) As CompanyInfo()
            Dim d As SqlDataReader = _
                            SqlHelper.ExecuteReader(cnString, "Company_Sel_ByID", companyID)
            Dim c As New ArrayList
            While d.Read
                Dim al As CompanyInfo = New CompanyInfo
                al.CompanyID = IIf(IsDBNull(d("CompanyID")), "", d("CompanyID"))
                al.CompanyName = IIf(IsDBNull(d("CompanyName")), "", d("CompanyName"))
                al.Address = IIf(IsDBNull(d("Address")), "", d("Address"))
                al.City = IIf(IsDBNull(d("City")), "", d("City"))
                al.State = IIf(IsDBNull(d("State")), "", d("State"))
                al.ZipCode = IIf(IsDBNull(d("ZipCode")), "", d("ZipCode"))
                al.BillToAddress = IIf(IsDBNull(d("BillToAddress")), "", _
                                      d("BillToAddress"))
                al.BillToCity = IIf(IsDBNull(d("BillToCity")), "", d("BillToCity"))
                al.BillToState = IIf(IsDBNull(d("BillToState")), "", d("BillToState"))
                al.BillToZipCode = IIf(IsDBNull(d("BillToZipCode")), "", _
                                      d("BillToZipCode"))
                al.DateAdded = IIf(IsDBNull(d("DateAdded")), "", d("DateAdded"))
                al.AddedBy = IIf(IsDBNull(d("AddedBy")), "", d("AddedBy"))
                al.ChangeStamp = IIf(IsDBNull(d("ChangeStamp")), "", d("ChangeStamp"))
                c.Add(al)
            End While
            Return CType (c.ToArray( GetType (CompanyInfo)), CompanyInfo())
        End Function
```

Tracker.Info: Business entity objects which map to the tables and fields in SQL Server. The classes in the *Tracker.Info* class are filled with serialized Arraylist types which are used in the other layers of the application. The following code is the *CompanyInfo* class, which is filled with data via the *GetCompany* method in the *CompanyDB* class, and sent from the *CompanyDB* class to its caller, which could be the *Tracker.BusinessLogic* project or the *TrackerWS* project. I am only showing a few of the fields from this class for brevity.

```
Namespace Tracker.Info
    <Serializable()> _
    Public Class CompanyInfo

        Private _companyID As Integer = 0
        Private _companyName As String = ""
        Private _address As String = ""
        Private _city As String = ""

        Sub New()
            MyBase.New()
        End Sub

        Sub New(ByVal CompanyID As Integer, _
            ByVal CompanyName As String, _
            ByVal Address As String, _
            ByVal City As String)
        End Sub

        Public Property CompanyID() As Integer
            Get
                Return _companyID
            End Get
            Set(ByVal value As Integer)
                _companyID = Integer.Parse(value)
            End Set
        End Property

        Public Property CompanyName() As String
            Get
                Return _companyName
            End Get
            Set(ByVal value As String)
                _companyName = value
            End Set
        End Property

        Public Property Address() As String
            Get
                Return _address
            End Get
            Set(ByVal value As String)
                _address = value
            End Set
        End Property

        Public Property City() As String
            Get
                Return _city
            End Get
            Set(ByVal value As String)
                _city = value
            End Set
        End Property
```

TrackerWS: Web Service which lies between the data access layer and the business logic layer. The XML Web Service accepts and returns the typed objects of the *Tracker.Info* project. The following code is the *GetCompany* method from the *DataServ.asmx.vb* class, which contains all of the public methods of the XML Web Service.

```
<WebMethod()> _
Public Function GetCompany(ByVal companyID As Integer) _
         As Tracker.Info.CompanyInfo()

    If Authenticater() Then
        Dim x As New Tracker.DataAccess.CompanyDB
        Return x.GetCompany(Integer.Parse(companyID))
    End If
End Function
```

The *TrackerWS* XML Web Service uses WS-Security from the Web Services Enhancements toolkit provided by Microsoft. Using this security model, each call to the web service is authenticated based on the hashed username and password token from the client. This guarantees the security of your data. Each method has a call to another method named *Authenticater* which uses the current SOAP context to verify that the username and password is valid by looking up the data in SQL Server. The code for the *Authenticater* method is as follows:

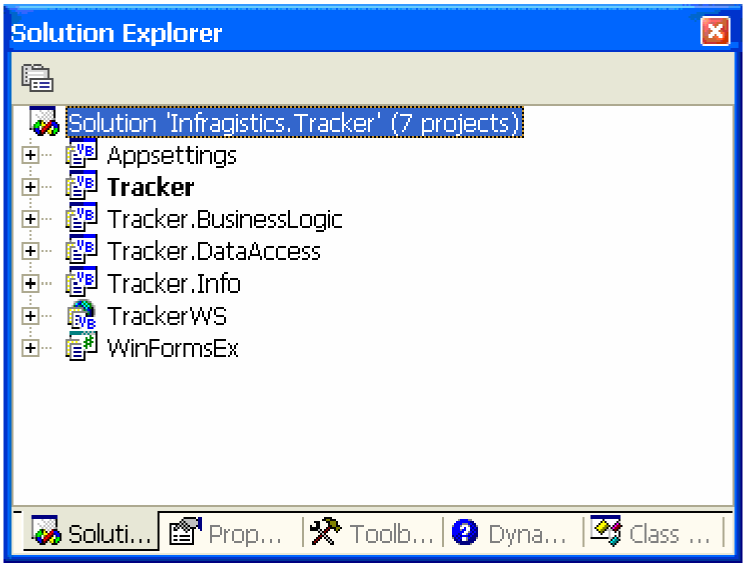
```
Private Function Authenticater() As Boolean

    Dim requestContext As SoapContext = HttpSoapContext.RequestContext
    Dim userToken As UsernameToken
    Dim returnValue As String
    If requestContext Is Nothing Then
        Throw New SoapException("Non-SOAP Message - Are you a hacker?", _
                        SoapException.ClientFaultCode)
    End If
    For Each userToken In requestContext.Security.Tokens
        If TypeOf userToken Is UsernameToken Then
            If userToken.PasswordOption = _
```

```
                    PasswordOption.SendHashed Then
                 Return True
                 Exit For
              Else
                 Throw New SoapException _
                    ("Password must be hashed", SoapException.ClientFaultCode)
              End If
           End If
        Next
     End Function
```

WinFormsEx: A helper application that displays the Tracker splash screen on the Windows Forms client. This application was found at the Microsoft Regional Directors site and is written by Juval Lowy of iDesign (http://www.idesign.com), and also has a cool implementation of the Singleton pattern to only allow a single instance of your application to run. The reason I used this application was to demonstrate how to call a C# application on another thread from a VB.NET application, plus I thought the idea of having a reusable splash screen application was very cool.

Hopefully by seeing some of the code, you will get an idea of how the Tracker application ticks. As you read the next several chapters, you will learn more about how each layer of the application is implemented.

To get an idea of what all of this looks like in Visual Studio .NET, figure 2.1 is a representation of the Solution Explorer for the *Infragistics.Tracker*.



Solution 'Infragistics.Tracker' (7 projects)
- Appsettings
- **Tracker**
- Tracker.BusinessLogic
- Tracker.DataAccess
- Tracker.Info
- TrackerWS
- WinFormsEx

Figure 2.1: The Infragistics.Tracker Solution

All of these projects may seem like a lot, but this architecture serves a purpose. It breaks up the dependency of the client on the database, and it allows you to further enhance the client application, the business logic or the database without dramatically affecting each of the respective layers. For example, while I was working on the application, I took this approach:

- Create the Database in SQL Server
- Create the Business Entity Objects - these map to the database fields.
- Create the Data Access Layer - interacts with the database, and uses the business entity objects.

- Create the Web Service - calls the data access layer directly, and returns business entity objects to the caller.
- Create the Business Logic Layer - calls the data access layer directly, or calls the XML Web Service.
- Create the Tracker client - calls the business logic layer.

As I was modifying various aspects of the application, there were times when I needed to change a field in SQL Server. Since the client code, the XML Web Service, or the business logic layer had no knowledge of the data source (because the business entity objects are holding the data in ArrayLists), all I needed to do was modify either the data access layer code or the business entity object.

This approach made it easy to make changes knowing that my client would not be broken, and during deployment, there may be times when only one of the major layers of the application needs to be re-deployed because of a change, not the entire application. Figure 2.2 shows you how all of this comes together from 10,000 feet.

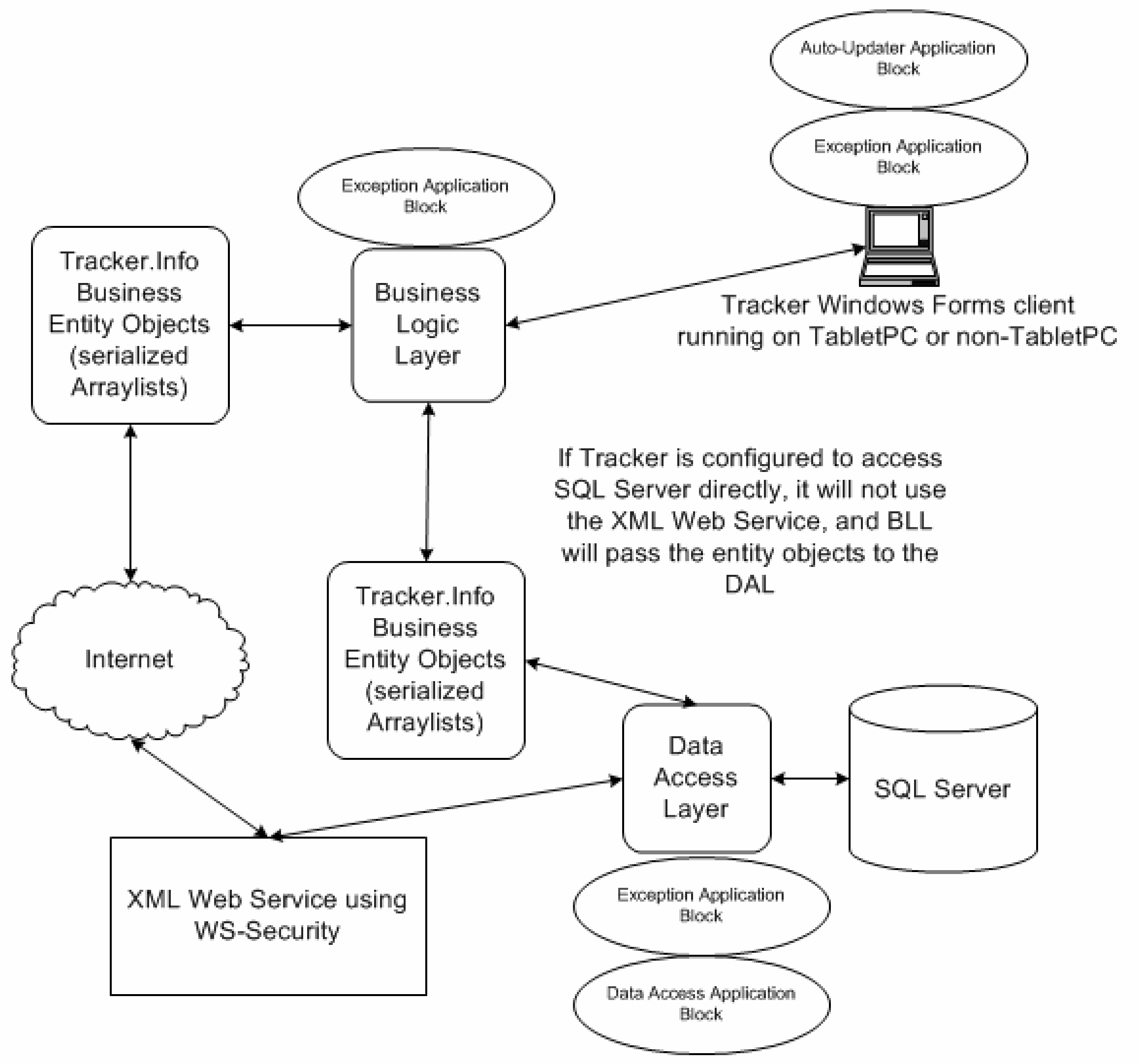

Auto-Updater Application Block
Exception Application Block
Exception Application Block
Tracker.Info Business Entity Objects (serialized Arraylists)
Business Logic Layer
Tracker Windows Forms client running on TabletPC or non-TabletPC
If Tracker is configured to access SQL Server directly, it will not use the XML Web Service, and BLL will pass the entity objects to the DAL
Internet
Tracker.Info Business Entity Objects (serialized Arraylists)
Data Access Layer
SQL Server
XML Web Service using WS-Security
Exception Application Block
Data Access Application Block

Figure 2.2: The Tracker Architecture from 10,000 feet

In figure 2.2, you can see how the various projects shown in figure 2.1 are used in the architecture. I have also shown in figure 2.2 where the application blocks from Microsoft are used. The application blocks saved a lot of code from being written, especially in the data access layer. In chapter 6, you'll learn more about the application blocks and how they are implemented.

Now that you have an idea of the architecture of Tracker, the next section looks at the design goals of Tracker, and what you should expect to learn by going through this application.

Tracker Design Goals

The design goals for tracker are simple:

- Create a cleanly architected Windows Forms application that showcases Visual Basic .NET and the .NET Framework.
- Effectively use the application blocks provided by Microsoft.
- Demonstrate a clean and effective user interface using Infragistics user interface elements of the NetAdvantage 2004 Volume 1 toolset.
- Show how to easily create an ink-enabled user interface for the Tablet PC using the ink-enabled framework of the NetAdvantage 2004 Volume 1 toolset.

Each of the next sections will highlight how these points are built into the design goals of Tracker.

Create a cleanly architected Windows Forms application that showcases Visual Basic .NET and the .NET Framework

As you read the rest of this chapter and the rest of the book, keep in mind that application architecture is a hotly debated topic by many of the smartest people that I know, all the time. I think that in each application that you write, you need to decide what is best for the task at hand. Immediately determining that an application's architecture is good or bad without knowing what the application is doing is not a good idea. You may have to write utility applications that are console based and you can write all of the code in the Sub Main of the Console1 class. This approach would suite the purpose of the task at hand.

When looking at the various Starter Kit applications found at http://www.asp.net (the official ASP.NET web site hosted by Microsoft), which showcase how to write ASP.NET applications, you will notice that each one of them is architected different. As a developer trying to learn your way around .NET, you might find it confusing as to which "best practice" is actually a best practice, since each application is architected differently. I honestly do not think that matters, since each of the applications can stand their ground in high-capacity environments. Take the IBuySpy e-Commerce application as an example. This application is being used all over the Internet, for real live e-Commerce web sites that get thousands of hits per day. I would say that is a good example of ASP.NET!

Another great example of architecture is anything that the legendary Rocky Lohtka has written. Rocky writes the "Adventures in Visual Basic .NET" column for MSDN, along with Billy Hollis, and is a famous author of books on architecture. Rocky's mission in life is to create the perfect architecture. His articles on MSDN are outstanding examples of how to write robust, reusable and scalable code, and are a must read before you design your next application. But Rocky's opinion is just that, his opinion. Not all applications need to be written using a specific data framework, nor do they need to be written to be able to handle XML Web Services, .NET Remoting, ASP.NET, Compact Framework applications and Windows Forms all at once.

The reason I am giving you my opinion on architecture is because Tracker is my opinion on a fairly cool, and I think proven, architecture. I think it is scalable, certainly reusable, and can be used by various clients, including Windows Forms, ASP.NET, .NET Remoting, Compact Framework applications, etc, without changing any of the core back end code. Writing Tracker was a fun adventure of up's and downs on dealing with some limitations of XML Web Services and how they handle serialization, but it ends up being a cool example of how to take a solid layered approach to application design. My inspiration for the Tracker architecture was the .NET Petshop application, which demonstrates ASP.NET, and was written by Microsoft to showcase .NET versus Java. The .NET Petshop shows how using .NET minimizes the amount of code needed to write an enterprise application using .NET, and how the .NET Petshop out-performs the Java Petshop 3 or 4 to 1. I did not go as deep as the .NET Petshop, which uses an Interface based approach to the application. I just separated out the layers, and show you how you can communicate in a simple way between each of the layers of the application. I really hope that you learn something by studying the application, and I really hope the code in the Tracker application will help you get your own applications done faster.

The last point of architecture decision would be the use of Visual Basic .NET. I have read over the last 2 years way too many opinions on language in .NET. Even though there are 31 total languages that take advantage of the .NET Framework, the only argument I ever hear is how C# is used for enterprise design, and Visual Basic .NET is used for "task-driven" applications. I think it is all ridiculous myself, and purely boils down to preference. I co-authored the Visual Basic .NET Bible for Wiley, but I also co-authored the C# Bible for Wiley. My Visual Studio .NET 2003 in 21 Days book from Sams has both C# and Visual Basic .NET code. The Visual C# in 21 Days from Sams I am writing at the moment, is of course, all written in C#. All of the code, either C# or Visual Basic .NET, is practically the same. Take away some brackets and semi-colons in a C# application, and it is a Visual Basic .NET application. Throughout the development of Tracker, I never thought "Man I wish that cool C# feature was available in Visual Basic .NET". It is quite the opposite. When writing the user interface, there are constant references to objects, that are nested several layers deep in classes. Just using the "With ... End With" statement in Visual Basic .NET saved me tons of typing, and possible mistakes in rapid cut and paste operations! So when it comes to language, my good buddy Jonathan Goodyear (aka AngryCoder http://www.angrycoder.com) wrote a great opinion in ASP.NET Pro magazine a few months back. His statement was that we should all be .NET Framework developers, and not get pigeon holed into a language or a platform. It will make us all better coders, and allow us to make better decisions based on the task at hand.

Effectively use the application blocks provided by Microsoft

At the .NET Architecture Center on MSDN, located at http://msdn.microsoft.com/architecture, you can find some of the most valuable information about writing robust and scalable applications using the .NET Framework. Along with the many articles and resources, you will find a link to the Patterns and Practices site on MSDN at http://www.microsoft.com/resources/practices, which showcases the Microsoft Application Blocks. The application blocks are pre-built, pluggable components written by Microsoft that provide an immediate framework for various tasks that you may need to implement in your applications. The following application blocks are available:

- Application Updater Application Block
- Asynchronous Invocation Application Block
- Caching Application Block
- Configuration Management Application Block
- Data Access Application Block
- Exception Management Application Block
- Service Aggregation Application Block
- User Interface Process Application Block

The Tracker application uses the following application blocks:

- Application Updater Application Block
- Data Access Application Block

- Exception Management Application Block

Each of the application blocks are thoroughly tested by Microsoft, and are recommended for use if your application can take advantage of the services they provide. Some are more complex than others to integrate, but overall, once you get the hang of them, you will truly be amazed at what they can offer your applications. In chapter 6, *Integrating the Application Blocks*, I will go into more detail of why I chose the three application blocks, why I did not need the other application blocks, and how they are worked into the Tracker code. At the end of the day though, the application blocks saved me from writing of many lines of code.

Demonstrate a clean and effective user interface using Infragistics user interface elements of the NetAdvantage 2004 Volume 1 Toolset

In order for an application to be accepted by end-users, it needs to be easy to use. Without a clean user interface, that mimics other applications they are used to, such as Microsoft Outlook, your hard work will be all for nothing. Users really do not care how cool or clean the underlying code is, they just care about getting their job done. Using the tools provided in the NetAdvantage 2004 Volume 1 toolset, you can rapidly create a very professional and easy to use user interface.

Many times, developers want to re-invent the wheel, write their own controls and tools. This is OK if you have the time to spend on such activities. Almost all of the time though, it is much easier to use someone else's tools, which are proven and tested. Using NetAdvantage, you have over 50 user interface elements to choose from for Windows Forms to improve the look and feel as well as the functionality of your application. In the Tracker application, all of the major user interface elements are created using the NetAdvantage 2004 Volume 1 toolset. This allows for a consistent look and feel across all forms, and allows a level of customization not available with the controls that ship with Visual Studio .NET.

***Note**: A word about Infragistics tools naming conventions: Infragistics refers to each control by tool type rather than explicit product name. You will notice internal assemblies for each control are explicitly referenced by assembly name in code, therefore they are referenced as such in the body of this booklet. However, for simplicity, all control assembly names will be shortened by dropping the "Ultra" prefix. For example, Infragistics refers to their Windows Forms grid as ".NET grid", the assembly name is "UltraWinGrid", this booklet will refer to it as "WinGrid".*

In Tracker, I took 2 approaches to creating the user interface. Since the bulk of the tutorials and samples that ship with NetAdvantage 2004 Volume 1 are all done in code, I wanted to create most of the UI with the built in designers. In all of the WinGrid elements, the Properties and Custom Properties dialogs are used to either create unbound fields or set various properties. In code, to keep each WinGrid consistent, I overrode the visual settings when the grid initializes on the Form. Using this approach, I was able to keep the look and feel of the grids consistent across all forms. Items such as Toolbars and Context Menus are consistent across each form; I simply copied the Toolbar from one form to another once I was happy with how it looked. All of the pop-up windows are inherited from a base class form that contains a Toolbar and a Tab control. This gave me the ability to add new inherited forms without worrying about what the other dialogs looked like.

I think you will really enjoy going through the code for the user interface of Tracker, it has some cool stuff, and I think it looks pretty good. Always remember, when designing a user interface, it needs to be user friendly, or users will fight you every inch of the way. Always work as close as you can with end-users when you are designing a user interface, your life will be alot easier when the application gets deployed.

Show how to easily create an ink-enabled user interface for the Tablet PC using the ink-enabled framework of the NetAdvantage 2004 Volume 1 toolset

Tablet PC's are going to be the next killer app. Once I started to use a Tablet PC, I was shocked at how cool they were. Just being able to have a fully digital notebook to take notes with changed the way I work. It is now up to all of us to write applications that target the Tablet PC. The Tracker application is Tablet PC ready. Each of the Windows Forms user interface elements in the NetAdvantage 2004 Volume 1 toolset is ink-enabled. This means that you can create a user interface using the tools from Infragistics, then simply drag the Ink Provider component onto your forms, and like magic, the controls on the form are ink aware. It cannot get any easier than that.

Note*: In order to develop ink-enabled applications, make sure you download and install the Tablet PC Platform SDK v1.5 at http://msdn.microsoft.com/library/default.asp?url=/downloads/list/windevtpc.asp.*

Summary for Today and What is Next

In this chapter, you learned about the architecture and design goals of Tracker. You learned about the projects in the Infragistics.Tracker solution, and you saw some of the code that each of the projects contains. The design goals for Tracker are pretty straightforward, to create a cleanly architected, user friendly, ink-enabled user interface using Windows Forms, Visual Basic .NET, and the Infragistics NetAdvantage 2004 Volume 1 toolset. In the next chapter, you will learn about the Tracker database, and how it was created. Each of the chapters from this point on go into more code discussion about how Tracker was built. I have listed a few links below that I hope you find useful, they are some things I referenced earlier in the chapter, and some articles that will help you get some insight to concepts covered in the chapter and the rest of the book.

Links on the World Wide Web

(This link is a must read, it contains extremely valuable information)
Application Architecture for .NET: Designing Applications and Services
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/distapp.asp

Achieve the Illusion of Handwriting on Paper When Using the Managed INK API
http://msdn.microsoft.com/msdnmag/issues/03/10/Tablet PC/

Check Out the Tablet PCs
http://msdn.microsoft.com/msdnmag/issues/03/07/editorsnote/default.aspx

.NET Architecture Center
http://msdn.microsoft.com/architecture/

Application Architecture: Conceptual View
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnea/html/eaappconapplications.asp

Distributing Objects in Visual Basic .NET
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnadvnet/html/vbnet07082003.asp

A Portal for My Data
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnadvnet/html/vbnet07082003.asp

Improving Web Application Security: Threats and Countermeasures
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnnetsec/html/ThreatCounter.asp

Books that you need to buy

Microsoft .NET Distributed Applications: Integrating XML Web Services and .NET Remoting
Author: Matthew *MacDonald* (this is my favorite book, also check out Visual Basic .NET Programmers Cookbook by Matthew *MacDonald, he is a really good author that explains complex issues in a very simple manner*)
http://www.amazon.com/exec/obidos/ASIN/0735619336/qid=1065419981/sr=2-1/ref=sr_2_1/102-8067133-2318565

Expert One-on-One Visual Basic .NET Business Objects
Author: Rockford Lhotka
http://www.amazon.com/exec/obidos/ASIN/1590591453/qid=1065420034/sr=2-1/ref=sr_2_1/102-8067133-2318565

Building Client/Server Applications Under *VB* .NET: An Example-Driven Approach
Author: Jeff Levinson
http://www.amazon.com/exec/obidos/tg/detail/-/1590590708/qid=1065420098/sr=1-1/ref=sr_1_1/102-8067133-2318565?v=glance&s=books

Chapter

3

Chapter 3: Implementing the Tracker Database

In this chapter, you will learn how the Tracker database is designed, and what steps I took to determine what the database should consist of. You will learn about the tables that are used, and how the views and stored procedures are used. Creating the database is probably the most important part of the application, since without a reasonable database design; the application would not be very useful. Even though you can simply open up Enterprise Manager and view all of the tables, reading this chapter is useful in understanding why the Tracker database works like it does.

Ideas for the Tracker Database

When I first started to figure out what the Tracker application was going to do, I investigated several other bug and task management applications. By doing a few searches on Google, I was able to download trial versions of several applications, or run online based applications that sort of did what I wanted. This insight gave me a head start on what Tracker should do.

As a software developer myself, I have used various tools in my life that helped me get things done. At the end of the day though, I normally ended up with a bunch of notebooks with lots of notes, and that is how I managed what I was doing on various projects. At the height of my organization, I had a notebook for each project I was working on, and each time I had to take notes, I wrote the data on the top of the page. This notebook, along with printed emails, went into the color-coded folder. Not very scientific, but it worked great for years. The only problem with this method was that if I grabbed the wrong notebook, I would take notes in another notebook, and then I would tell myself to remember that. Normally I did not, so you can see the reason for application like Tracker. Organization and time management is the key to success!

One of my favorite tools for managing projects is Groove, which can be found at http://www.groove.net Groove is cool because it is a peer to peer online/offline application, and has built in tools for project management. The only thing I did not like about Groove was the amount or memory it needed to run. A key concept of Groove is the Workspace, which is the parent to all things for a project. In Tracker, I decided to mimic this functionality, there is a top level Workspace that is the container for the various functions of the application.

Once wrote down all of the things that I liked about the various tools I have used in my life, and once I ran through some of the sample applications that I downloaded from the Internet, I wrote up the usage scenarios that you read in Chapter 1, Overview of the Tracker Application. Though I did not 100% stick to the original usage scenarios in the end product, writing it all out gave me a guide for what I needed to track in the database.

What is a Task, and What Do We Track

Tracker manages tasks during the process of developing software, so the key aspect of this was the status and type of task, or issue, that needed to be tracked. Looking at how Outlook handles tasks, it was pretty easy to figure out the types of status a task might have. For a type of task, I went back to the original functional specification that I wrote, and I determined that tasks could be bugs, suggestions, requests and just plain old tasks. At this point, I knew I had a main Issues table, and then 2 other tables, IssueStatus and IssueType. I also needed to prioritize the type of issue being tracked, so of course an IssuesPriority table would be needed, which would contain various priorities for an issue.

> Note:
> Originally, Tracker was going to track Issues, not Tasks, which is why the database has tables created around Issues and not Tasks. This is just a naming thing that I could not change once I got so far into the application. Keep in mind as we move forward that an Issue is a Task, and a Task is an Issue.

Knowing that I wanted to have a Workspace which was the main container for the application, I knew that each task would be linked to a workspace. So I needed a Workspace table. Being a consultant, I looked at each Workspace as a job for a client. And being a good consultant, I normally have more than one job per client. A Workspace needed to be linked to a Company, and each company would contain Users. The users would be general contacts for the company that I might need to get in touch with regarding a project, or Workspace. Users would also need to be in a certain role, indicating whether or not they can perform certain functions in the application. When a job is completed, there should be a way to update the Workspace in Tracker to indicate that it is done, so a WorkspaceStatus table would be useful for setting the status of a workspace. You can probably see how this is coming together.

Let's document what we have so far.

Table Name	Description
Workspace	Contains a description of the project, and who the project is for, which would be the Company table
WorkspaceStatus	A child table of Workspaces, which contains Status descriptions for a given Workspace.
Company	List of Customers, which would have Workspaces
Users	List of people that work for a Company
Roles	A child table of Users, which contains Role descriptions for a User, such as Manager, Developer or User
Issues	Contains the data for the actual Task being recorded, each Issue is associated with a Workspace
IssueStatus	A child table of issues, which contains Status descriptions for an Issue
IssuePriority	A child table of Issues, which contains Priority descriptions for an Issue

So far things are coming together with what tables I need in the database. Going back to the original functional specification, I also wanted to attach documents to an issue, so an IssuesAttachments table would need to be added. The IssuesAttachments table would contain a description of the document that I was attaching, and the location of the document.

Another key aspect of Tracker is keeping track of hours for a developer. A Timesheet table would need to track the amount of hours a developer has spent on a specific issue. This would allow for further customization of the application, such as printing out invoices for a client. Though this is not part of the Tracker design, I would like to add it in the future. For each action in Tracker, I wanted to keep track of the history of an issue. The history would be something like "Changed status from X to Y", or "Deleted Attachment". Keeping track of History would allow a manager to track everything that happened with a specific issue, which is useful if there is ever a problem that needs to be looked into.

By adding those additional tables, our Tracker database now looks like this:

Table Name	Description
Workspace	Contains a description of the project, and who the project is for, which would be the Company table
WorkspaceStatus	A child table of Workspaces, which contains Status descriptions for a given Workspace.
Company	List of Customers, which would have Workspaces
Users	List of people that work for a Company
Roles	A child table of Users which contains Role descriptions for a User, such as Manager, Developer or User
Issues	Contains the data for the actual Task being recorded, each Issue is associated with a Workspace
IssueStatus	A child table of issues which contains Status descriptions for an Issue
IssuePriority	A child table of Issues which contains Priority descriptions for an Issue
IssueAttachments	A child table of Issues which contains linked documents
Timesheet	A child table of Issues which tracks hours spent on an Issue
History	A child table of Issues which tracks the changes over the lifetime of an Issue
Config	Configuration table which contains the location of the where the IssueAttachments should be stored, such as a network path

The tables are now defined, so the next step is to implement the Tracker database SQL Server.

Creating the Tracker Database in SQL Server

Choosing SQL Server was the logical choice for the Tracker database. Knowing that the application would be downloaded by many people over the Internet, if I created and developed everything using SQL Server, I knew it would port directly to MSDE. MSDE is free, and pretty much does most of the things that the full-blown SQL Server can do, with a few minor limitations. Microsoft Access would not have been a good choice, because Tracker needs to be robust enough to support multiple users, plus I wanted to use stored procedure and views.

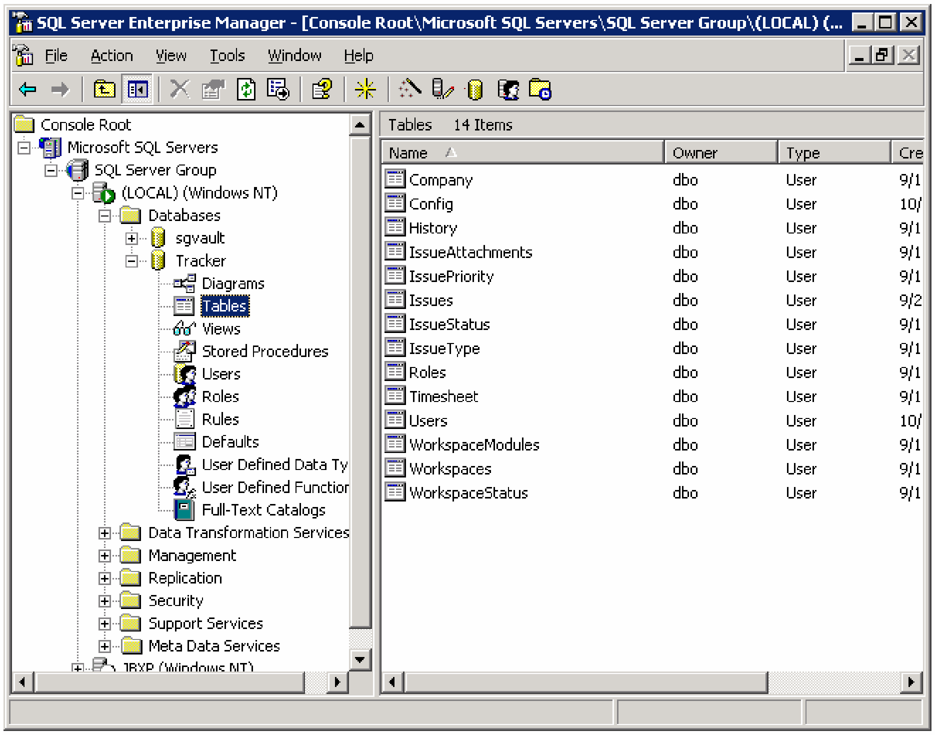
Note:
The data classes in Tracker are written in such a way that you could easily port Tracker.DataAccess project to work with Oracle. This is not part of the e-Book, but after reading chapter 4, Implementing the Data Access Layer, you will see how easily that can be done.

To get an idea of the fields that the Tracker database needs, the next sections will list each of the tables, and the fields required.

If you created the Tracker database yourself make sure you add the SQL Server user that the Tracker application expects, which is:

The *dbo* user is *trackeradmin*, with a password of *trackeradmin*.

When the Tracker database is created, your Enterprise Manager should look something like figure 3.1.

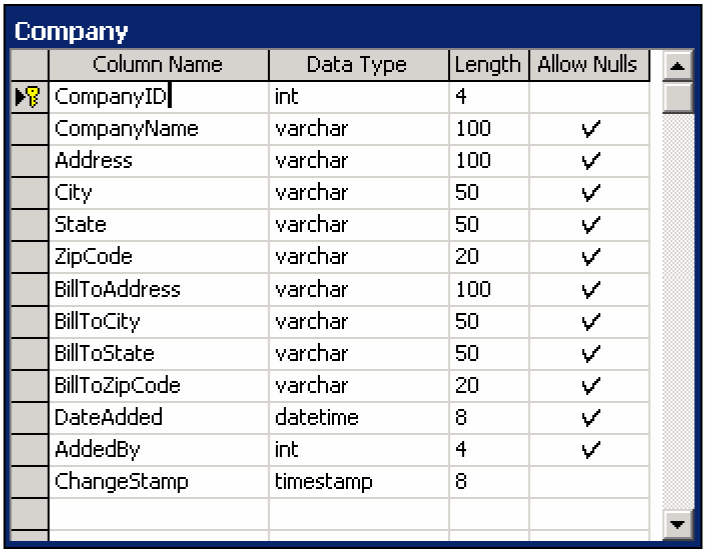


Name	Owner	Type	Cre
Company	dbo	User	9/1
Config	dbo	User	10/
History	dbo	User	9/1
IssueAttachments	dbo	User	9/1
IssuePriority	dbo	User	9/1
Issues	dbo	User	9/2
IssueStatus	dbo	User	9/1
IssueType	dbo	User	9/1
Roles	dbo	User	9/1
Timesheet	dbo	User	9/1
Users	dbo	User	10/
WorkspaceModules	dbo	User	9/1
Workspaces	dbo	User	9/1
WorkspaceStatus	dbo	User	9/1

Figure 3.1: Enterprise Manager with the Tracker database

Adding the Company Table

The Company table tracks all of the companies that have workspaces. Each record in the Workspaces table has a *CompanyID* field that maps to the *CompanyID* in the Company table. The Users table also has a *CompanyID* field, giving a one-to-many relationship between a company and a user. Figure 3.2 represents the Company table and its fields.



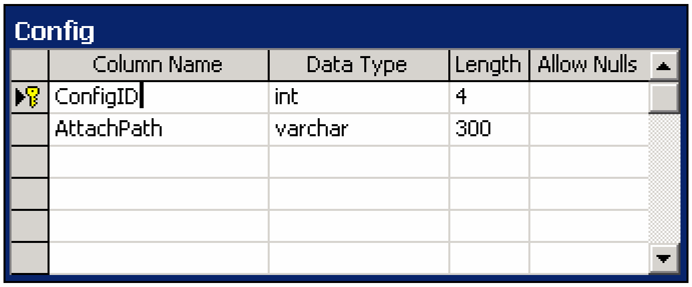
Company

Column Name	Data Type	Length	Allow Nulls
CompanyID	int	4	
CompanyName	varchar	100	✓
Address	varchar	100	✓
City	varchar	50	✓
State	varchar	50	✓
ZipCode	varchar	20	✓
BillToAddress	varchar	100	✓
BillToCity	varchar	50	✓
BillToState	varchar	50	✓
BillToZipCode	varchar	20	✓
DateAdded	datetime	8	✓
AddedBy	int	4	✓
ChangeStamp	timestamp	8	

Figure 3.2: The Company Table

Adding the Config Table

The Config table is used for storing settings for the Tracker application. At the moment, there is only one field, *AttachPath*. The *AttachPath* field is used to store the network location of the documents added to the Issues in Tracker. As the Tracker application matures, there will probably be more fields added to the Config table. Figure 3.3 is the Config table details.



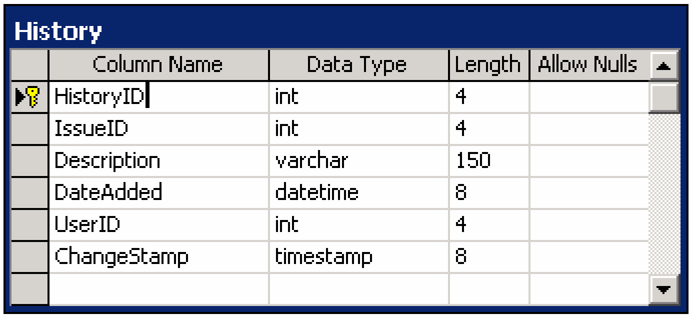
Config

Column Name	Data Type	Length	Allow Nulls
ConfigID	int	4	
AttachPath	varchar	300	

Figure 3.3: The Config Table

Adding the History Table

The History table will keep track of changes made to each item in the Issues table. This is extremely useful for tracking what changes have occurred to an item over its lifetime. In Chapter 10, *Where to Go From Here*, I will expand upon how I think the History table can be used effectively with some minor changes to the way Tracker handles each task. Figure 3.4 represents the History table.



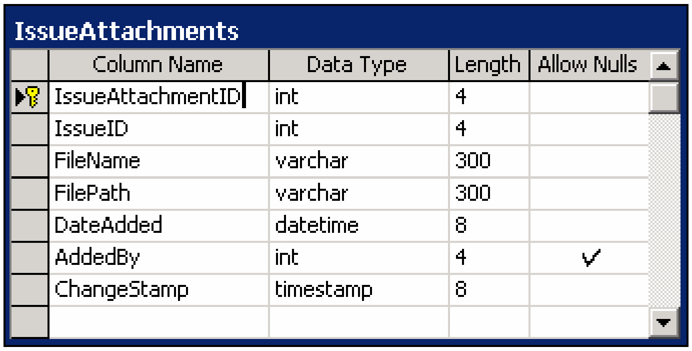
History

Column Name	Data Type	Length	Allow Nulls
HistoryID	int	4	
IssueID	int	4	
Description	varchar	150	
DateAdded	datetime	8	
UserID	int	4	
ChangeStamp	timestamp	8	

Figure 3.4: The History Table

Adding the IssueAttachments Table

The IssueAttachements table stored the description and location of attachments added for an Issue. In the Tracker application, the dialog to add an attachment simply requests a description, which is mapped to the *FileName* field in the IssueAttachments database. Using the *AttachPath* field from the Config table, Tracker copies the file to the correct location, and then adds the record to the IssueAttachments database. In figure 3.5 you can see the fields of the IssueAttachments table.



IssueAttachments

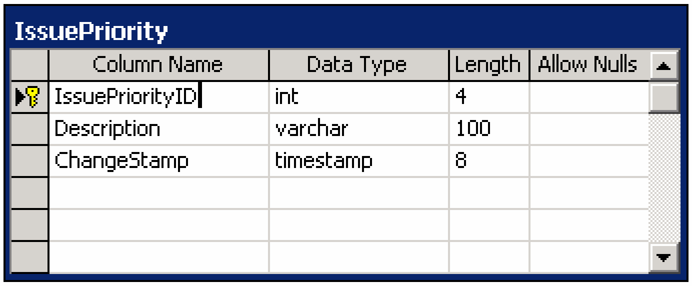
Column Name	Data Type	Length	Allow Nulls
IssueAttachmentID	int	4	
IssueID	int	4	
FileName	varchar	300	
FilePath	varchar	300	
DateAdded	datetime	8	
AddedBy	int	4	✓
ChangeStamp	timestamp	8	

Figure 3.5: The IssueAttachments Table

In my experience, this is not the best design, but will work for the Tracker application. The problem with putting the document path in the table will cause havoc if you decide to move the data to another network location. Realistically, we should be storing the filename only, and when the file needs to be retrieved, look up the path in the Config table, and build the file path in Tracker. It would also be nice, in a perfect world, if Tracker just stored the documents on a SharePoint Server, thus giving a more robust solution to managing the documents. With Microsoft Office 2003 and SharePoint Server, there are many possibilities with using .NET applications and document management features.

Adding the IssuePriority, IssueType and IssueStatus Tables

The IssuePriority, IssueType and IssueStatus tables are all the same in that they contain a *Description* field that represents each row in the table. Each Issue in the Issues table requires a priority, type and status. Using these tables, the dropdowns in the Tracker user interface are populated with the *Description* and its respective primary key. Figures 3.6, 3.7 and 3.8 show the IssuePriority, IssueType and IssueStatus tables respectively. Below each figure, there is a list of the Description rows for each table.

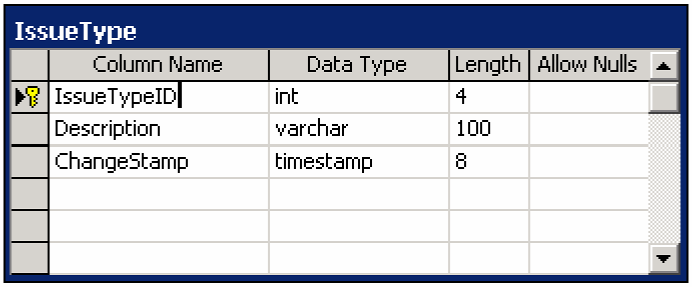


IssuePriority

Column Name	Data Type	Length	Allow Nulls
IssuePriorityID	int	4	
Description	varchar	100	
ChangeStamp	timestamp	8	

Figure 3.6: The IssuePriority Table

Sample Data:
Critical
High
Medium
Low

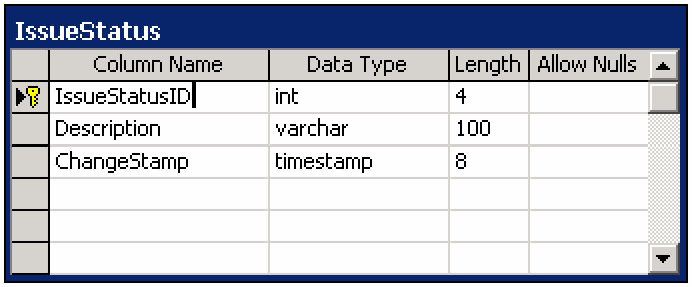


IssueType

Column Name	Data Type	Length	Allow Nulls
IssueTypeID	int	4	
Description	varchar	100	
ChangeStamp	timestamp	8	

Figure 3.7: The IssueType Table

Sample Data:
Bug
Suggestion
Request
Task



IssueStatus

Column Name	Data Type	Length	Allow Nulls
IssueStatusID	int	4	
Description	varchar	100	
ChangeStamp	timestamp	8	

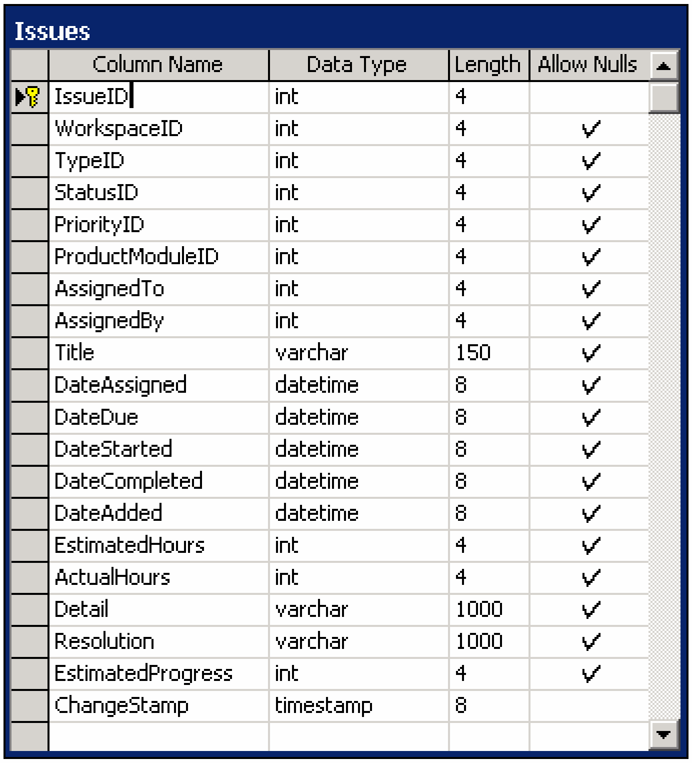
Figure 3.8: The IssueStatus Table

Sample Data:
Working On
Not Started
Waiting
Cancelled
Completed

For each of these tables, I would add another field, named *ImageID* or something similar. This field would map to the image that I want displayed in the dropdowns when a user is adding or editing a Task, or when the Tasks are displayed in the various WinGrid's throughout the application. This would allow the user interface to be more data-driven, which is always a best practice.

Adding the Issues Table

The Issues table is the heart and soul of the Tracker application. Everything revolves around the data in the Issues table. Most of the data is either a foreign-key field to another table, or a *DateTime* data type representing the various statuses of an issue. Figure 3.9 gives you an idea of what fields each Task added to the Issues table will track.



Issues

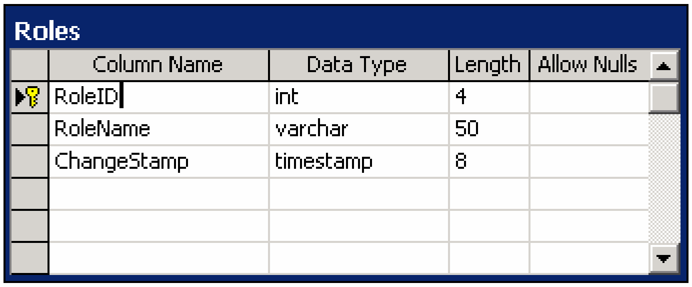
Column Name	Data Type	Length	Allow Nulls
IssueID	int	4	
WorkspaceID	int	4	✓
TypeID	int	4	✓
StatusID	int	4	✓
PriorityID	int	4	✓
ProductModuleID	int	4	✓
AssignedTo	int	4	✓
AssignedBy	int	4	✓
Title	varchar	150	✓
DateAssigned	datetime	8	✓
DateDue	datetime	8	✓
DateStarted	datetime	8	✓
DateCompleted	datetime	8	✓
DateAdded	datetime	8	✓
EstimatedHours	int	4	✓
ActualHours	int	4	✓
Detail	varchar	1000	✓
Resolution	varchar	1000	✓
EstimatedProgress	int	4	✓
ChangeStamp	timestamp	8	

Figure 3.9: The Issues Table

In the Tracker application, the *Resolution* field is not used. My idea behind *Resolution* was to allow Tasks to be turned into Knowledge Base articles. For example, if there is a bug reported in an application, then a Task is registered in Tracker. The Task sits in the Issues table marked as a *Bug*, and eventually gets resolved, or a workaround is created. In this case, the *IssueType* might be changed to *Workaround Created*, and the *IssueStatus* would be changed to *Completed*. During this process, data would be entered into the *Resolution* field, which could then be used later as a fix list for the next version of the application for the corresponding Workspace.

Adding the Role Table

The Roles table simply lists the types of roles a user might have in the Tracker application. For each User in the Users table, there is a corresponding *RoleID* field. Within Tracker, there are certain functions that only certain Roles can perform. This is a common way to handle application level access based on what type of user is logging into the system. Figure 3.10 shows you the fields in the Roles table.



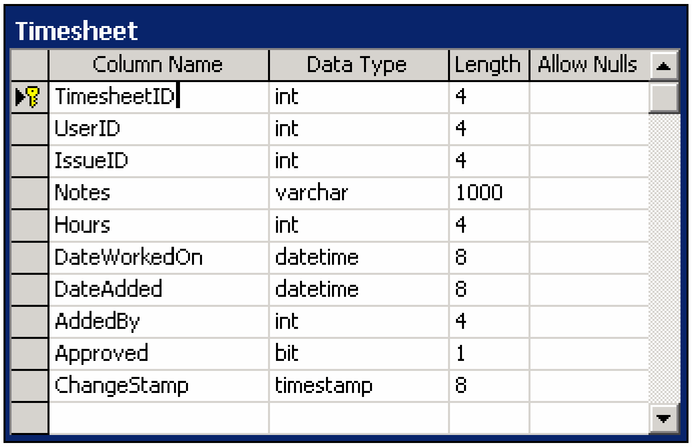
Roles

Column Name	Data Type	Length	Allow Nulls
RoleID	int	4	
RoleName	varchar	50	
ChangeStamp	timestamp	8	

Figure 3.10: The Roles Table

Adding the Timesheet Table

The Timesheet table tracks the hours and description spent on each task in the Issues table. This is done on a per user basis, meaning that each entry into the Timesheet table requires not only an *IssueID* but also a *UserID*. Using the Timesheet table, you can track the number of hours that a developer has worked on a project, and you can also track the number of hours each Issue in the Issues table has taken to complete. Figure 3.11 is the Timesheets table and its fields.



Timesheet

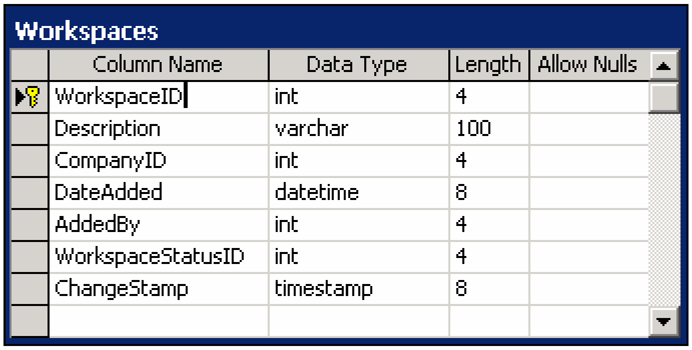
Column Name	Data Type	Length	Allow Nulls
TimesheetID	int	4	
UserID	int	4	
IssueID	int	4	
Notes	varchar	1000	
Hours	int	4	
DateWorkedOn	datetime	8	
DateAdded	datetime	8	
AddedBy	int	4	
Approved	bit	1	
ChangeStamp	timestamp	8	

Figure 3.11: The Timesheets Table

You will notice the *Approved* field in the Timesheets table. If the Tracker application were to be used as a billing tool, you would only want to bill a customer based on time that has been approved by the manager. In the next version of Tracker, I would really like to add another module that carries the Timesheet to the actual printing of an invoice to be mailed to a customer. That would be pretty cool, and pretty useful.

Adding the Workspaces and WorkspaceStatus Tables

The Workspaces table can be looked at as the "container" for the rest of the tables. Everything in Tracker resolves back to an Issue, and each Issue is associated with a Workspace. The fields for the Workspaces table are shown in figure 3.12.

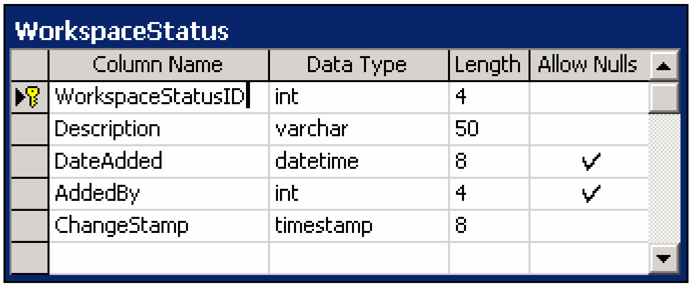


Workspaces

Column Name	Data Type	Length	Allow Nulls
WorkspaceID	int	4	
Description	varchar	100	
CompanyID	int	4	
DateAdded	datetime	8	
AddedBy	int	4	
WorkspaceStatusID	int	4	
ChangeStamp	timestamp	8	

Figure 3.11: The Workspaces Table

You will notice the *WorkspaceStatusID*. Since eventually each project you work on will be completed, you can change the status of the workspace. This is similar to changing the status of an Issue. Figure 3.12 shows the WorkspaceStatus table, with some sample data of below the figure.



WorkspaceStatus

Column Name	Data Type	Length	Allow Nulls
WorkspaceStatusID	int	4	
Description	varchar	50	
DateAdded	datetime	8	✓
AddedBy	int	4	✓
ChangeStamp	timestamp	8	

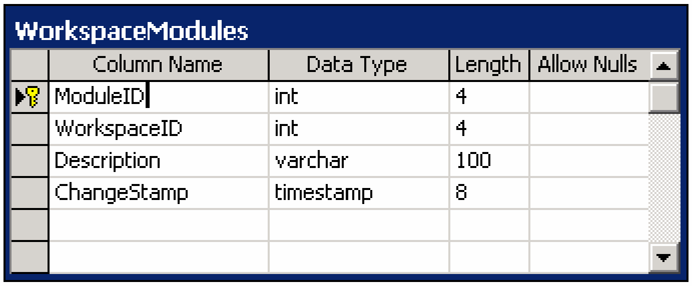
Figure 3.12: The WorkspaceStatus Table

Sample Data:
Open
Frozen - Completed
Frozen - Not Completed
Closed - Completed
Closed - Not Completed

At the moment, the Tracker application lists all of the Workspaces on the main navigation menu. A nice feature would be to toggle the items listed on the main menu of Tracker based on the status of a Workspace.

Adding the WorkspaceModules Table

The final table added to the Tracker database is the WorkspaceModules table. This table lists the modules of the application represented by Workspace. For example, if I were to create a new workspace for the Tracker application, I would add new modules which represent the steps in completing the application, such as “Create data access code for Issue table” or “Test context menu clicks”. When a task is added to the Issues table, you are also selected the module that the task represents. When you create a new Workspace, the first thing you should do is add the list of all of the modules that make up the application you are tracking. This way, when you need to add new Tasks, they will show up in the dropdown list. Figure 3.13 gives you an idea of what the WorkspaceModules table looks like.



WorkspaceModules

Column Name	Data Type	Length	Allow Nulls
ModuleID	int	4	
WorkspaceID	int	4	
Description	varchar	100	
ChangeStamp	timestamp	8	

Figure 3.13: The WorkspaceModules Table

In the original incarnation of Tracker, I had a Products table, and then a ProductsModules table. This would allow a Workspace to have many Products, or Projects, and then each Product would have modules. I thought this was a little too complex for the purposed of this application, so I added it to the wish list for version 2.

The Tracker Big Picture

Now that you have an idea of the tables in the Tracker database, and what their purpose is, you can see a portion of the relational mappings from the Issues table to each of the other tables listed in the previous sections. Figure 3.14 shows you how the tables relate to one another, and will give you a pretty good insight of what needs to go into the user interface of Tracker to actually track issues.

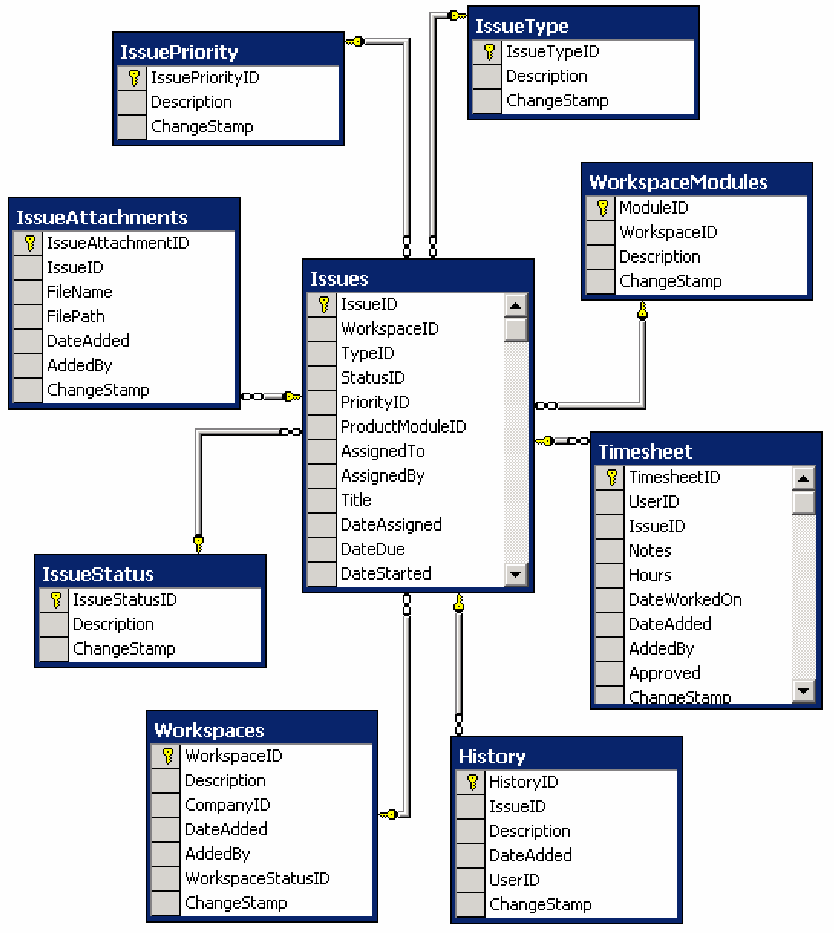


Figure 3.13: A Partial Relational Diagram of Tracker

Using Stored Procedures for Data Access

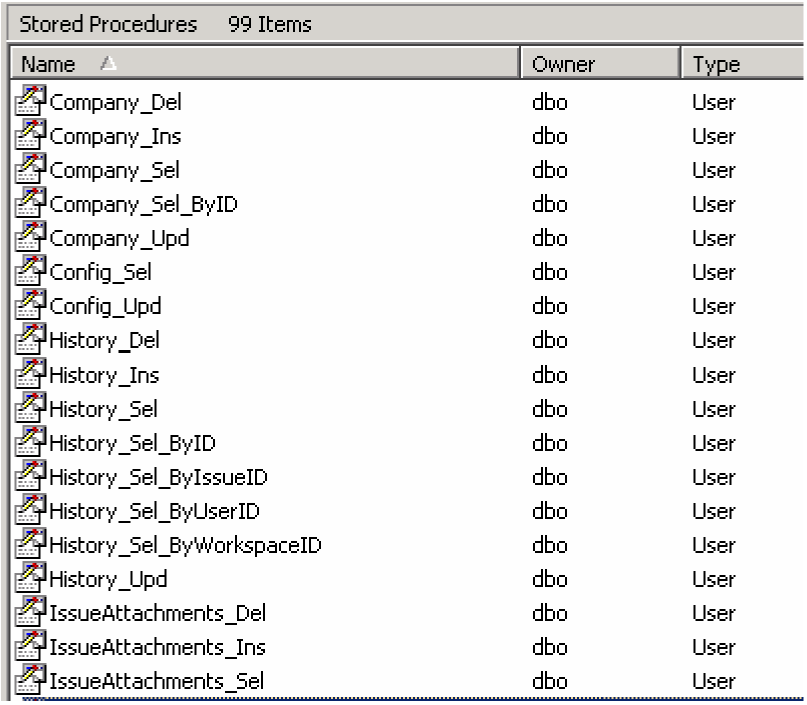
The Tracker application uses stored procedures in SQL Server for all of its data access code. I have always used stored procedures for data access, not only for the performance improvement, but also to separate the hundreds or thousands of lines of spaghetti code that can result from using ad-hoc queries in the application code. Each table in the Tracker database has a minimum of 5 stored procedures, for the following tasks:

Select All
Select by ID (primary key)
Insert
Update
Delete

The naming convention I followed for the stored procedures is:

Tablename_action

Where the *action* is either *Sel*, *Sel_ByID*, *Ins*, *Upd* or *Del*. Some tables have more than 5 stored procedures, based on the requirements of the data access layer, which you will learn about in the next chapter. To give you an idea of what the stored procedures look like in Enterprise Manager, take a look at figure 3.14.



Stored Procedures 99 Items

Name	Owner	Type
Company_Del	dbo	User
Company_Ins	dbo	User
Company_Sel	dbo	User
Company_Sel_ByID	dbo	User
Company_Upd	dbo	User
Config_Sel	dbo	User
Config_Upd	dbo	User
History_Del	dbo	User
History_Ins	dbo	User
History_Sel	dbo	User
History_Sel_ByID	dbo	User
History_Sel_ByIssueID	dbo	User
History_Sel_ByUserID	dbo	User
History_Sel_ByWorkspaceID	dbo	User
History_Upd	dbo	User
IssueAttachments_Del	dbo	User
IssueAttachments_Ins	dbo	User
IssueAttachments_Sel	dbo	User

Figure 3.14: Stored Procedure cutout from Enterprise Manager

You can see in figure 3.14 the additional stored procedures for the History table. In the following code, I have listed the Insert and Select code for the Company table's stored procedures.

```
CREATE Procedure Company_Del
                    @CompanyID int, @ChangeStamp timestamp

                    AS
                    Delete From Company Where
                    CompanyID = @CompanyID
                    AND ChangeStamp = @ChangeStamp


CREATE Procedure Company_Ins

                    @CompanyName varchar(100),
                    @Address varchar(100),
                    @City varchar(50),
                    @State varchar(50),
                    @ZipCode varchar(20),
                    @BillToAddress varchar(100),
                    @BillToCity varchar(50),
                    @BillToState varchar(50),
                    @BillToZipCode varchar(20),
                    @AddedBy int

                    AS

                    Insert Into Company(
                    CompanyName,  Address, City, State,
                    ZipCode, BillToAddress, BillToCity,  BillToState,
                    BillToZipCode, AddedBy)

                    Values (@CompanyName, @Address, @City,
                    @State, @ZipCode, @BillToAddress, @BillToCity,
                    @BillToState,  @BillToZipCode,  @AddedBy)
```

The code to notice in the Company_Del stored procedure is the where clause. We are looking at not only the CompanyID field, but the ChangeStamp field also.

```
Where CompanyID = @CompanyID AND ChangeStamp = @ChangeStamp
```

The ChangeStamp field in every table is the TimeStamp data type, which is a special byte array data type which gets updated automatically by SQL Server on a per row basis when data changes.

This gives you the ability to easily add offline access for Tracker, using the ChangeStamp field that is in every table as the indicator as to whether or not a record has changed since the data was retrieved. If the Tracker application were using Datasets, this would be built in by the Dataset always tracking the original value of a row. Since Tracker uses ArrayLists, the ChangeStamp column is the prefect solution keeping track of concurrency in the database.

Stored Procedure Soap Box

Using stored procedures is always a best practice, not only for the performance you gain by the stored procedure being compiled in SQL Server, but also as a last stop place you can go to modify the data being sent back to the caller of the procedure. Many applications use stored procedures for the business logic layer. I do not like this sort of implementation, but it certainly demonstrates the power of stored procedures. I would recommend always using stored procedure for you data access code as simple Inserts, Updates, Selects and Deletes.

Summary for Today and What is Next

In this chapter, you learned about the Tracker database, how it is designed, and how you can use the tables in the actual Tracker client application. Now that you see what the Tracker application will track, you can further enhance the tables in the database to fit your own needs. How that affects the other layers of the application is another story, which you will learn in the next several chapters. Next up, you will learn about the data access layer, and how it uses the Tracker database to serve up data to the Tracker client.

Chapter

4

Chapter 4: Implementing the Data Access Layer

In this chapter, you will learn about the data access layer in Tracker. In the previous chapter, you learned how the database was defined, and how the tables map to each other. Creating the data access layer is fairly simple once the you have all of your tables and fields defined, it is just a matter of writing the code to execute your stored procedures.

Understanding Data Access Layers

Traditionally, a data access layer, or DAL, sits between the database and an application. The application might be a Windows Forms or ASP.NET application, or it could be an XML Web Service or another assembly, such as a business logic layer. The main reason for a DAL is to allow separation from the type of data a caller is expecting based on the database and the data types in the database.

For example, in the Tracker application, the DAL has code that retrieves data from stored procedures in SQL Server, and sends the data to a caller as an ArrayList. The ArrayLists are based on specific object types, such as Company or Issue, defined in the Tracker.Info assembly, but in the end, they are just a string of data that can be consumed by any application.

If you decided that SQL Server was not going to be your database, you could simply modify the code in the DAL to read data from another source, such as Microsoft Access or Oracle. The XML Web Service, the business logic layer, and the client would not know the difference. It is this kind of flexibility that can make or break the long term maintenance of your applications. Who knows that the next version of SQL Server will offer, or if Access will all of a sudden support 500 simultaneous users, you cannot always depend on the data store being the same. You may even want to use XML files on the file system to store data. Using a DAL, you are isolating all of the data access code in your application from the dependent parts.

Figure 4.1 will give you an idea of where the DAL fits into the big picture of Tracker.

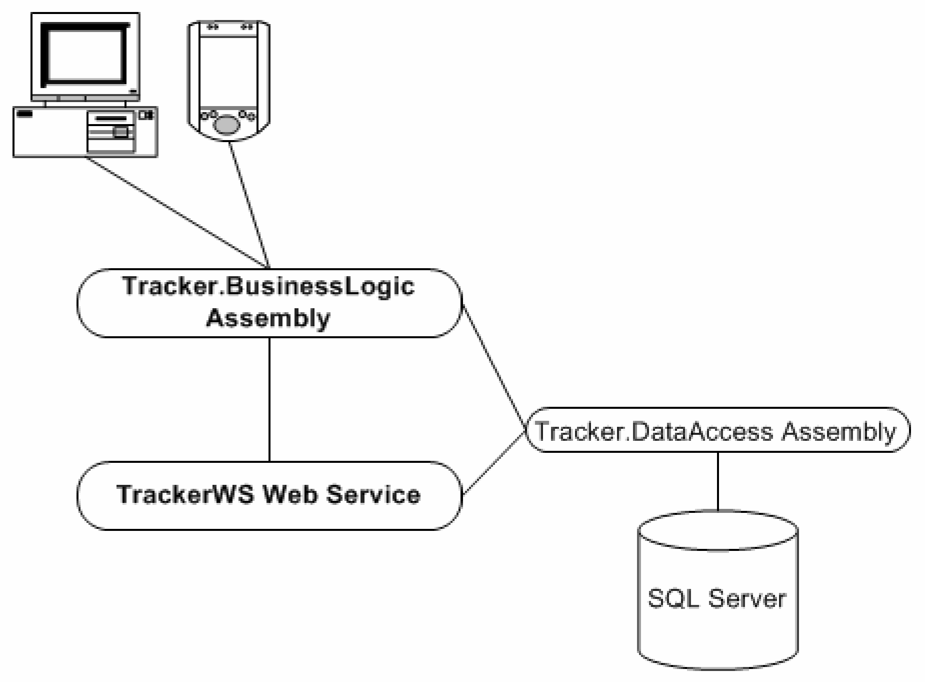


Figure 4.1: Tracker DAL Placement.

In Tracker, data is received from the user interface in 2 ways:

1. Business Logic Layer talking to the XML Web Service, which then talks to the DAL
2. Business Logic Layer talking directly to the DAL

The reason there are two ways to communicate with the DAL, and in turn, with SQL Server, is I wrote the application with two uses in mind. First, as a remote client, which would access the data via an XML Web Service, and second, an application running on a LAN, which means the XML Web Service is out of the picture. I thought this was cool to demonstrate, since it is just a matter of the business logic layer calling the correct application to get the data the client needs.

> Note:
> The DAL for Tracker is the Tracker.DataAccess project, which compiles to the Tracker.DataAccess assembly. If the application is using the XML Web Service for data access, the DAL will need to be distributed along with the XML Web Service as a private assembly. If the Tracker application is being used for LAN access, then distribute the Tracker.DataAccess assembly as a private assembly with the client application.

As I mentioned a few times before, the DAL does not return or accept the Dataset type from a caller. Everything is using ArrayLists. ArrayLists are cool because they are lightweight and flexible, and fully support data binding in either Windows Forms or ASP.NET. There are samples out on the Internet of using Datasets, and strongly typed Collection classes for data access, so I also wanted to give you another way of dealing with your data.

To understand how the ArrayLists are used by the DAL, you need to understand the Tracker.Info assembly, and what makes it tick.

Understanding the Tracker.Info Project

In order to make the client application aware of the data the DAL is producing, the Tracker.Info project contains custom objects (business entity objects) that map directly (and in some cases indirectly) to the tables in the database. These classes are marked as *Serializable*, which means they can be transported across the wire via an XML Web Service, or passed by value via .NET Remoting, and serialized and then deserialized back down on the client. In the Tracker.Info project, you will find classes that are named with the table in SQL Server, and the word *Info* at the end. I used the word *Info* because these classes actually contain all of the information needed by all consumers of the data.

To get an idea of what one of these classes looks like, the following code represents the *CompanyInfo* class.

```
Namespace Tracker.Info

    <Serializable()> _
    Public Class CompanyInfo

#Region "Private member variables"
        Private _companyID As Integer = 0
        Private _companyName As String = ""
        Private _address As String = ""
        Private _city As String = ""
        Private _state As String = ""
        Private _zipCode As String = ""
        Private _billToAddress As String = ""
        Private _billToCity As String = ""
        Private _billToState As String = ""
        Private _billToZipCode As String = ""
        Private _dateAdded As Date = DateTime.Now
        Private _addedBy As Integer = 0
        Private _changeStamp As Object = ""
#End Region

#Region "Constructors"
        Sub New()
            MyBase.New()
        End Sub

        Sub New(ByVal CompanyID As Integer, _
            ByVal CompanyName As String, _
            ByVal Address As String, _
            ByVal City As String, _
            ByVal State As String, _
            ByVal ZipCode As String, _
            ByVal BillToAddress As String, _
            ByVal BillToCity As String, _
            ByVal BillToState As String, _
            ByVal BillToZipCode As String, _
            ByVal DateAdded As Date, _
            ByVal AddedBy As Integer, _
            ByVal ChangeStamp As Object)
        End Sub

#End Region

#Region "Public Properties"
        Public Property CompanyID() As Integer
            Get
                Return _companyID
            End Get

            Set(ByVal value As Integer)
                _companyID = Integer.Parse(value)
            End Set
        End Property

        Public Property CompanyName() As String
            Get
                Return _companyName
            End Get

            Set(ByVal value As String)
                _companyName = value
            End Set
        End Property

        Public Property Address() As String
            Get
                Return _address
            End Get

            Set(ByVal value As String)
                _address = value
            End Set
        End Property

        Public Property City() As String
            Get
                Return _city
            End Get

            Set(ByVal value As String)
                _city = value
            End Set
        End Property

        Public Property State() As String
            Get
                Return _state
            End Get

            Set(ByVal value As String)
                _state = value
            End Set
        End Property

        Public Property ZipCode() As String
            Get
                Return _zipCode
            End Get

            Set(ByVal value As String)
                _zipCode = value
            End Set
        End Property

        Public Property BillToAddress() As String
            Get
                Return _billToAddress
            End Get
            Set(ByVal value As String)
                _billToAddress = value
            End Set
        End Property

        Public Property BillToCity() As String
            Get
                Return _billToCity
            End Get

            Set(ByVal value As String)
                _billToCity = value
            End Set
        End Property

        Public Property BillToState() As String
            Get
                Return _billToState
            End Get

            Set(ByVal value As String)
                _billToState = value
            End Set
        End Property

        Public Property BillToZipCode() As String
            Get
                Return _billToZipCode
            End Get

            Set(ByVal value As String)
                _billToZipCode = value
            End Set
        End Property

        Public Property DateAdded() As Date
            Get
                Return _dateAdded
            End Get

            Set(ByVal value As Date)
                _dateAdded = DateTime.Parse(value)
            End Set
        End Property

        Public Property AddedBy() As Integer
            Get
                Return _addedBy
            End Get

            Set(ByVal value As Integer)
                _addedBy = Integer.Parse(value)
            End Set
        End Property

        Public Property ChangeStamp() As Object
            Get
                Return _changeStamp
            End Get

            Set(ByVal value As Object)
                _changeStamp = value
```

```
            End Set
        End Property

#End Region

    End Class

End Namespace
```

The *CompanyInfo* class is simply a representation of the fields in the Company table of the Tracker database. There are a few things to take notice about the code. Consider these lines of code:

```
Namespace Tracker.Info

    <Serializable()> _
    Public Class CompanyInfo
```

The Namespace declaration of Tracker.Info puts this type in the Tracker namespace, which is used in each project. The <Serializable()> attribute on the class makes this class serializable. If this were not included, the type would not be able to get deserialized at the consumer.

In the private member declarations:

```
#Region "Private member variables"
        Private _companyID As Integer = 0
        Private _companyName As String = ""
        Private _address As String = ""
        Private _city As String = ""
        Private _state As String = ""
        Private _zipCode As String = ""
        Private _billToAddress As String = ""
        Private _billToCity As String = ""
        Private _billToState As String = ""
        Private _billToZipCode As String = ""
        Private _dateAdded As Date = DateTime.Now
        Private _addedBy As Integer = 0
        Private _changeStamp As Object = ""
#End Region
```

Each field is set to an initial value, based on the data type of its private member variable. This is necessary for using the New keyword at the client when implementing data binding on the type. If you were to create a new instance of this type, and attempt to extract the instance from an array of the same types, and then data bind the object, you would get null reference exception errors. This is kind of a pain in the neck, but definitely needs to be done to make this work when data binding at the client.

The remainder of the code in the class is the Public Property declarations for the fields from the database, which Get and Set the values of the private member variables.

```
        Public Property CompanyID() As Integer
            Get
                Return _companyID
            End Get

            Set(ByVal value As Integer)
                _companyID = Integer.Parse(value)
            End Set
```

```
End Property

Public Property CompanyName() As String
    Get
        Return _companyName
    End Get

    Set(ByVal value As String)
        _companyName = value
    End Set
End Property
```

The classes in the Tracker.Info assembly perform only 1 function … to hold data. Next, let's look at how to put data into these classes, and get data out of them.

Using the Tracker.Info Objects from the DAL

The classes in the Tracker.DataAccess project have all of the code that interacts with SQL Server. When selecting data from the database, the rows of data from SQL Server are added to ArrayLists based on a specific object type from Tracker.Info assembly. These ArrayLists are then casted to the correct type for consumption by either the XML Web Service or the business logic layer. If data is being inserted or updated to SQL Server, then the consumer will pass into the methods of the DAL the specific type that is being manipulated, and the DAL will read the values from that type.

To see this in action, let's take a look at the code that will retrieve all of the records from the Company table in the Tracker database.

Retrieving Data from The DAL

The following code is located in the *CompanyDB* class of the *Tracker.DataAccess* project. The method being called is named GetCompany, and it returns a data type of *CompanyInfo()*, which represents an array of the type *CompanyInfo* from the *Tracker.Info.CompanyInfo* type.

```
Public Function GetCompany() As Tracker.Info.CompanyInfo()

    ' Call the Company_Sel stored procedure, which returns
    ' all of the records in the company table
    Dim d As SqlDataReader = SqlHelper.ExecuteReader _
            (ConnectionSettings.cnString, _
            CommandType.StoredProcedure, "Company_Sel")


    ' Create a new ArrayList datatype
    Dim c As New ArrayList

    ' Loop thru the records in the SqlDataReader
    While d.Read

        ' Create a new CompanyInfo object
        Dim al As CompanyInfo = New CompanyInfo

        ' Add the data from SQL Server to the
        ' corresponding field in the COmpanyInfo object
        '
        ' NOTE:  Some fields removed for brevity
```

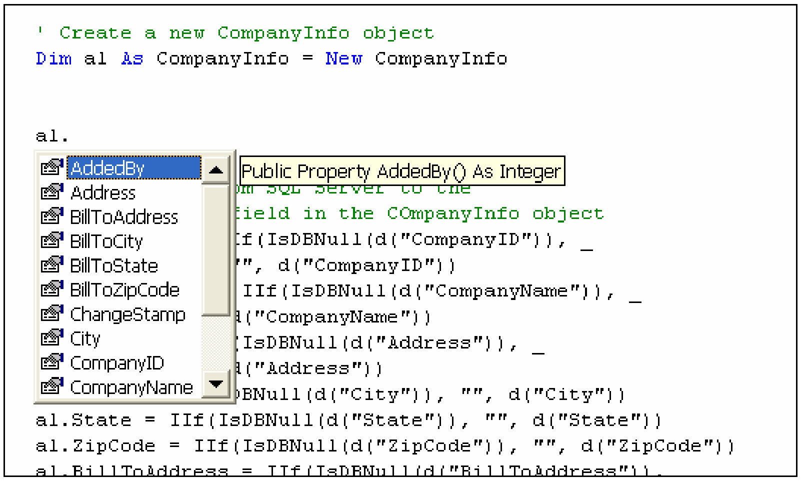
```
                al.CompanyID = IIf(IsDBNull(d("CompanyID")), _
                                "", d("CompanyID"))
                al.CompanyName = IIf(IsDBNull(d("CompanyName")), _
                            "", d("CompanyName"))
                al.Address = IIf(IsDBNull(d("Address")), _
                            "", d("Address"))
                al.City = IIf(IsDBNull(d("City")), "", d("City"))

                ' Add the CompanyInfo object to the ArrayList
                c.Add(al)
            End While

            ' Cast the ArrayList to the type CompanyInfo()
            ' and return it to the caller
            Return CType(c.ToArray(GetType(CompanyInfo)), CompanyInfo())
        End Function
```

As you can see, it is a fairly simple process. The nice thing about Visual Studio .NET is the auto-list members capability. Since the Tracker.Info assembly is referenced in the Tracker.DataAccess project, Visual Studio .NET reads its accessible members at design time to give you auto-list members on the type. After this line of code:

```
                ' Create a new CompanyInfo object
                Dim al As CompanyInfo = New CompanyInfo
```

The variable *al* is now of type *CompanyInfo*, so when referencing this variable in the IDE, you see something like figure 4.2.

Figure 4.2: Visual Studio .NET Auto-List Members on Custom Types



```
' Create a new CompanyInfo object
Dim al As CompanyInfo = New CompanyInfo


al.
                  om SQL Server to the
                  field in the COmpanyInfo object
                  If(IsDBNull(d("CompanyID")), _
                  "", d("CompanyID"))
                   IIf(IsDBNull(d("CompanyName")), _
                  d("CompanyName"))
                  (IsDBNull(d("Address")), _
                  d("Address"))
                  BNull(d("City")), "", d("City"))
al.State = IIf(IsDBNull(d("State")), "", d("State"))
al.ZipCode = IIf(IsDBNull(d("ZipCode")), "", d("ZipCode"))
al.BillToAddress = IIf(IsDBNull(d("BillToAddress")),
```

Now that you see how to get data into the custom type, what do you do with it on the client?

This code:

```
            ' Cast the ArrayList to the type CompanyInfo()
            ' and return it to the caller
            Return CType(c.ToArray(GetType(CompanyInfo)), CompanyInfo())
```

Will return the array of *CompanyInfo* objects, and because of the serilization capabilities of the .NET Framework, you can access this data from a consumer as a type of *CompanyInfo*.

Take a look at this code, which could be used from a Windows Forms application to get all of the companies, and list them out to the console window.

```
Dim _companies As Tracker.Info.CompanyInfo()
Dim _company As Tracker.info.CompanyInfo
Dim _dal As New Tracker.DataAccess.CompanyDB

' Fill the _companies varaible with
' the data from the GetCompany method
_companies = _dal.GetCompany()

' Loop thru the array of companies
For Each _company In _companies
    Console.WriteLine(_company.CompanyName)
Next
```

If you simply wanted to data bind the data in the *_companies* type, you could simply to this:

```
' Fill the _companies varaible with
' the data from the GetCompany method
_companies = _dal.GetCompany()

' Bind to grid
DataGrid1.DataSource = _companies
```

Finally, if you wanted to retrieve a specific *CompanyInfo* item from the *CompanyInfo()* array, and then data bind just a single item to controls on a forms, you could use this code:

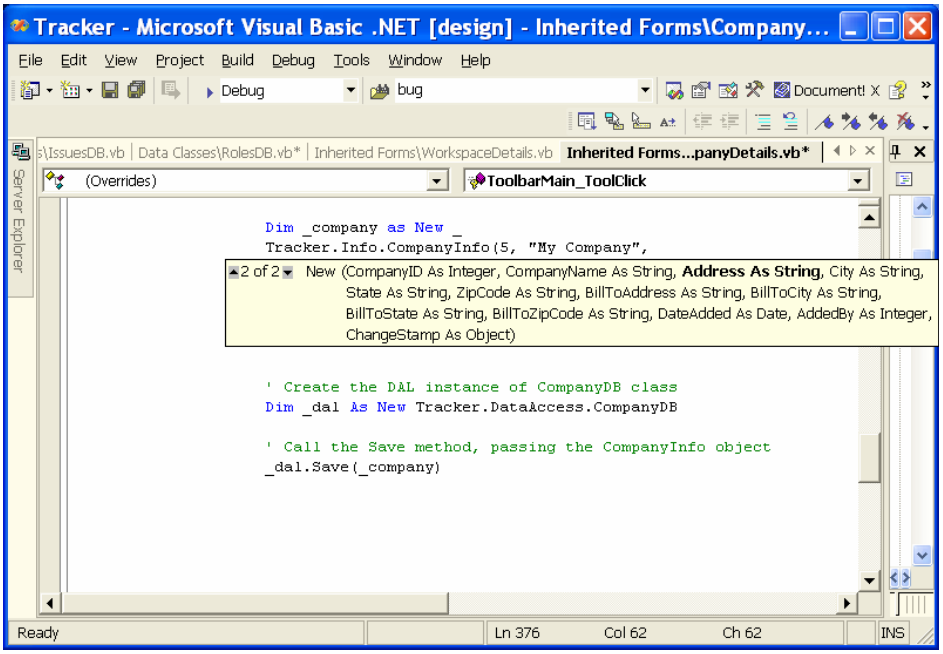
```
' Get the CompanyInfo item at the 0 index
_company = _companies.GetValue(0)

' Setup the data binding, let all the magic happen
Me.Address.Data bindings.Add("Text", _company, "Address")
Me.City.Data bindings.Add("Text", _company, "City")
```

Updating Data to the DAL

The *Save* and *Update* methods of the DAL accept the custom types defined in the Tracker.Info assembly, and then read the information from the type to insert or update into the database. From a client, you would fill the *CompanyInfo* object with data, and then simply pass the object to the method of the DAL.

In figure 4.3 I have strategically placed the *Save* code of the *CompanyDB* class below the tool tip-help of Visual Studio .NET. This way, you can see how the *CompanyInfo* object is filled with information, and then passed to the DAL.

Figure 4.3: Adding a New Company


Tracker - Microsoft Visual Basic .NET [design] - Inherited Forms\Company...
File Edit View Project Build Debug Tools Window Help
Debug
bug
Document!
s\IssuesDB.vb | Data Classes\RolesDB.vb* | Inherited Forms\WorkspaceDetails.vb | Inherited Forms...panyDetails.vb*
(Overrides)
ToolbarMain_ToolClick
Server Explorer
Ready
Ln 376
Col 62
Ch 62
INS

```
Dim _company as New _
Tracker.Info.CompanyInfo(5, "My Company",
```

2 of 2 New (CompanyID As Integer, CompanyName As String, **Address As String**, City As String, State As String, ZipCode As String, BillToAddress As String, BillToCity As String, BillToState As String, BillToZipCode As String, DateAdded As Date, AddedBy As Integer, ChangeStamp As Object)

```
' Create the DAL instance of CompanyDB class
Dim _dal As New Tracker.DataAccess.CompanyDB

' Call the Save method, passing the CompanyInfo object
_dal.Save(_company)
```

In the *Save* method of the DAL, the *CompanyInfo* object is taken, the field values are read, added as *SqlParameters* to the *SqlParameters* collection, and the updated to the database.

An Insert to SQL Server happens if the *CompanyID* field is a zero, and an Update will occur if there is a value greater than zero in the *CompanyID* field.

I have removed some of the fields in the following code to make it more readable.

```
Public Function Save(ByVal company As CompanyInfo) As Boolean
    If company.CompanyID = 0 Then
        ' Insert a new Company
        Dim params() As SqlParameter = New SqlParameter(9) {}
        params(3) = New SqlParameter("@State", company.State)
        params(4) = New SqlParameter("@ZipCode", company.ZipCode)
        params(5) = New SqlParameter("@BillToAddress",
        params(9) = New SqlParameter("@AddedBy", company.AddedBy)
        SqlHelperParameterCache.CacheParameterSet(cnString, _
                "Company_Ins", params)
        SqlHelper.ExecuteNonQuery(cnString, _
                CommandType.StoredProcedure, "Company_Ins", params)
    Else
        ' Update the existing Company
        Dim params() As SqlParameter = New SqlParameter(11) {}
        params(0) = New SqlParameter("@CompanyID", company.CompanyID)
        params(3) = New SqlParameter("@City", company.City)
        params(4) = New SqlParameter("@State", company.State)
        params(10) = New SqlParameter("@AddedBy", company.AddedBy)
        SqlHelperParameterCache.CacheParameterSet(cnString, _
                "Company_Upd", Params)
        SqlHelper.ExecuteNonQuery(cnString, _
                CommandType.StoredProcedure, "Company_Upd", Params)
    End If
    Return True
 End Function
```

In figure 4.3, you saw how to add data to the *CompanyInfo* object using the *New* constructor. In Windows Forms, if you are using data binding, then you do not need to do anything, you can simply call the *Save* method, and pass it the data bound object. For example, in this code:

```
        ' Get the CompanyInfo item at the 0 index
        _company = _companies.GetValue(0)

        ' Setup the data binding, let all the magic happen
        Me.Address.Data bindings.Add("Text", _company, "Address")
        Me.City.Data bindings.Add("Text", _company, "City")
```

The *_company* object can be passed to the *Save* method after you are done updating the fields in the form. This is used extensively in the Tracker application; you can look at any of the code in the *Inherited Forms* folder of the Tracker Windows Forms project to see this in action.

Using The Microsoft Data Application Block

As you were looking at the code for the various ways to interact with the custom entity objects and the DAL, you probably noticed these two lines of code:

```
                SqlHelperParameterCache.CacheParameterSet(cnString, _
                        "Company_Upd", Params)
                SqlHelper.ExecuteNonQuery(cnString, _
                        CommandType.StoredProcedure, "Company_Upd", Params)
```

SQLHelper is the class that ships with the Microsoft Data Application Block, which contains all of the code you need to access SQL Server. The only thing you need to do is pass the connection string, the name of the stored procedure, and the parameters of the stored procedure, and the *SQLHelper* class will handle the rest.

In Chapter 6, *Integrating the Application Blocks*, you are going to learn more about how this is implemented.

Summary for Today and What is Next

In this chapter, you learned about the data access layer, or DAL, used in the Tracker application. You also learned how the business entity objects in the Tracker.Info class are critical to the Tracker implementation. You should see that using the Info classes gives you the same power and flexibility as using a Dataset, but without the additional information passed with the Dataset object. Moving forward, you should consider implementing a DAL is your own applications as a way to ensure the separation of the client and the database, giving you more flexibility in what your back end data serves up to the consumers of the data.

Chapter

5

Chapter 5: Implementing the Business Logic Layer

In this chapter, you will learn about the business logic layer in Tracker, and the purpose it serves. You learned in the previous chapter that the data access layer sites between the consumer of the data and the SQL Server database and returns business entity objects based on the tables in SQL Server to the caller. In the Tracker client application, the business logic layer is used in the Windows Forms to communicate with either the DAL directly or with the XML Web Service. You are going to learn how this is accomplished in the business logic layer, and how the XML Web Service is used to send and receive data from the DAL.

Understanding Business Logic Layers

Like the DAL you learned about in the previous chapter, a business logic layer, or BLL, is used to separate application functionality from the client application itself. This is not to say the client application has no logic, rather it means the client does not contain any business rules that act upon the data. The reason for a BLL is not only to have a cleaner, reusable code base, but to take logic out of the client that might change based on the data requirements, thus affecting the actual application.

In the days of ASP and COM+, you may have written Visual Basic 6 ActiveX DLL's that contained the business logic for your application, and then communicated data to and from the client. This is a similar approach, except in Tracker, we are not using COM+ Services.

> Note:
> The BLL for Tracker is the Tracker.BusinessLogic project, which compiles to the Tracker.BusinessLogic assembly. The BLL is distributed as a private assembly with the Tracker client application.

To refresh your memory on how the pieces of Tracker work together, notice in figure 5.1 how the Tracker.BusinessLogic assembly is intercepting all calls to and from the actual client application. The client does not know anything about where its data is coming from, it simply interacts with the BLL and asks for information, or sends information to the BLL that needs to be inserted or updated in the database.

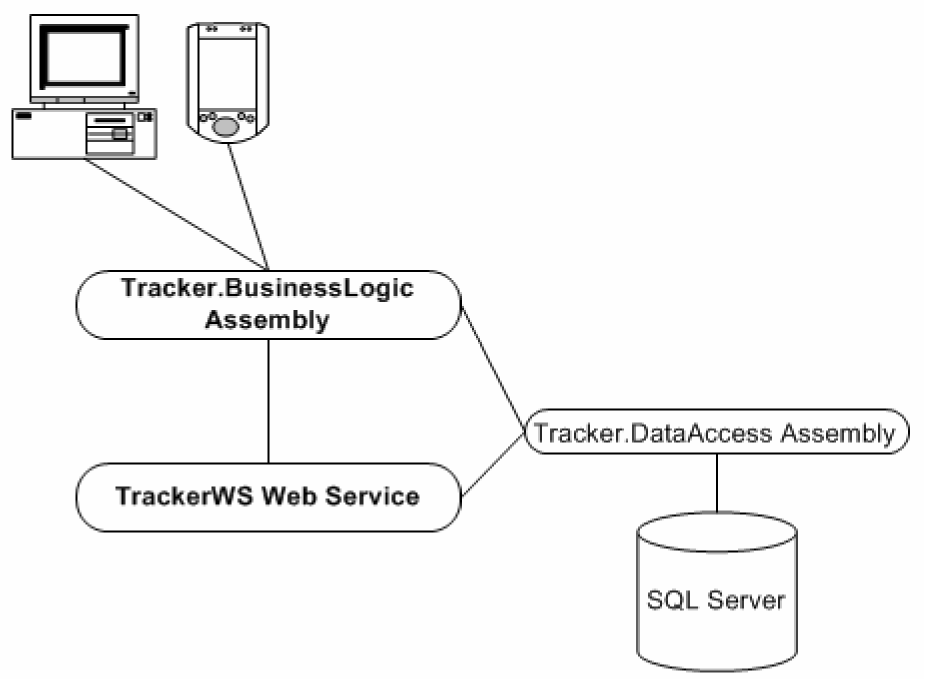


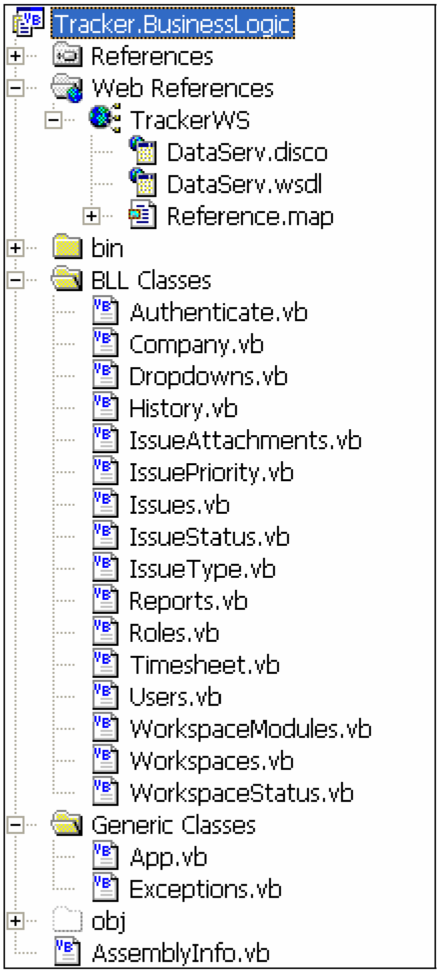
Figure 5.1: Tracker DAL Placement.

Business Logic in the Tracker Business Logic Layer

You might be thinking about all of the arduous code written in the BLL that manipulates and works on the data coming from the DAL. In the case of Tracker, there is not really any business logic at all. The only logic contained in the BLL is whether the data source is the XML Web Service or the SQL Server sitting on the LAN. Each class in the BLL determines what method to call on the appropriate data source, and either sends or receives business entity objects to and from the data source. So the classes from the Tracker.Info assembly are used by the Tracker.BusinessLogic assembly to transfer data to and from the client.

Even though there is no real hard core business logic in the Tracker BLL, it is not just a gratuitous layer added to the application to be cool. The simple fact that my client does not need to be aware of the data source is reason enough to separate out that piece from the Windows Forms code.

In the Tracker.BusinessLogic project, there is a class for each business entity object defined in the Tracker.Info class. These classes work in a similar fashion to the classes in the DAL, in that they are expecting the types in the Tracker.Info assembly. Figure 5.2 is a screen shot of the Tracker.BusinessLogic project, to give you an idea of how it is put together.



```
Tracker.BusinessLogic
+ References
- Web References
    - TrackerWS
          DataServ.disco
          DataServ.wsdl
      + Reference.map
+ bin
- BLL Classes
      Authenticate.vb
      Company.vb
      Dropdowns.vb
      History.vb
      IssueAttachments.vb
      IssuePriority.vb
      Issues.vb
      IssueStatus.vb
      IssueType.vb
      Reports.vb
      Roles.vb
      Timesheet.vb
      Users.vb
      WorkspaceModules.vb
      Workspaces.vb
      WorkspaceStatus.vb
- Generic Classes
      App.vb
      Exceptions.vb
+ obj
  AssemblyInfo.vb
```

Figure 5.2: Tracker.BusinessLogic project

Within each of the classes, the structure is as follows:

- A *Get* method which gets all records from the data source
- An *overloaded Get* method which accepts an ID based on the type to get a specific record
- An *overloaded Get* method which accepts a type from an enumeration in the Info class which tells the DAL which type of data to return
- A *Save* method which accepts a type and sends it to the DAL
- A *Delete* method which accepts an ID and deletes a record from the database

The following code shows the methods of the *Company* class, minus the actual implementation code.

```
Namespace Tracker.BusinessLogic
    <Serializable()> _
    Public Class Company
        Inherits App

        Function GetCompany(ByVal companyID As Integer) As Tracker.Info.CompanyInfo()

        End Function

        Public Function GetCompany() As Tracker.Info.CompanyInfo()
        End Function

        Public Function Save(ByVal company As Tracker.Info.CompanyInfo)

        End Function

        Function DeleteCompany(ByVal companyID As Integer, _

        End Function
    End Class
End Namespace
```

You probably notice how this structure is very similar to the DAL. In most of the classes of the Tracker.BusinessLogic assembly, the structure of the class is identical to that of the DAL.

To understand how this all works, let’s look at some code from the *Company* class in the Tracker.BusinessLogic project, and understand how it communicates with the data sources.

Understanding how the BLL Sends and Receives Data

In the Tracker client application, there is a reference to the Tracker.BusinessLogic and the Tracker.Info assemblies. In each form of the application, when data is needed, or when data needs to be updated, there is code that instantiates a new instance of a type in the Tracker.BusinessLogic assembly. In every form, you will see code that looks like this:

```
Dim bl as New Tracker.BusinessLogic.something
```

The *.something* is one of the types in the Tracker.BusinessLogic assembly. For example, if you needed to fill a WinGrid with all of the customer’s data, you would write this code:

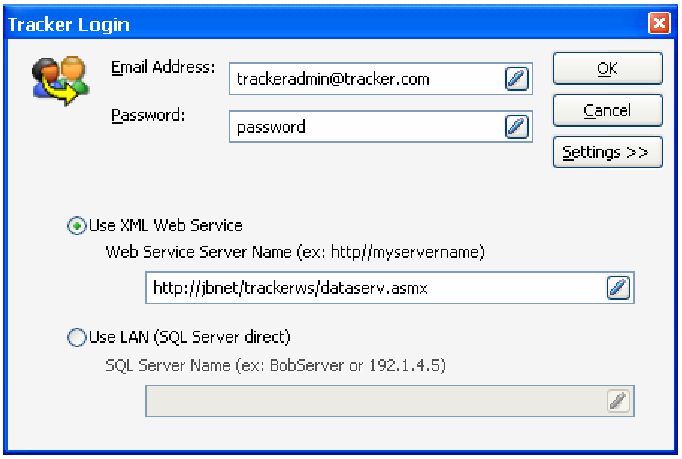
```
Dim bl As New Tracker.BusinessLogic.Company
UltraWinGrid1.DataSource = bl.GetCompany()
```

In the *Company* class, there is a method named *GetCompany* that returns an array of *CompanyInfo* objects to the caller. The actual code for *GetCompany* looks like this:

```
        Public Function GetCompany() As Tracker.Info.CompanyInfo()
            If GetConnectionString Then
                Select Case TypeOfConnection
                    Case "WS"
                        Dim proxy As TrackerWS.DataServ1Wse = GetProxy()
                        Try
                            Return proxy.GetCompanies()
                        Catch ex As Exception
                            Throw New ApplicationException(Exceptions.HandleError(ex))
                        End Try
                    Case "LAN"
                        Dim x As New Tracker.DataAccess.CompanyDB
                        Return x.GetCompany
                End Select
            End If
        End Function
```

If you notice, there is a variable named *TypeOfConnection*. Based on this value, one of two methods is called, the *GetCompanies* method from the XML Web Service, or the *GetCompany* method of the *CompanyDB* class. Based on the client configuration, the XML Web Service is used, or the DAL is accessed directly.

To determine what method of connection to use, the Tracker login screen prompts the user for the location of the data, as figure 5.3 demonstrates.



Tracker Login

Email Address: trackeradmin@tracker.com

Password: password

● Use XML Web Service

Web Service Server Name (ex: http//myservername)

http://jbnet/trackerws/dataserv.asmx

○ Use LAN (SQL Server direct)

SQL Server Name (ex: BobServer or 192.1.4.5)

Figure 5.3: The Tracker login screen

When the user logs in, this information is serialized to a file called AppSettings.Dat, which contains how this session of Tracker is going to get its data. This serialized file is stored in the application path, the same place as the private assembly for Tracker.BusinessLogic. When a method is called in the Tracker.BusinessLogic assembly, the file is read, the location of the data is determined, and the correct method is called to act upon the request.

The information for AppSettings.dat is based on the *UserSettings* class of the *AppSettings* project, which is listed here:

```
Imports System.Runtime.Serialization

' This class saves information about the user
' the it is serialized to disk, and used in place of the app.config file
' to store the user preferences
' LAN means the service is looking for a SQL Server connection, and uses certain security
' WS means the service is looking for a Web Service, and uses a certain security
Namespace Tracker

<Serializable()> _
Public Class AppSettings
        Public LastName As String
        Public FirstName As String
        Public EmailAddress As String
        Public TypeOfConnection As String
        Public LANLocation As String
        Public WSLocation As String
        'Public Password As String

        Sub New(ByVal FirstName As String, ByVal LastName As String, _
                ByVal EmailAddress As String, ByVal TypeOfConnection As String, _
                ByVal LANLocation As String, ByVal WSLocation As String)

            Me.FirstName = FirstName
            Me.LastName = LastName
            Me.EmailAddress = EmailAddress
            Me.TypeOfConnection = TypeOfConnection
            Me.LANLocation = LANLocation
            Me.WSLocation = WSLocation
        End Sub

    End Class
End Namespace
```

Each class in the Tracker.BusinessLogic project inherits a class named *App*. App class has a read only public property named *GetConnectionString*, which reads the serialized

AppSettings.Dat file, and returns to the caller the data that it needs to determine what data source to interact with. The following code is in the *App* class. Keep in mind; this is interacting with a serialized file on the file system which contains the data of *AppSettings* class that is listed above.

```
    ReadOnly Property GetConnectionString() As String
        Get
            Dim fileName As String = String.Format("{0}\{1}", Path.GetDirectoryName _
           (System.Reflection.Assembly.GetExecutingAssembly.Location), "AppConfig.dat")
            If File.Exists(fileName) Then
                Dim aL As ArrayList, p As Tracker.AppSettings
                aL = CType(LoadBinaryData(fileName), ArrayList)
                For Each p In aL
                    _firstName = p.FirstName
                    _lastName = p.LastName
                    _emailAddress = p.EmailAddress
                    _typeOfConnection = p.TypeOfConnection

                    ' To test out the WS-Security, change the following line to
                    ' _password = "BadPassword", or change the _emailAddress to something
                    ' other than what is stored in the binary formatter

                    ' _password = p.Password
                    _LANLocation = p.LANLocation
                    _WSLocation = p.WSLocation
                    ' Set the type of connection they are going to be using
                    If _typeOfConnection = "LAN" Then
                      _connectionString = "Data Source=" & _LANLocation & ";Initial
                      Catalog=Tracker;User ID=trackeradmin;Password=trackeradmin"
                    ElseIf _typeOfConnection = "WS" Then
                        _connectionString = p.WSLocation
                    Else
                        Return False
                    End If
                    Return True
                Next
            Else
                Throw New ApplicationException("AppConfig.Dat File Does Not Exist")
            End If
        End Get
    End Property

    Private Function LoadBinaryData(ByVal path As String) As Object
        Dim fs As FileStream = New FileStream(path, FileMode.Open)
        Dim bf As New BinaryFormatter
        LoadBinaryData = bf.Deserialize(fs)
        fs.Close()
    End Function
```

The other main task of the *App* class is to add the security token to the XML Web Service call. In the web service, which we will get to in the next section, WS-Security is used to verify that the application calling the web service is valid. It does this by authenticating the user against the Users table in the Tracker database. In each method of each class in the Tracker.BusinessLogic assembly, this code executes if the *TypeOfConnection* set in the AppSettings.Dat file is “WS”.

```
        Public Function GetCompany() As Tracker.Info.CompanyInfo()
            If GetConnectionString Then
                Select Case TypeOfConnection
                    Case "WS"
                        Dim proxy As TrackerWS.DataServ1Wse = GetProxy()
```

The *GetProxy* function, listed here

```
    Function GetProxy() As TrackerWS.DataServ1Wse
        Dim proxy As New TrackerWS.DataServ1Wse
        proxy.Url = ConnectionString
        Dim reqCtx As SoapContext = proxy.RequestSoapContext
        Dim tok As New UsernameToken(EmailAddress, Password, PasswordOption.SendHashed)
        reqCtx.Security.Tokens.Add(tok)
        reqCtx.Security.Elements.Add(New Signature(tok))
        Return proxy
    End Function
```

Will return a new instance of the *DataServ1Wse* class, which is the *DataServWS* web reference added to the Tracker.BusinessLogic project. The security token is then passed to the web service, and verified each time a call is made. Notice this line of code:

```
Dim tok As New UsernameToken(EmailAddress, Password, PasswordOption.SendHashed)
```

This sends the password hashed, which means it cannot be read by prying eyes when sent over the wire. Using WS-Security is pretty straightforward, and certainly makes your web service calls very secure.

Now that you have a good idea of how the client interacts with the BLL, and what the BLL actually does, let's take a look at the XML Web Service. Before that though, I want to cover a brief note on application settings.

A Brief Note on Application Settings

There are a couple of challenges when writing a tiered application.

- Where do I store my application settings?
- How does each layer know where the data source is?

There are, of course, many more challenges, but this is a vexing one. In an ASP.NET application, the Web.Config file stores application settings. In a stand-alone Windows Forms application, the App.Config stores application settings. But how does an assembly which is consumed by the applications know where to get it's settings from?

The answer is not simple, and like most of the solutions I like to offer up, it depends on the situation.

In the Tracker application, I am happy with the assemblies reading the AppSettings.dat file if they are on the client, and the Web.Config if they are on the server with the XML Web Service. But this may not be the best solution. If I wanted to make the Tracker.BusinessLogic assembly completely unaware of any application settings file, I could add this code to each of the classes:

```
Sub New(ConnectionString as String)

End Sub
```

Then each time the class is instantiated on the client, I would pass connection information to the newly created class. The code might look like this:

```
Dim bl as New Tracker.BusinessLogic.CompanyDB(cn)
```

I could also store the application settings in Isolated Storage on the client. This special storage area is created for each user on a computer, and is completely safe from any outside malicious access.

A third option would be to use a common registry key for the application, and always use that as the location for things such as database settings. But what happens if the database changes? Or if the registry location changes?

I don't think that matters, since any way you slice it, you need to worry about that problem with each approach. You might say "Just save the settings in SQL Server". Ok, but how do we know where to connect to get those settings?

A fourth option, and not a terribly bad one, would be to use a public XML Web Service that sends an encrypted token back to a caller with information about the connection location. The XML Web Service could possibly change, but at least there are no configuration settings on the client that need to be messed with.

In all of the so called solutions I have offered, there is probably one or three problems with each of them. When you are designing you applications, keep these ideas in mind, one of them may work out for you, but also keep in mind, it all depends on how much control you have over the consumers of the application, and how quickly or easily you might need to change application settings on the clients.

Integrating the XML Web Service

Note: The following information is essential to your success in working with the TrackerWS Xml Web Service. You will need to use it if you ever select Update Web Reference from the Tracker.BusinessLogic project, or if you change anything in the web service and need to update the WSDL. Before you attempt anything, make sure you have the following installed on your computer:

Microsoft Web Services Enhancements (WSE) 1.0 SP1 found at
http://www.microsoft.com/downloads/details.aspx?familyid=06255a94-2635-4d29-a90c-28b282993a41&displaylang=en

WSE Settings Tool for Visual Studio .NET found at
http://www.microsoft.com/downloads/details.aspx?FamilyId=E1924D29-E82D-4D9A-A945-3F074CE63C8B&displaylang=en

A key feature of the Tracker application is the XML Web Service using WS-Security. Like any other web service created using Visual Studio .NET, the TrackerWS project is an XML Web Service project, with a class that inherits *System.Web.Services.Service*. In this class, there are public functions marked with the *WebMethod* attribute, allowing those methods to be callable via the web service. In the TrackerWS project, all of the public functions are located in the *DataServ.asmx.vb* class. The functions in this class are mapped similar to the functions in the DAL and BLL, there is a method for each type of *Select*, *Insert*, *Update* or *Delete* that maps to the database.

To give you an idea of what the web service looks like, the code for the methods that interact with the *CompanyDB* class is listed below.

```
#Region "Company Class Service Info"

    <WebMethod()> _
    Public Function GetCompanies() As Tracker.Info.CompanyInfo()

        If Authenticater() Then
            Dim x As New Tracker.DataAccess.CompanyDB
            Return x.GetCompany
        End If

    End Function

    <WebMethod()> _
    Public Function GetCompany(ByVal companyID As Integer) _
            As Tracker.Info.CompanyInfo()
        If Authenticater() Then
            Dim x As New Tracker.DataAccess.CompanyDB
            Return x.GetCompany(Integer.Parse(companyID))
        End If

    End Function

    <WebMethod()> _
    Public Function SaveCompany(ByVal company As Tracker.Info.CompanyInfo)

        If Authenticater() Then
            Dim x As New Tracker.DataAccess.CompanyDB
            x.Save(company)
        End If

    End Function

    <WebMethod()> _
    Public Function DeleteCompany(ByVal companyID As Integer, _
            ByVal changeStamp As String)

        If Authenticater() Then
            Dim x As New Tracker.DataAccess.CompanyDB
            Return x.Delete(Integer.Parse(companyID), changeStamp)
        End If

    End Function

#End Region
```

You will notice that these methods are pretty straightforward. They simply call the corresponding method in the DAL to interact with the data. The unique aspect of each of these methods is this code wrapped around each function call:

```
        If Authenticater() Then

        End If
```

If you recall earlier, the BLL appends a token to the call to the web service which includes the information about the caller, specifically the username and password. In the *DataServ* web service, there is a method named *Authenticater* that reads that information, which is part of the SOAP header of the current *SoapContext*, and validates it against SQL Server.

The *Authenticater* method looks like this:

```
    Private Function Authenticater() As Boolean
        Dim requestContext As SoapContext _
                = HttpSoapContext.RequestContext

        Dim userToken As UsernameToken
        Dim returnValue As String

        If requestContext Is Nothing Then
            Throw New SoapException("Non-SOAP Message - Are you a hacker?", _
                            SoapException.ClientFaultCode)
        End If

        For Each userToken In requestContext.Security.Tokens
            If TypeOf userToken Is UsernameToken Then
                If userToken.PasswordOption = _
                            PasswordOption.SendHashed Then
```

```
                    Return True
                    Exit For
                Else
                    Throw New SoapException _
                    ("Password must be hashed", _
                        SoapException.ClientFaultCode)
                End If
            End If
        Next

    End Function
```

Authenticater reads the tokens, and then verifies them against the *GetPassword* method in the *PasswordProvider* class, which is also part of the web service. All of this code is transparent to you once you set up the Web.Config file with the correct assembly name and method to use to authenticate the security tokens passed in the current context.

```
Imports Microsoft.Web.Services
Imports Microsoft.Web.Services.Security
Imports System.Data.SqlClient
Imports System.Configuration

Public Class PasswordProvider
    Implements IPasswordProvider

    Public Function GetPassword( _
        ByVal usernameToken As UsernameToken) As String _
        Implements IPasswordProvider.GetPassword

        Dim d As SqlDataReader = SqlHelper.ExecuteReader _
                    (ConfigurationSettings.AppSettings("cn"), _
                "Users_Sel_ByEmail", usernameToken.Username)
        While d.Read
            Return IIf(IsDBNull(d("Password")), "", d("Password"))
        End While

    End Function
End Class
```

Integrating WS-Security into the application was very straightforward. I did use an excellent book, which I must recommend to everyone interested in using the Web Services Enhancements from Microsoft. On Amazon.com, look up:

Web Services Enhancements by Bill Evjen, published by Wiley

This is an outstanding complete reference on everything about the ins and outs of using Web Services Enhancements.

The next step in making the web service callable from the BLL is to simply add it to the application as a Web Reference. Sounds simple, right? Well, no, which leads us to …

The Not So Simple Marshalling of Types via XML Web Services

When you add a Web Reference using Visual Studio .NET, information is read from the web service, and a proxy is created in your application. This proxy is then used by you to consume the methods in the web service with early binding. This all works great, if you are using strings, integers or Datasets. This does not work great, however, if you expect the BLL to understand the type of a custom business entity object.

This code, from the web service, returns a type of *CompanyInfo* to the caller:

```
    <WebMethod()> _
    Public Function GetCompanies() As Tracker.Info.CompanyInfo()
```

The problem with XML Serialization and .NET is that there is no way the caller can actually use a type of *Tracker.Info.CompanyInfo* from the default proxy created by Visual Studio .NET. The code generated by Visual Studio .NET needs to be modified so you can directly consume any type of complex object. If the proxy class is not modified, you will end up getting an error that looks something like this:

```
C:\Shared\Infrasgistics Projects\Tracker.BusinessLogic\BLL
Classes\Authenticate.vb(66): Value of type '1-dimensional array of
TrackerWS.UsersInfo' cannot be converted to '1-dimensional array of
Tracker.Info.UsersInfo' because 'TrackerWS.UsersInfo' is not derived from
'Tracker.Info.UsersInfo'.
```

To understand how this works, you will need to make sure that you are viewing the hidden files in the Solution Explorer, so you can see the auto generated *Reference.vb* class file created by Visual Studio .NET, as figure 5.4 demonstrates.

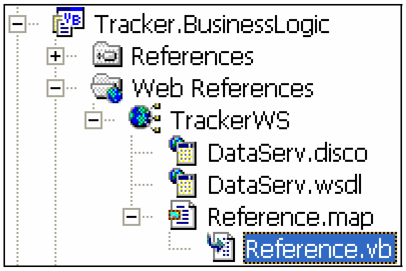


Figure 5.4: The Reference.vb file generated by Visual Studio .NET

If you open this file up in the code editor, you will see a class for *DataServ1Wse*, which starts like this:

```
    Public Class DataServ1Wse
        Inherits Microsoft.Web.Services.WebServicesClientProtocol
```

If you look at the *DataServ.asmx.vb* file in the *TrackerWS* project, you will notice the class is actually named *DataServ1.vb*. The *DataServ1Wse* proxy class is created automatically by the WSE Settings Tool for Visual Studio .NET that you installed earlier. In the *Reference.vb* class, there is actually a *DataServ1* public class, and a *DataServ1Wse* public class. This is done so an application can call methods from both the WSE enhanced methods and the non-WSE methods in the Xml Web Service. The class declaration for the *DataServ1* class looks like this:

```
   Public Class DataServ1
     Inherits System.Web.Services.Protocols.SoapHttpClientProtocol
```

Notice that class used by WSE class inherits:

```
Microsoft.Web.Services.WebServicesClientProtocol
```

And the non-WSE class inherits

```
System.Web.Services.Protocols.SoapHttpClientProtocol
```

This is a very nice feature of the WSE Settings Tool. Otherwise, you would need to manually change the class to the correct Soap protocol. You do need to however, at this point, make one very important change:

Modify this line of code:

```
Public Class DataServ1
  Inherits System.Web.Services.Protocols.SoapHttpClientProtocol
```

To look like this:

```
Public Class DataServ1XX
    Inherits System.Web.Services.Protocols.SoapHttpClientProtocol
```

The reason for this is a naming collision in the added web reference. You can actually name it whatever you want, just keep in mind that you need to reference what you named it if you add non-WSE enhanced methods to the *DataServ.asmx.vb* file.

Next, in the same *Reference.vb* file, scroll until you find the *End Class* for *DataServ1XX (or whatever you just renamed it to)*, you will see something like this:

```
Public Class IssuesInfo

    '<remarks/>
    Public IssueID As Integer

    '<remarks/>
    Public WorkspaceID As Integer

    '<remarks/>
    Public TypeID As Integer

    '<remarks/>
    Public StatusID As Integer

    '<remarks/>
    Public PriorityID As Integer
```

You will see similar classes like this for each of the types in the Tracker.Info assembly. This is because the WSDL generated by Visual Studio .NET for the web service is looking at it for type serialization information. Unfortunately, this information is not of any use to anyone. In fact, it will cause you weeks of problems trying to figure out why you cannot send and receive custom business entity objects via web services.

To get around the problem, you need to do 2 things:

1. Delete all of the classes created in the *Reference.vb* file which map to the custom business entity objects. This means delete everything after the *End Class* of *DataServ1xx*, until you hit the *End Namespace* for the file.
2. Add the *Imports Tracker.Info* to the top of the class file. This will make the *DataServ1WSE* and *DataServ1XX* classes look at the Tracker.Info class for their type definitions.

This sounds crazy, and it is, but it is the only way to make it work. You are probably thinking, what if I change any methods in the *DataServ1.asmx.vb* class in the TrackerWS project? Yes, you will need to modify this *Reference.vb* class each time you make any changes. While developing the Tracker application, I did this a lot in the first couple days, but then it was weeks before I needed to do it again before final deployment. This was possible because of the architecture, when I needed to make changes; they were normally in the DAL or BLL. The classes in the Tracker.Info, and the methods for the web service were all defined up front, so little or no change was necessary to the web service.

For more information on types and XML Web Services, look up Q326790 at http://support.microsoft.com. You can also do a search on http://www.dotnet247.com for "web service type marshal" (without the quotes), and you will find a vast array of good posts on this topic.

Some Notes on Security

You may have noticed that there is no security setup between the BLL and the DAL if you are using the LAN to access the SQL Server data. I did this intentionally. You can implement your favorite role based security against Active Directory, or whatever mechanism you want to implement. By using only a SQL Server based user name and password to determine authentication to the SQL Server, you are leaving yourself open a bit. But since this is on your secure private LAN, that may not matter too much. You can also modify the App.Config file to use Integrated Security for SQL Server, instead of the TrackerAdmin user. This is your choice. The security showcase for this version of Tracker is the implementation of WS-Security.

Summary for Today and What is Next

In this chapter, you learned about the business logic layer, or BLL, and how it is used in the Tracker application. You also learned about the WS-Security implementation in Tracker, and how the XML Web Service is used in the BLL project. This chapter and the previous two chapters covered the fundamental building blocks of the Tracker application. The next chapter will look at the Application Blocks, and how they are used throughout the projects in the Infragistics.Tracker solution.

Chapter

6

Chapter 6: Integrating the Application Blocks

When developing any type of application, there are always certain functions that need to be done. Most of the time, you just write the code, and include these functions as part of the application. Microsoft realized this, and wrote some of the more complex code for us in the Application Blocks for .NET. The application blocks consist of pre-built assemblies that can be dropped into your applications to provided services such as data access, exception handling and auto-updating, just to name a few. The Tracker application uses these application blocks, and you probably will too after you realize how easy they are to implement and how much time it will save you. In this chapter, you are going to learn about the three application blocks used in Tracker, and how they are implemented. These application blocks are:

- Data Access Application Block
- Exception Management Application Block
- Updater Application Block

The application blocks that are not used in Tracker, but can be downloaded from the MSDN site at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/PAIBlock.asp are:

- Aggregation Application Block
- Asynchronous Invocation Application Block
- Caching Application Block
- Configuration Management Application Block
- User Interface Process Application Block

The Configuration Management Application Block would be very useful in Tracker, and it almost made it in to the application. This application block handles the reading and writing of configuration files in various ways. It is extremely useful, and this is on the wish list for the next version of Tracker. The Asynchronous application block almost made it in to Tracker also. This application block gives you the ability to use a standard way of asynchronously invoking methods. If Tracker were an ASP.NET application, it would use the Caching Application Block. The User Interface Process Application Block and the Aggregation Application Block did not fit the purpose of Tracker. Though I think the User Interface Application Block is very powerful, it would have made Tracker way too complex.

Implementing the Data Access Application Block

Earlier in chapter 4, you read about the DAL, and how it communicated with the SQL Server database. In each method that interacted with the database, there was a call to the *SqlHelper* or *SqlHelperParameterCache* class, which looked like this:

```
SqlHelperParameterCache.CacheParameterSet _
    (cnString, "Company_Upd", Params)
```

```
SqlHelper.ExecuteNonQuery(cnString, _
    CommandType.StoredProcedure, "Company_Upd", Params)
```

The SqlHelper class, which is the source code for the Data Access Application Block, contains the tested and optimized data access code that interacts with SQL Server or MSDE. Using the application block can be considered a best practice; the block handles any type of SQL statements, and handles all of the resource management with SQL Server, such as connection management. Using the application block, you can:

- Call stored procedures or SQL text commands.
- Specify parameter details.
- Return *SqlDataReader*, *DataSet*, or *XmlReader* objects.
- Use strongly typed *DataSets*.

To implement the application block, you do not need to install anything. The SqlHelper class file is part of the Tracker.DataAccess and TrackerWS projects. If you want to check out the samples, or get the assembly for the application block so you do not need to add the SqlHelper class to you applications, you can find the Data Access Application Block at this URL:

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/daab-rm.asp

To use the code from the SqlHelper class, you only need to call the static methods on the class. The class is marked as *NotInheritable*, and the methods in the class that you can access are marked as Shared. This way, you do not need to use the *New* keyword every time you need to access the SqlHelper methods.

In each of the classes in the Tracker.DataAccess project, the System.Data.SqlClient namespace is imported, as well as the Tracker.Info namespace. The following code demonstrates the *CompanyDB* class, each class in the Tracker.DataAccess project will look the same.

```
Imports System.Data.SqlClient
Imports Tracker.Info

Namespace Tracker.DataAccess
    <Serializable()> _
    Public Class CompanyDB
```

Once the SqlHelper source is added to a project, and the SqlClient namespace is referenced in a class, you can use the SqlHelper class to communicate with SQL Server.

In the Tracker application, the only methods from the SqlHelper class that are used are the *ExecuteReader* and the *ExecuteScaler* methods. These methods work just like their SqlClient counterparts, The *SqlClient.ExecuteReader* method, for example, will return a *SqlDataReader* to the caller. Since the DAL is either filling the custom types with data, or executing stored procedures that do not return anything, those are the only two methods that were needed.

The main reason to use the application block is the simplicity of the call. For example, to retrieve all of records in the Company table in the Tracker database, you would write this code using the SqlHelper:

```
Dim d As SqlDataReader = SqlHelper.ExecuteReader(ConnectionSettings.cnString, _
                    CommandType.Text, "Select * from Company")

While d.read
   Console.WriteLine(d("CompanyName"))
End While
```

Notice that the SqlHelper.ExecuteReader method takes the connection string, the type of command object, and the actual command itself. This method is heavily overloaded, there are 9 ways you can execute SqlHelper.ExecuteReader, the example I have provided is the simplest.

If you were not to use the SqlHelper class, your code would need to manage the connection, and would probably look something like this:

```
Dim cn As SqlConnection
cn = New SqlConnection(ConnectionSettings.cnString)
cn.Open()

Dim cmd As SqlCommand
cmd = New SqlCommand("Select * from Company", cn)

Dim dr As SqlDataReader

Try
   cn.Open()
   dr = cmd.ExecuteReader(CommandBehavior.CloseConnection)

   While dr.read
      Console.WriteLine(d("CompanyName"))
   End While

Catch ex As Exception

Finally
   If Not cn Is Nothing Then
      cn.Dispose()
   End If
End Try
```

If you notice, there is 1 line of code using SqlHelper, and over 10 lines of code using standard ADO.NET code. Even though that statement is a slight exaggeration, I still think it is a no-brainer to use the Data Access Application Block.

***Note**: I know many people that have written their own custom SqlHelper like classes. The recommendation I would have if you have already done this is to try out the application block from Microsoft, the benefit of their code optimization and their thorough testing makes it at least worth a look.*

In Tracker.DataAccess, all of the interaction with SQL Server is done via stored procedures. The code that actually gets the Company data into the CompanyInfo type looks like this:

```
Dim d As SqlDataReader = SqlHelper.ExecuteReader(ConnectionSettings.cnString, _
       CommandType.StoredProcedure, "Company_Sel")

Dim c As New ArrayList
   While d.Read
   Dim al As CompanyInfo = New CompanyInfo
   al.CompanyID = IIf(IsDBNull(d("CompanyID")), "", d("CompanyID"))
```

You do not have to worry about managing the connection, which I think is the most important piece of the code when dealing with the database. The connection to the database is always the most expensive resource, and sometimes the garbage collector will not collect it fast enough if you do not call Dispose. Using the Data Access Application Block, these worries are gone.

To handle stored procedures that expect parameters, you can create a parameter collection, and then pass that collection to the ExecuteScaler method of the application block. In the following code, there are 10 parameters added to the *params()* object, which is a *SqlParameter* type.

```
Dim params() As SqlParameter = New SqlParameter(9) {}
params(0) = New SqlParameter("@CompanyName", company.CompanyName)
params(1) = New SqlParameter("@Address", company.Address)
params(2) = New SqlParameter("@City", company.City)
params(3) = New SqlParameter("@State", company.State)
params(4) = New SqlParameter("@ZipCode", company.ZipCode)
params(5) = New SqlParameter("@BillToAddress", company.BillToAddress)
params(6) = New SqlParameter("@BillToCity", company.BillToCity)
params(7) = New SqlParameter("@BillToState", company.BillToState)
params(8) = New SqlParameter("@BillToZipCode", company.BillToZipCode)
params(9) = New SqlParameter("@AddedBy", company.AddedBy)
SqlHelperParameterCache.CacheParameterSet(cnString, "Company_Ins", params)
SqlHelper.ExecuteNonQuery(cnString, _
         CommandType.StoredProcedure, "Company_Ins", params)
```

This is standard ADO.NET code for creating a parameter collection. The only difference is what gets called to execute the stored procedure. By calling this line of code:

```
SqlHelperParameterCache.CacheParameterSet(cnString, "Company_Ins", params)
```

The SqlHelper class will cache the parameters, and then use those cached parameters when this code is called:

```
SqlHelper.ExecuteNonQuery(cnString, _
         CommandType.StoredProcedure, "Company_Ins", params)
```

This optimizes the execution of the code, and will allow the cached parameters to be used for subsequent consecutive calls to the same stored procedure.

The nice thing about the Data Access Application Block is that you need not configure anything. Simply pass the connection string parameter, along with the text of the command you are executing and any parameters that might go with it, and you can use it immediately.

Implementing the Updater Application Block

The Updater application block will handle automatic updates of your application by checking configuration files via a predefined network location. Using a pull model for checking the versions of your application or its components, the Updater Application Block will seamlessly check for new versions of your application, and update the client automatically. The main features of the updater block are:

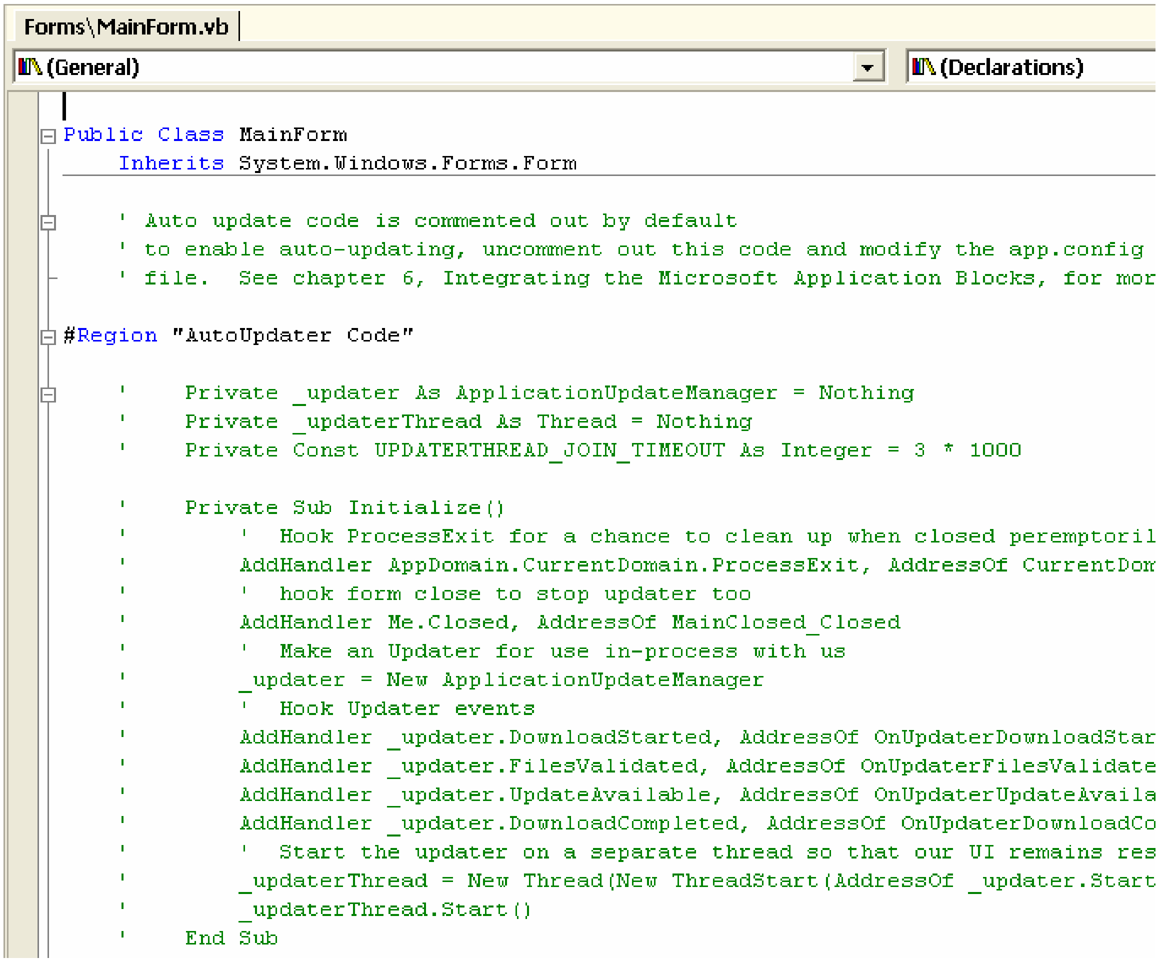
- Implement a "pull" based update solution for .NET applications.
- Use cryptographic validation techniques to verify the authenticity of application updates before applying them.
- Perform post-deployment configuration tasks without user intervention.
- Write applications that automatically update themselves to the latest available version

The Tracker Application is configured to use the updater component. The actual implementation code is commented out in the *MainForm.vb* class. I did this because if the application block is not configured properly, you will spend a lot of time figuring out why Tracker is crashing all the time. To make it work is very simple, it is just a matter of un-commenting some code and updating some paths in the App.Config file.

Before you start dealing with implementing this in Tracker, I suggest you download the Updater Application Block from this URL, and read the documentation.

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/updater.asp

In figure 6.1, you can see where the code for the application updater has been added.



```
Public Class MainForm
    Inherits System.Windows.Forms.Form

    ' Auto update code is commented out by default
    ' to enable auto-updating, uncomment out this code and modify the app.config
    ' file.  See chapter 6, Integrating the Microsoft Application Blocks, for mor

#Region "AutoUpdater Code"

    '    Private _updater As ApplicationUpdateManager = Nothing
    '    Private _updaterThread As Thread = Nothing
    '    Private Const UPDATERTHREAD_JOIN_TIMEOUT As Integer = 3 * 1000

    '    Private Sub Initialize()
    '        '  Hook ProcessExit for a chance to clean up when closed peremptoril
    '        AddHandler AppDomain.CurrentDomain.ProcessExit, AddressOf CurrentDom
    '        '  hook form close to stop updater too
    '        AddHandler Me.Closed, AddressOf MainClosed_Closed
    '        '  Make an Updater for use in-process with us
    '        _updater = New ApplicationUpdateManager
    '        '  Hook Updater events
    '        AddHandler _updater.DownloadStarted, AddressOf OnUpdaterDownloadStar
    '        AddHandler _updater.FilesValidated, AddressOf OnUpdaterFilesValidate
    '        AddHandler _updater.UpdateAvailable, AddressOf OnUpdaterUpdateAvaila
    '        AddHandler _updater.DownloadCompleted, AddressOf OnUpdaterDownloadCo
    '        '  Start the updater on a separate thread so that our UI remains res
    '        _updaterThread = New Thread(New ThreadStart(AddressOf _updater.Start
    '        _updaterThread.Start()
    '    End Sub
```

Figure 6.1: Commented out Application Updater code in Tracker.

Making the application auto-update is not as easy as just commenting out the code. You need to modify how the application gets executed, and you will need to modify the App.Config file with the appropriate path information of where the assemblies are located that need to get updated, and where to look for the updates.

In a nutshell, here is the process (this is borrowed directly from the help documentation):

- On the server, the manifest utility supplied with the Updater Application Block is used to generate a manifest file for each application update. The manifest lists all the files included in the update, a hashed signature for each file (including the manifest itself), and optionally a post-processor to be executed on the client after the files are downloaded and validated.

- A controller application is used to start and stop the core application updater, which is implemented in the Updater Application Block. Controllers are applications that start / stop the application updater and respond to its events. You can use one of the controllers provided as QuickStart applications or develop your own. The application configuration file associated with the controller is used to determine three fundamental configuration settings for the application update process:

  - The applications to be updated, including the location of the client configuration file and server manifest file used to determine the latest version of the application on the client and on the server.

  - The downloader component to use when copying files. Downloader components must implement the IDownloader interface defined in the Updater Application Block. The Updater Application Block includes a downloader that uses the Background Intelligent Transfer Service (BITS) to copy files. Additionally, you can develop your own custom downloaders.

  - The validator component that should be used to validate the downloaded files. Validator components must implement the IValidator interface defined in the Updater Application Block. The Updater Application Block includes two validators, a symmetric key-based validator and an RSA public/private key-based validator. Additionally, you can develop your own custom validators.

- The application updater periodically initiates the update process. When this happens, the application updater uses the specified downloader to copy the manifest file for each application specified in the application configuration file to the client. If updates are available, the downloader copies the updated files to a temporary directory on the client.

- The application updater loads the specified validator and validates the downloaded files. If the files are valid, they are copied to the appropriate application folder and the configuration file for the application launcher is updated to reflect the new version.

Understanding that process, you will need to modify the App.Config file with appropriate paths. The following example is how a portion of the App.Config looks for the Tracker application.

```
<application name="Tracker" useValidation="false">
   <client>
      <baseDir>c:\Tracker</baseDir>
      <xmlFile>c:\Tracker\AppStart.exe.config</xmlFile>
      <tempDir>c:\Tracker\TempFiles</temp
   </client>

   <server>
       <xmlFile>http://localhost/trackerupdateshare/ServerManifest.xml</xmlFile>
       <xmlFileDest>c:\trackerupdateshare\UpdateServer\ServerManifest.xml
          </xmlFileDest>
       <maxWaitXmlFile>60000</maxWaitXmlFile>
   </server>
</application>
```

The Controller application to be used would be the AppStart.exe that ships with the Quickstart samples for the Updater Application Block. The AppStart.exe is how Tracker would actually execute, you would not execute the Tracker.exe application directly. This controller application uses the information from the manifest files to determine what application to execute, and where to look for updates.

An example of how this might work would be the following:

1. Create a directory on your server, or on a share on your machine (which is what I did), and name it TrackerUpdateShare.
2. Add a sub-folder for the current version of the application, which is 1.0.0.0

3. Add the AppStart.exe to a folder named 1.0.0.0 in the Tracker project.
4. Modify the App.Config file to look at the correct network locations for the server and the client.

When you execute the AppStart.exe application, it will read the manifest on the local machine, and then execute the Tracker application. Inside Tracker, the Application Updater code will execute in the background, and download a new version of the application, if one exists, from the server. You would add sub-folders named with the actual application version you wanted to update, such as 2.0.0.0 or 1.0.5.1. As long as the manifest file on the client is looking in the correct place, you application will update auto-magically.

The way I have just described the implementation is in its simplest form. You can add various security checks, and use the manifest utility on the server side to hash public and private keys that are verified before an application updates. The QuickStart examples in the Updater Application block are very useful, and cover a variety of different ways you can update your clients.

Implementing the Exception Management Block

The Exception Management Application Block provides a highly customizable way for you to centrally deal with exceptions that might occur in your applications. Using information stored in the App.Config file, the exception block uses publishers to determine how an exception should be handled. This publisher could be an Email message, an entry to the event log, and a custom event that you write yourself.

Before getting too much further, you should download and install the Exception Management application block from this URL:

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/updater.asp

To publish exceptions in Tracker, there is a public shared method in the Exceptions class named *HandleError*, which takes a type of *Exception* as a parameter. In this method, the *Publish* method of the *ExceptionManager* class is executed, with the exception parameter that is passed to the method. The following code demonstrates the *Exceptions* class, and the *HandleError* method.

```
Imports Microsoft.ApplicationBlocks.ExceptionManagement

Public Class Exceptions

    Shared Function HandleError(ByVal ex As Exception) As String
        ExceptionManager.Publish(ex)
        Return ex.ToString
    End Function

End Class
```

The *Publish* method uses the default publisher defined in the App.Config file. For Tracker, the event log is used to log all errors. You can see how the default publisher is defined in the App.Config file in the following code. (This line of code must be on a single line in the App.Config file)

```
<exceptionManagement mode="on">

<publisher
mode="on"
assembly="Microsoft.ApplicationBlocks.ExceptionManagement"
type="Microsoft.ApplicationBlocks.ExceptionManagement.DefaultPublisher"
logName="Application"
```

```
applicationName="Infragistics.Tracker" />

</exceptionManagement>
```

Notice the keys that are used:

- logName
- applicationName

By specifying the values of where you want the default publisher to handle the exceptions message (not the exception, just the message in the exception); you have a centralized place to track what exceptions have occurred in the application. This is the same configuration used on the server side, in the Web.Config file for the TrackerWS application.

In the Tracker code, in the *Try … Catch* blocks; there is a call to the *ShowError* method, which in turn called the HandleError method. The reason I have 2 methods is to minimize the amount of code I needed to write for the prompts back to the user. The *ShowError* method actually displays the *MessageBox*, while the HandleError method publishes the exception. In *HandleError*, a string of the message is returned back to the caller. The following code demonstrates how this is implemented for each *Try … Catch* block in Tracker.

```
Try
 ‘ code that could cause an exception
Catch ex As Exception
   ShowError(ex)
End Try
```

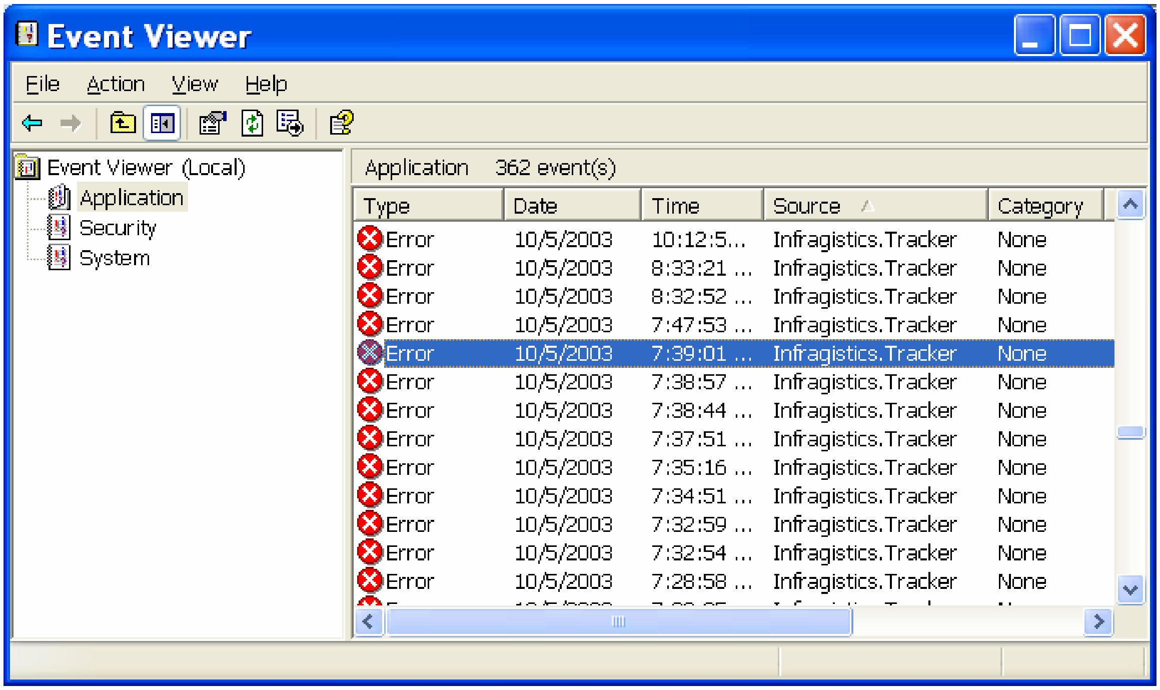
The *ShowError* method looks like this:

```
Public Function ShowError(ByVal ex As Exception)

    ' this code calls the HandleError method, which
    ' published this exception the the Event log
   MessageBox.Show(Exceptions.HandleError(ex), _
                App.MessageCaption, _
                MessageBoxButtons.OK, _
                MessageBoxIcon.Information)
End Function
```

Which in turn calls the *HandleError* method to actually publish the exception:

```
Shared Function HandleError(ByVal ex As Exception) As String
    ExceptionManager.Publish(ex)
    Return ex.ToString
End Function
```

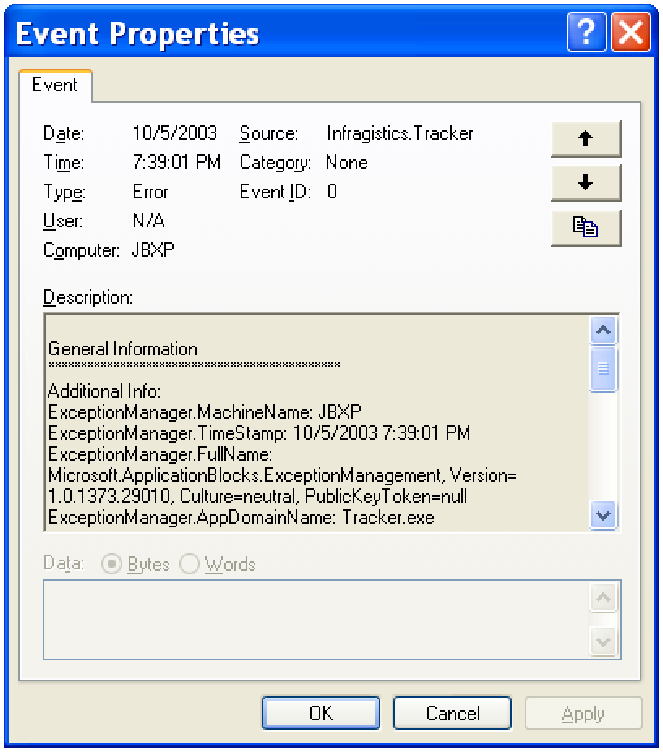
When an exception occurs, it is logged to the event log. Figure 6.2 will give you an idea of what you can expect in the event log and figure 6.3 is one of the events that contain the exception information.



Application 362 event(s)

Type	Date	Time	Source	Category
Error	10/5/2003	10:12:5...	Infragistics.Tracker	None
Error	10/5/2003	8:33:21 ...	Infragistics.Tracker	None
Error	10/5/2003	8:32:52 ...	Infragistics.Tracker	None
Error	10/5/2003	7:47:53 ...	Infragistics.Tracker	None
Error	10/5/2003	7:39:01 ...	Infragistics.Tracker	None
Error	10/5/2003	7:38:57 ...	Infragistics.Tracker	None
Error	10/5/2003	7:38:44 ...	Infragistics.Tracker	None
Error	10/5/2003	7:37:51 ...	Infragistics.Tracker	None
Error	10/5/2003	7:35:16 ...	Infragistics.Tracker	None
Error	10/5/2003	7:34:51 ...	Infragistics.Tracker	None
Error	10/5/2003	7:32:59 ...	Infragistics.Tracker	None
Error	10/5/2003	7:32:54 ...	Infragistics.Tracker	None
Error	10/5/2003	7:28:58 ...	Infragistics.Tracker	None

Figure 6.2: Exceptions published to the Event Log



Date: 10/5/2003 Source: Infragistics.Tracker
Time: 7:39:01 PM Category: None
Type: Error Event ID: 0
User: N/A
Computer: JBXP

Description:

```
General Information
*********************************************
Additional Info:
ExceptionManager.MachineName: JBXP
ExceptionManager.TimeStamp: 10/5/2003 7:39:01 PM
ExceptionManager.FullName:
Microsoft.ApplicationBlocks.ExceptionManagement, Version=
1.0.1373.29010, Culture=neutral, PublicKeyToken=null
ExceptionManager.AppDomainName: Tracker.exe
```

Data: ● Bytes ○ Words

Figure 6.3: Exception message in the event log.

That is extremely powerful, and extremely useful. One thing to watch out for though; during testing, I was running into lots of errors due to reasons beyond my control (of course it had nothing to do with my code). Before I knew it, the application log was full, thus causing the Exception Management Block to actually have an exception. Make sure you deal with the event logs in a way that they do not fill up. You do not need to log every exception, maybe just the ones you really care about.

Summary for Today and What is Next

In this chapter you learned about the application blocks, and how they are used in the Tracker application. I think the application blocks are extremely powerful, and offer a level of functionality that would take quite a while for you to write yourself. The simplest application blocks to use are the Exception Management and the Data Access Application Blocks; you may want to start with those in your own applications. In the next chapter, we start talking about user interface design, and what thought process went into the Tracker application's user interface.

7

Chapter 7: Designing a User Interface

The next several chapters concentrate on the user interface of Tracker, and how it was designed and implemented. In this chapter, we will look at what makes a user interface acceptable, and why using the built in components that ship with Visual Studio .NET will not always be accepted by end users when you are designing applications.

***Note**: A word about Infragistics tools naming conventions: Infragistics refers to each control by tool type rather than explicit product name. You will notice internal assemblies for each control are explicitly referenced by assembly name in code; therefore they are referenced as such in the body of this booklet. However, for simplicity, all control assembly names will be shortened by dropping the "Ultra" prefix. For example, Infragistics refers to their Windows Forms grid as ".NET grid", the assembly name is "UltraWinGrid", this booklet will refer to it as "WinGrid".*

What is a Makes Good User Interface?

That is a good question. You can toil for weeks and months on highly sophisticated code, which handles all the important processing of your application. But if the user interface which interacts with that code is not "good", then it will all be for nothing. Put simply, a good user interface is one that users do not reject.

To create a good user interface, you need to understand what applications your target audience is using, and how they use these applications. If your target audience is used to applications that look like this:

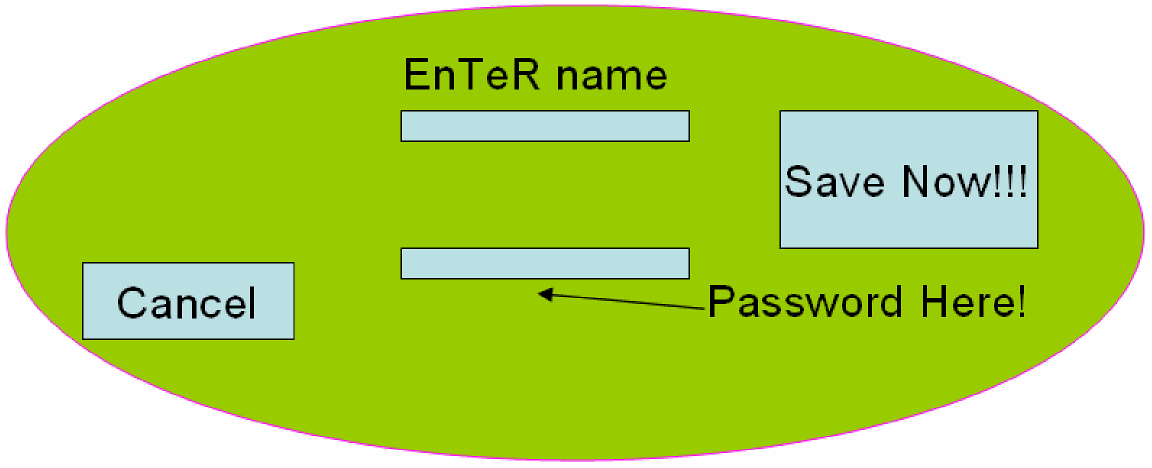


Figure 7.1: Custom user interface

Then they might not appreciate when you give them something that looks like this:

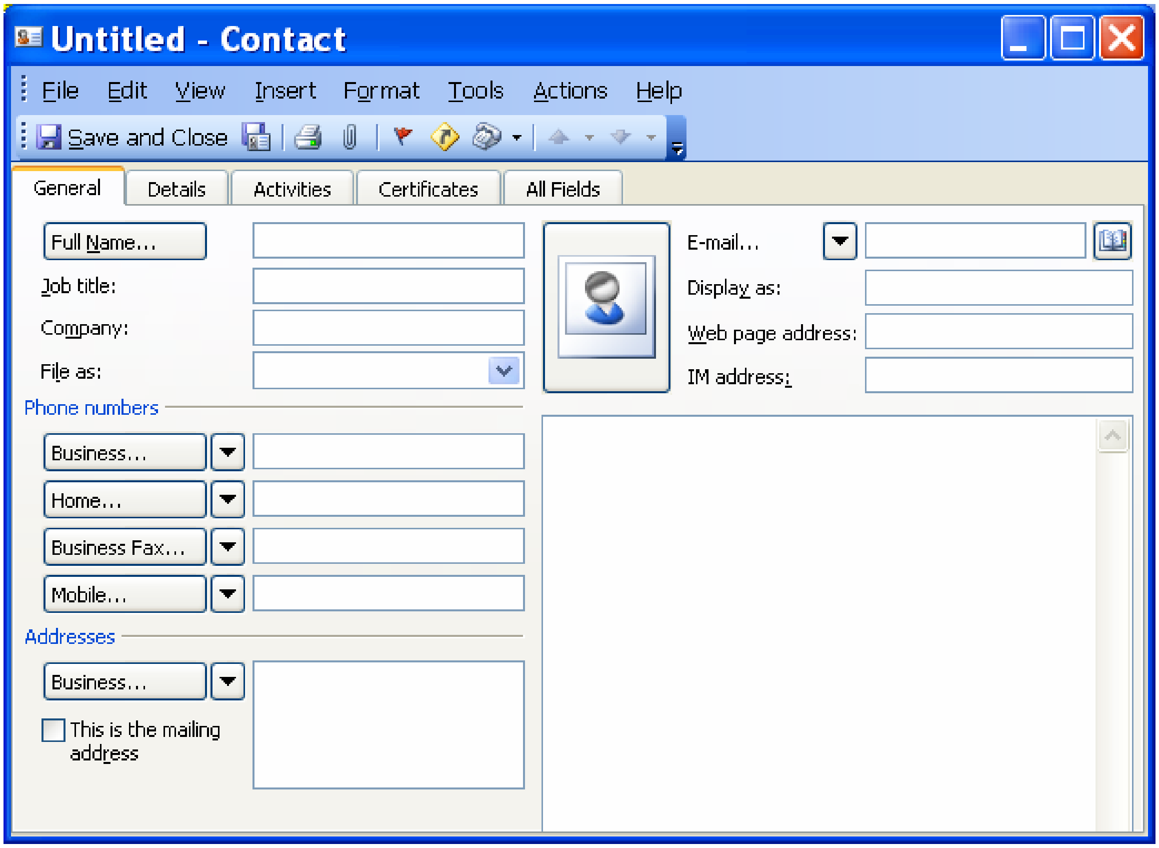


Figure 7.2: Outlook New Contact screen

The idea is that a user will accept an application if they are comfortable with how it works, and if it helps them get their job done quickly. You need to determine what the end user is used to using, and tailor your user interface designs around that model.

Quicken, the accounting package by Intuit, is the best selling personal accounting package in the world. Quicken defined how a user interface should work for keeping track of your check book on a computer. Microsoft Money, in my opinion, is a better product, but is not as widespread as Quicken. Is it because the user interface is so different? No. The data entry screens are very similar, and actually, Microsoft Money has a slicker user interface. The reason everyone uses Quicken is it was there first. If Microsoft Money was there first, the tables would be turned. This is an example of the better user interface not winning the acceptance battle. It is what the users are comfortable with, how fast they can click around without guessing, that is what makes Quicken sell more product than Money.

Note: *This is an unscientific comparison of Money vs. Quicken. Everyone I know uses Quicken, so I am just assuming it sells more, I have no hard numbers to back up my claim.*

This lesson can be used when you are designing the next user interface for a project. Do not go nuts and try to re-invent the way your users are going to work. Give them something they will feel comfortable with, that is not too drastic a change from the way they get their work done in their current applications. If you try to do too much at once, there will be a support and training nightmare on your hands.

Understanding that a user interface can only be judged successful by the acceptance of users, let's take a look at the most widely used application on the planet. Microsoft Outlook.

Microsoft Outlook took the world by storm somewhere in the mid 1990's. As a replacement to Schedule+, which was a client for Microsoft Exchange, Outlook did things that every user needs. It keeps your calendar, handles your contacts, allows you to track tasks, and best of all, is a robust email client. Microsoft knew that good features, and a good design, would facilitate the acceptance of Outlook by end users. They were right. As of today, Outlook is on its 11th version (actually 4th or 5th iteration, the version is 11 in Office 2003).

Why is Outlook so successful? The main reason: Nobody ever needs to read the help file. That's right, your Grandmother is using Outlook, and she has never opened the help file. The user interface is such that you do not need to think about "how" you are going to do something. Almost every new application on the market today tries to mimic the look and feel of Outlook. If an application can work like Outlook does, then software companies know they will not need to show people how to use it, the application would be friendly enough that the user *wants* to click around and figure it out, but does not necessarily have to.

Quick Question: *Name the order of the main menu tools in all Microsoft applications.*

You probably said "File, Edit, View, Tools, Help" faster then you could open your mouth.

What does CTRL+F do in Visual Studio .NET?

Opens the Find dialog.

What does it do in Word, Excel, Outlook, Access, Project, PowerPoint, etc .. etc … etc …

Opens the Find dialog.

I think you see where this is going. Consistency is a good thing. Your personal preference may not be what CTRL+F does, but everyone (or almost everyone) uses Microsoft applications, so they expect CTRL+F to open the Find dialog.

Should you mimic Outlook in your next interface design? Not if your company uses Eudora Pro for Linux. I think you see the point; you need to create your applications so they will be more accepted by the end user.

Choosing the Right Tools

Now that you know what a good user interface should look like, you need the right tools for the job. Even though Visual Studio .NET has lots of cool new controls, most of them do not affect the presentation layer. There is the HelpProvider, the various dialog controls, and many other non-visual controls, but if you go through the list, it is not much better then Visual Basic 6 was. In fact, they got rid of the DataRepeater control, a favorite of mine! To create a truly cool user interface (without re-inventing the wheel or writing your own controls), you would turn to something like NetAdvantage 2004 Volume 1 from Infragistics. Why would you spend your money on controls that you might be able to write yourself? First, you would not be able to write the entire array of user interface elements included with NetAdvantage 2004 Volume 1 toolset, and second, your job is to write applications, not to write applications that will help write the application.

To see what the options you have for user interface elements using the Infragistics toolset, take a look at the list in the Toolbox. Figure 7.3 is a list of the Windows Forms UI elements in the NetAdvantage 2004 Volume 1 toolset.

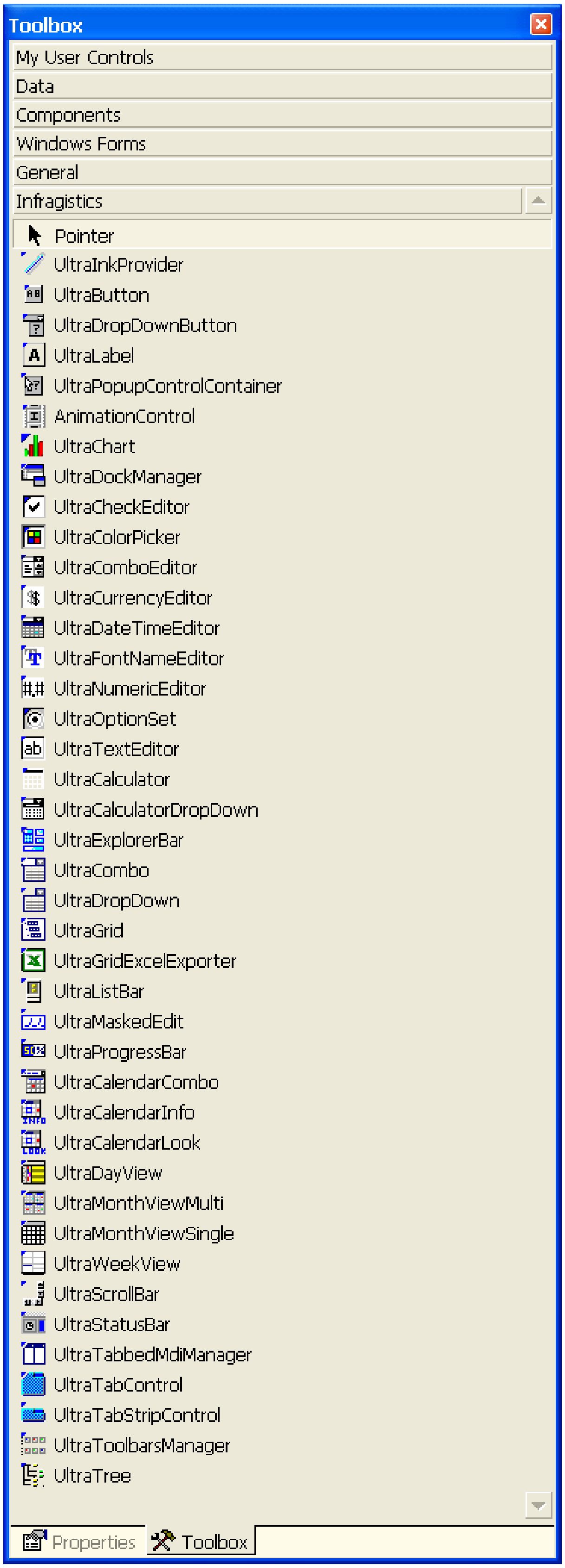
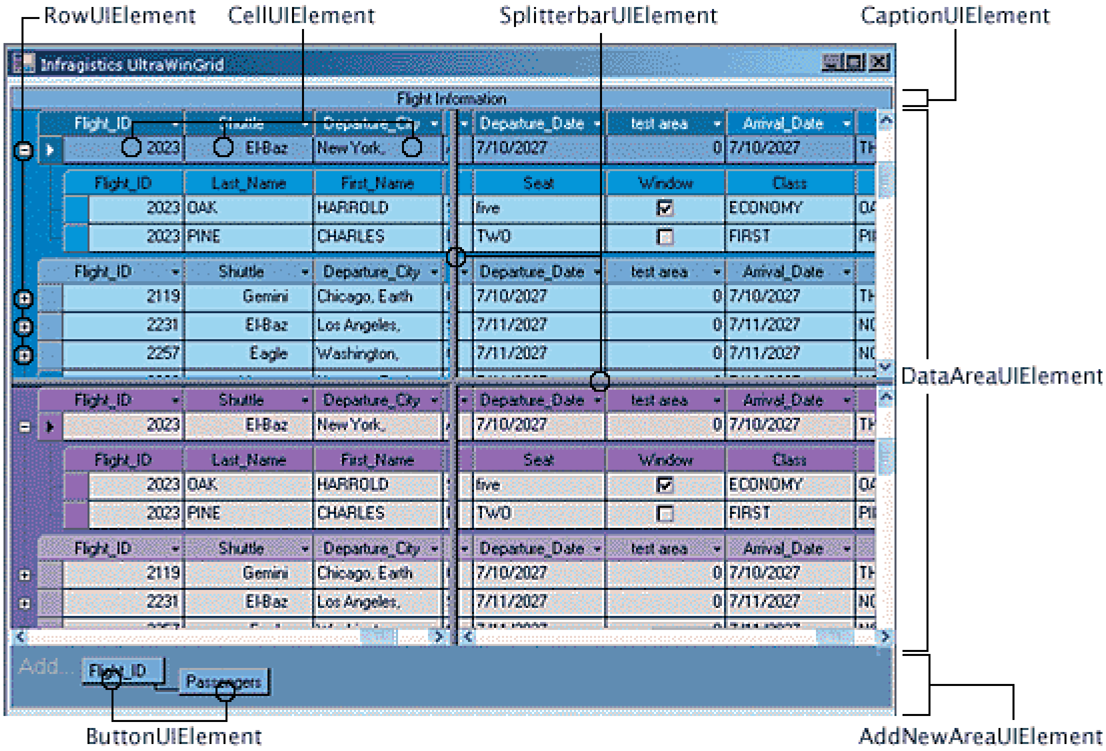

Toolbox
My User Controls
Data
Components
Windows Forms
General
Infragistics
Pointer
UltraInkProvider
UltraButton
UltraDropDownButton
UltraLabel
UltraPopupControlContainer
AnimationControl
UltraChart
UltraDockManager
UltraCheckEditor
UltraColorPicker
UltraComboEditor
UltraCurrencyEditor
UltraDateTimeEditor
UltraFontNameEditor
UltraNumericEditor
UltraOptionSet
UltraTextEditor
UltraCalculator
UltraCalculatorDropDown
UltraExplorerBar
UltraCombo
UltraDropDown
UltraGrid
UltraGridExcelExporter
UltraListBar
UltraMaskedEdit
UltraProgressBar
UltraCalendarCombo
UltraCalendarInfo
UltraCalendarLook
UltraDayView
UltraMonthViewMulti
UltraMonthViewSingle
UltraWeekView
UltraScrollBar
UltraStatusBar
UltraTabbedMdiManager
UltraTabControl
UltraTabStripControl
UltraToolbarsManager
UltraTree
Properties
Toolbox

Figure 7.3: Toolbox filled with Infragistics Presentation Layer elements.

As you can see, there are lots of items to choose from when designing a user interface. The nice thing about these tools is that they do not overlap with what Microsoft ships with Visual Studio .NET. You can decide what UI elements you need in your application, and choose whether or not to use tools from Infragistics or controls from Microsoft.

Similar to the framework which Windows Forms is built on, the Infragistics toolset is built on their own Presentation Layer Framework. This framework is used throughout all of the user interface elements you see in figure 7.3. To give you an idea of the how the various elements can make up a single tool, examine the WinGrid displayed in figure 7.4.



Flight_ID	Shuttle	Departure_City	Departure_Date	test area	Arrival_Date
2023	El-Baz	New York,	7/10/2027	0	7/10/2027

Flight_ID	Last_Name	First_Name	Seat	Window	Class
2023	OAK	HARROLD	five	☑	ECONOMY
2023	PINE	CHARLES	TWO	□	FIRST

Flight_ID	Shuttle	Departure_City	Departure_Date	test area	Arrival_Date
2119	Gemini	Chicago, Earth	7/10/2027	0	7/10/2027
2231	El-Baz	Los Angeles,	7/11/2027	0	7/11/2027
2257	Eagle	Washington,	7/11/2027	0	7/11/2027

Figure 7.4: Infragistics .NET Grid (WinGrid) User Interface Element.

Since the Presentation Layer Framework is consistent across all of the UI elements, you can expect the same properties, methods and events in a Combo UI element that you would have in the WinGrid UI element. For example, to customize the look and feel of the WinGrid, you would use the InitializeLayout event of the WinGrid.

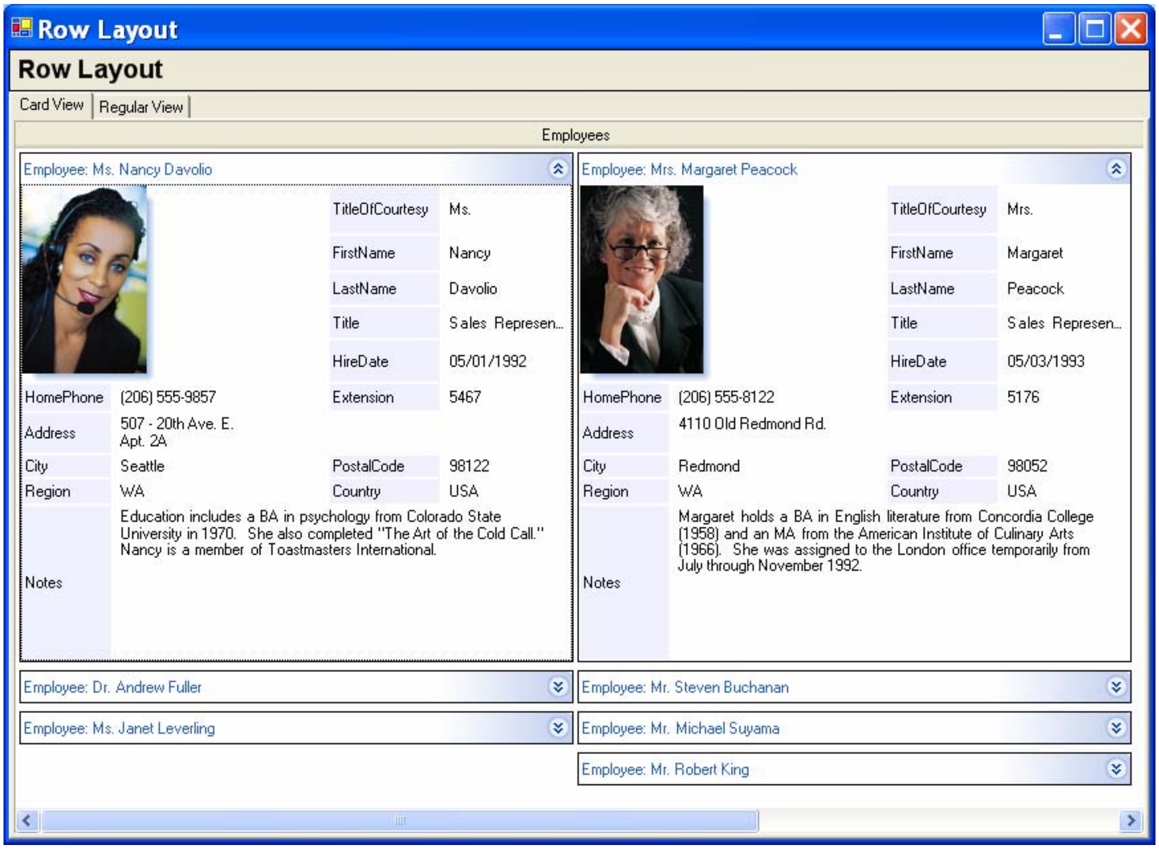
```
Private Sub IssueCombo_InitializeLayout(ByVal sender As System.Object, _
        ByVal e As Infragistics.Win.UltraWinGrid.InitializeLayoutEventArgs)
        Handles IssueCombo.InitializeLayout

    With e.Layout.Override
        .BorderStyleRow = UIElementBorderStyle.None
        .BorderStyleCell = UIElementBorderStyle.None
        .CellClickAction = UltraWinGrid.CellClickAction.RowSelect
        .RowSizing = RowSizing.Free
        .BorderStyleRow = UIElementBorderStyle.Dotted
    End With
```

```
        e.Layout.Bands(0).Columns("Title").Width = IssueCombo.Width

    End Sub
```

The same code can also be used in the InitializeLayout event of the Combo element. Since both of the tools are based on a common framework, the code for both tools is the same.

Creating a professional user interface in a fairly rapid amount of time is important too. Using the Infragistics tools, you can create clean and professional user interfaces without having to labor over creating your own custom controls based on the .NET Framework.

Figure 7.5, for example, took about 20 minutes to design and implement with full data binding.



Employee: Ms. Nancy Davolio

TitleOfCourtesy Ms.
FirstName Nancy
LastName Davolio
Title Sales Represen...
HireDate 05/01/1992
HomePhone (206) 555-9857
Extension 5467
Address 507 - 20th Ave. E.
Apt. 2A
City Seattle
PostalCode 98122
Region WA
Country USA
Notes Education includes a BA in psychology from Colorado State University in 1970. She also completed "The Art of the Cold Call." Nancy is a member of Toastmasters International.

Employee: Mrs. Margaret Peacock

TitleOfCourtesy Mrs.
FirstName Margaret
LastName Peacock
Title Sales Represen...
HireDate 05/03/1993
HomePhone (206) 555-8122
Extension 5176
Address 4110 Old Redmond Rd.
City Redmond
PostalCode 98052
Region WA
Country USA
Notes Margaret holds a BA in English literature from Concordia College (1958) and an MA from the American Institute of Culinary Arts (1966). She was assigned to the London office temporarily from July through November 1992.

Employee: Dr. Andrew Fuller

Employee: Mr. Steven Buchanan

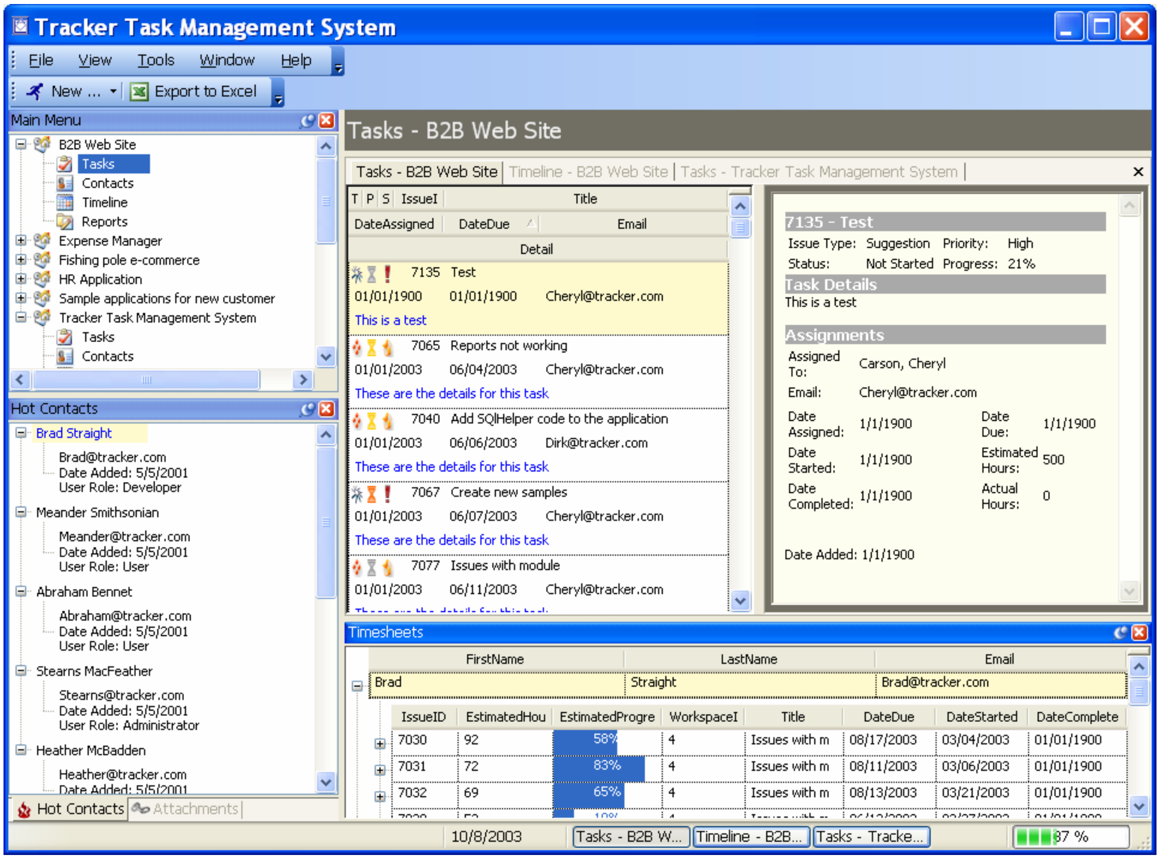
Employee: Ms. Janet Leverling

Employee: Mr. Michael Suyama

Employee: Mr. Robert King

Figure 7.5: Infragistics WinGrid Using the RowLayout Feature in Volume 1.

The user interface for Tracker, which you will learn more about in the next 2 chapters, took about 7 days to create, including wiring up all of the data to the controls. You can see in figure 7.6 that there are obvious user interface elements that are user friendly, and easy to figure out. This is not to say the Tracker is a shrink-wrap ready product, it is rather a good architecture with a nice UI built using the Presentation Layer Framework from Infragistics.


Tracker Task Management System
File View Tools Window Help
New ... Export to Excel
Main Menu
B2B Web Site
Tasks
Contacts
Timeline
Reports
Expense Manager
Fishing pole e-commerce
HR Application
Sample applications for new customer
Tracker Task Management System
Tasks
Contacts
Hot Contacts
Tasks - B2B Web Site
Tasks - B2B Web Site | Timeline - B2B Web Site | Tasks - Tracker Task Management System
Timesheets
Hot Contacts | Attachments
10/8/2003 | Tasks - B2B W... | Timeline - B2B... | Tasks - Tracke... | 87 %

Brad Straight
Brad@tracker.com
Date Added: 5/5/2001
User Role: Developer

Meander Smithsonian
Meander@tracker.com
Date Added: 5/5/2001
User Role: User

Abraham Bennet
Abraham@tracker.com
Date Added: 5/5/2001
User Role: User

Stearns MacFeather
Stearns@tracker.com
Date Added: 5/5/2001
User Role: Administrator

Heather McBadden
Heather@tracker.com
Date Added: 5/5/2001

T P S IssueI	Title	
DateAssigned	DateDue	Email
Detail		
7135	Test	
01/01/1900	01/01/1900	Cheryl@tracker.com
This is a test		
7065	Reports not working	
01/01/2003	06/04/2003	Cheryl@tracker.com
These are the details for this task		
7040	Add SQlHelper code to the application	
01/01/2003	06/06/2003	Dirk@tracker.com
These are the details for this task		
7067	Create new samples	
01/01/2003	06/07/2003	Cheryl@tracker.com
These are the details for this task		
7077	Issues with module	
01/01/2003	06/11/2003	Cheryl@tracker.com

7135 - Test
Issue Type: Suggestion Priority: High
Status: Not Started Progress: 21%

Task Details
This is a test

Assignments
Assigned To: Carson, Cheryl
Email: Cheryl@tracker.com
Date Assigned: 1/1/1900 Date Due: 1/1/1900
Date Started: 1/1/1900 Estimated Hours: 500
Date Completed: 1/1/1900 Actual Hours: 0

Date Added: 1/1/1900

FirstName	LastName	Email
Brad	Straight	Brad@tracker.com

IssueID	EstimatedHou	EstimatedProgre	WorkspaceI	Title	DateDue	DateStarted	DateComplete
7030	92	58%	4	Issues with m	08/17/2003	03/04/2003	01/01/1900
7031	72	83%	4	Issues with m	08/11/2003	03/06/2003	01/01/1900
7032	69	65%	4	Issues with m	08/13/2003	03/21/2003	01/01/1900

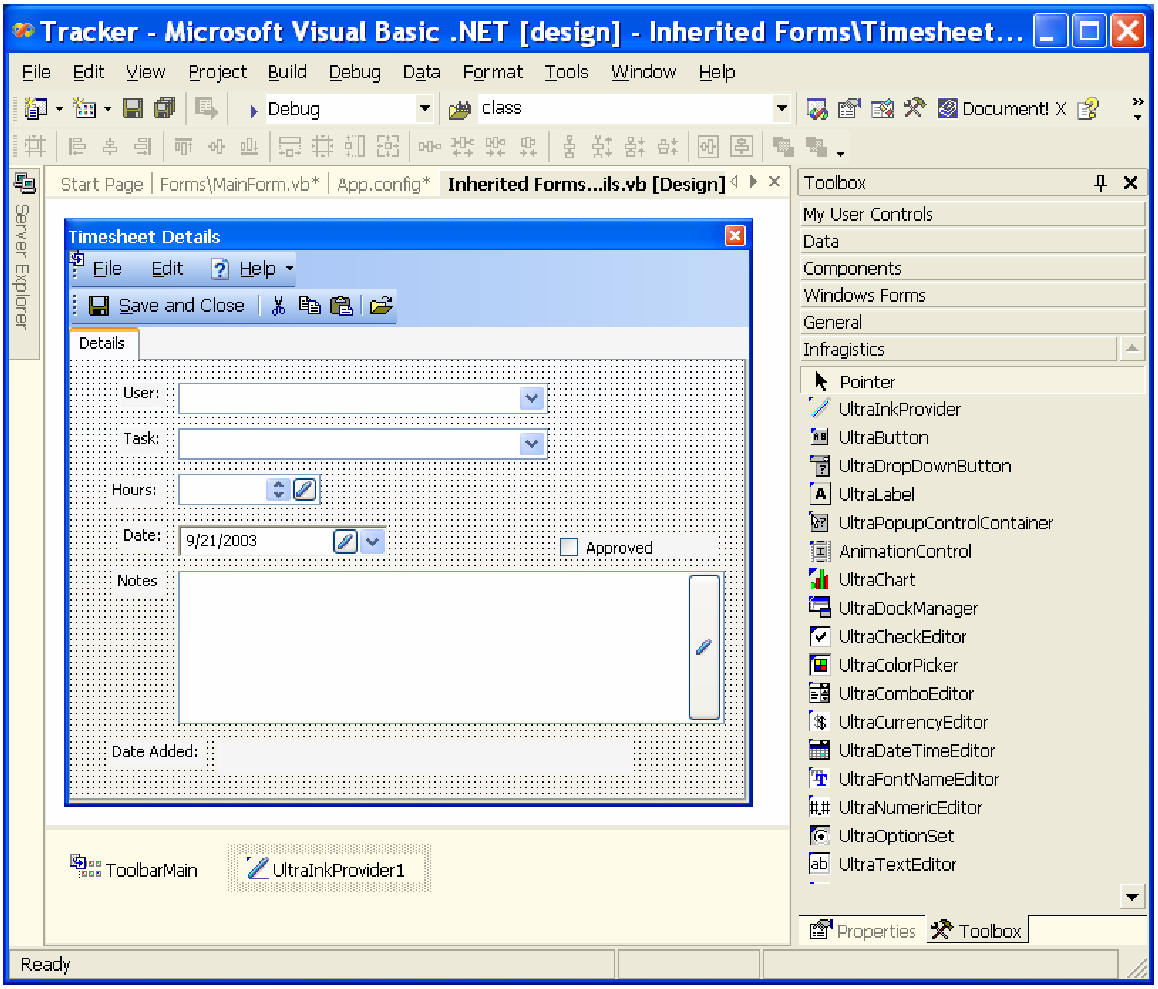
Figure 7.6: Tracker User Interface Created with NetAdvantage 2004 Volume 1 Toolset.

Just like choosing Visual Studio .NET as your tool to build robust applications based on the .NET Framework, the Presentation Layer toolset of choice to create awesome user interfaces should be the UI elements in the NetAdvantage 2004 Volume 1 suite.

What about Tablet PC's?

I said in an earlier chapter that the Tablet PC is the next killer app. It is actually a piece of hardware that runs Windows XP Tablet PC Edition. The good news about the Tablet PC is that you can start writing killer apps for it right now using the .NET Ink Provider tool that ships with NetAdvantage 2004 Volume 1. By simply dragging the .NET Ink Provider from the Toolbox to a form with UI elements from NetAdvantage 2004 Volume 1, the UI elements automatically are ink-enabled.

The .NET Ink Provider is a non-visual tool that enhances the capabilities of existing NetAdvantage 2004 Volume 1 elements on a Windows Form by adding a button that you can click which drops down the ink pad. Figure 7.7 demonstrates the .NET Ink Provider at design time. Notice the small buttons with the pen graphic on the user interface elements.



Date: 9/21/2003

Figure 7.7: The .NET Ink Provider Tool at Design Time in Visual Studio .NET.

At runtime, the ink pad looks like figure 7.8. Note that figure 7.8 is showing the keypad, the number pad, and the free text pad. This gives an end user of your applications ultimate flexibility when working with your application.

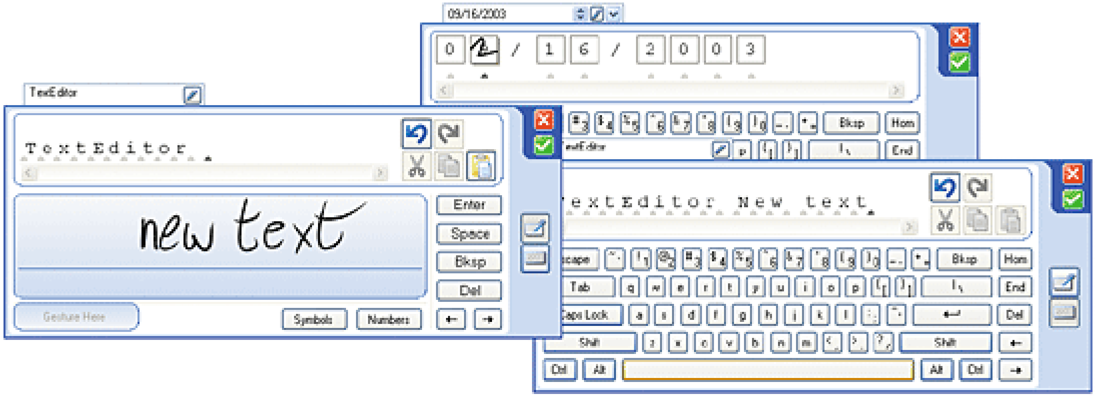


Figure 7.8: The Ink Pad at Runtime.

To get an idea of the control that the .NET Ink Provider will give you, figure 7.9 shows the various options available on the tool.

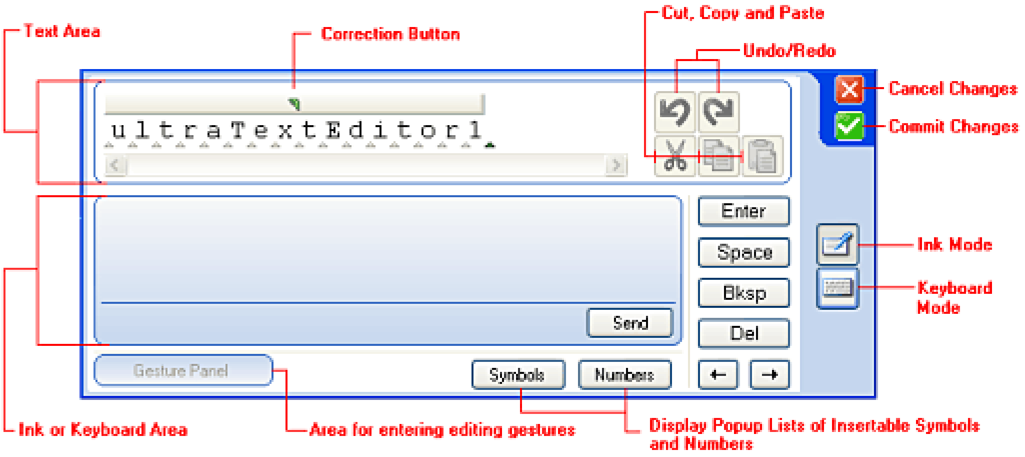


Figure 7.9: The Ink Pad Features and Options.

The capabilities that the ink enabled tools from NetAdvantage 2004 Volume 1 give you as a developer to write robust Tablet PC aware applications is almost overwhelming. Along with the many other UI elements that are in the Toolbox from the Infragistics toolset, the .NET Ink Provider will just add to the types of user interfaces you can create.

Summary for Today and What is Next

This chapter discussed the various aspects of a good user interface, and what it takes to determine what a good user interface is. Acceptance by end users of your applications is the ultimate goal, so tailoring the look and feel and functionality of applications they are used to using on an everyday basis is extremely important. You also learned how easy it is to ink-enable applications using the .NET Ink Provider that ships with the NetAdvantage 2004 Volume 1 toolset. Creating ink enabled applications is becoming more and more important, and using the tools available in the Volume 1 toolset makes this easier than ever.

8

Chapter 8: Implementing the User Interface

The next several chapters concentrate on the user interface of Tracker, and how it was developed. In this chapter, you'll learn how the main form of the application was created, where the dockable controls fit in, and how the MDI child forms were created.

Creating the Tracker User Interface

When deciding what the Tracker user interface was going to look like, there were some things I knew I needed to do:

- Create a Visual Studio .NET like interface with dockable windows
- Create an Outlook 2003 look and feel
- Use a MDI designed application

The reason for this design choice was the following:

- With the dockable sections of the screen, I knew that a user with a large screen resolution, maybe 1600 x 1200, could take full advantage of having many windows open and fully visible on their display. If the user has a smaller resolution, maybe 1024 x 768, the screen area can be increased or decreased by pinning and unpinning any of the docked windows.

- With the Outlook 2003 look and feel, I tried to create an interface that was easy to read, and sort of forward looking. If you have used the 3 pane look and feel of Outlook 2003, you will probably agree that it has a very slick look and feel.

- Since there would need to be several forms that need to be displayed full screen, a MDI design was a must. In the past, I have created user controls and just loaded them dynamically at runtime, but the .NET tabbed MDI UI element from NetAdvantage 2004 Volume 1 allowed me to have a very slick looking MDI application.

The main form (MainForm.vb) of the application would contain the dockable windows. Deciding what the dockable windows would contain was easy:

- The Main Menu
- Hot Contacts
- Issue Attachments
- Timesheets

This was easy to decide because I knew that each of these data elements contained important information about the workspace as a whole, not necessarily a specific Task in the Issues table. Also, none of the items listed deserved a full screen on their own; there just was not that much data to display.

For the MDI forms, there are a few main tables that need more than just a bunch of rows displayed in a WinGrid. They needed a specific user interface to be designed around how the data needed to be presented to the user. These tables are:

- Issues (Tasks)
- Timeline
- Company and Users
- Reports

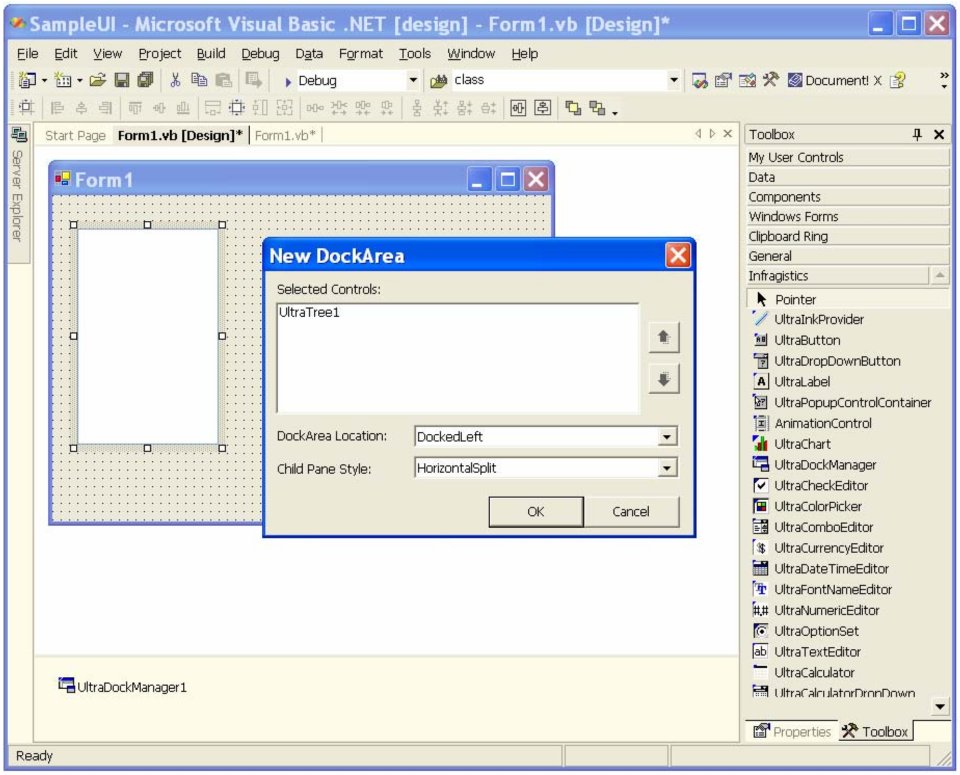
With that decided, it was time to start adding the controls.

***Note**: I am not going to go step by step in a tutorial fashion on how the application was created, I think you can inspect the code and forms for days to check all of that out. I am rather going to go through how the MainForm.vb form was created in a simple sample application, talk about key points in the design, and support the screen shots with an explanation and then whatever the key code for the form is necessary.*

The first UI elements that needed to be added to the MainForm.vb were the dockable windows. To accomplish this, I used the following UI for their corresponding windows:

- Main Menu – WinTree (.NET Tree from NetAdvantage 2004 Volume 1)
- Hot Contacts – WinTree (.NET Tree from NetAdvantage 2004 Volume 1)
- Timesheets – WinGrid (.NET Grid from NetAdvantage 2004 Volume 1)
- Attachments – WinGrid (.NET Grid from NetAdvantage 2004 Volume 1)

Using the DockManager from NetAdvantage 2004 Volume 1, when you add the UI elements to a form, you can simply right-click on the element and tell it where to dock. Figure 8.1 shows this in action.



New DockArea

Selected Controls:
UltraTree1

DockArea Location: DockedLeft

Child Pane Style: HorizontalSplit

Figure 8.1: Docking UI Elements with the DockManager

As you are adding elements to dock, you can choose the area of the screen they should sit. Once you have docked an element, you can drag it around the other docked areas of the form, and drop it wherever you desire. Figure 8.2 demonstrates the UltraTree2 element being dragged to the lower left area of the UltraGrid1 element, which I had already docked.

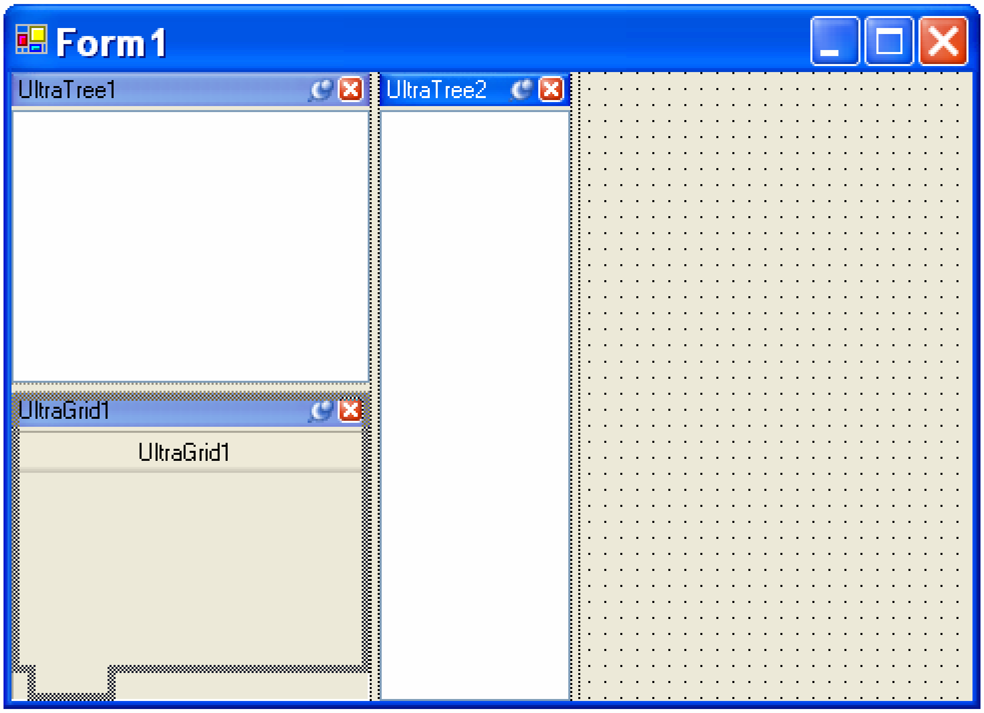


Figure 8.2: Docking UI Elements at design time.

After I added the remaining WinGrid, I added the TabbedMDIManager tool. Using the TabbedMDIManager, you can have your MDI child windows appear in tabbed interface. This works exactly like the tabbed MDI interface represented in figure 8.3 that we have all grown to love using Visual Studio .NET.

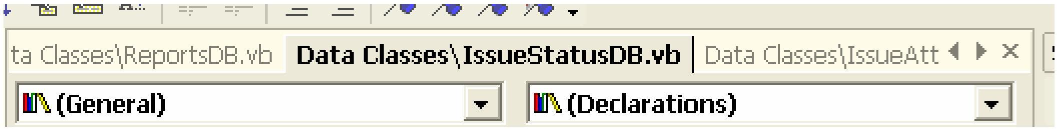


Figure 8.3: Tabbed MDI of Visual Studio .NET.

To create a tabbed MDI, simply drag the TabbedMDIManager from the Toolbox onto your form. It will display as a non-visual control. When you create new instances of MDI children, they will automatically show up in a tabbed interface. Figure 8.4 is the result of adding 2 forms to the sample application, and simply showing them with the following code:

```
Private Sub Form1_Load(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles MyBase.Load

    Dim f As New Form2
    f.MdiParent = Me
    f.Show()
```

```
        Dim f2 As New Form3
        f2.MdiParent = Me
        f2.Show()

    End Sub
```

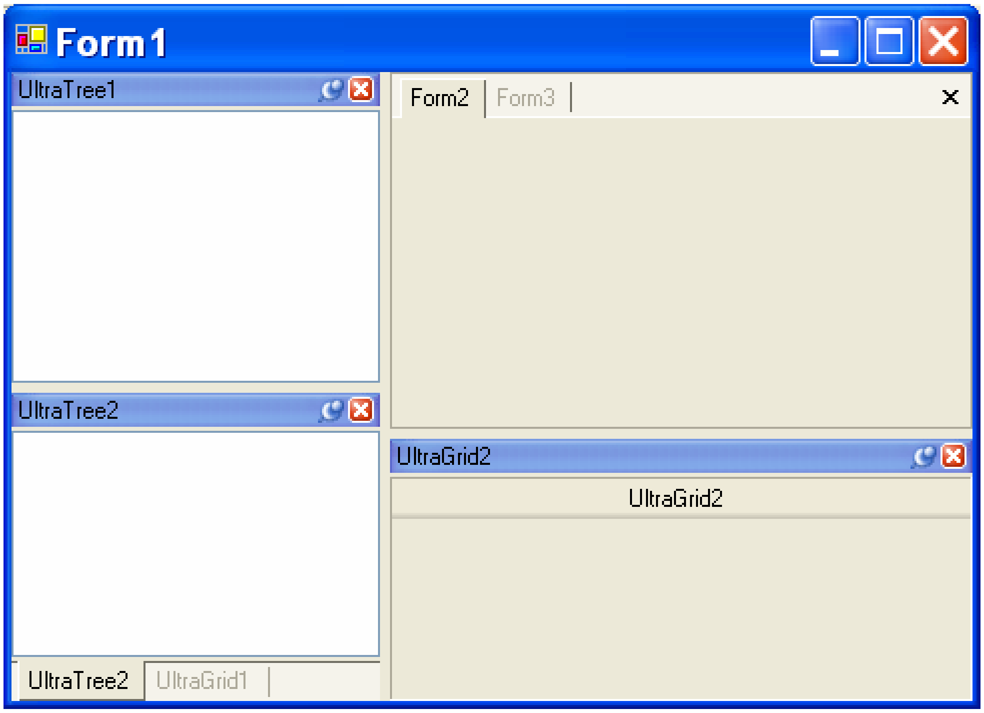


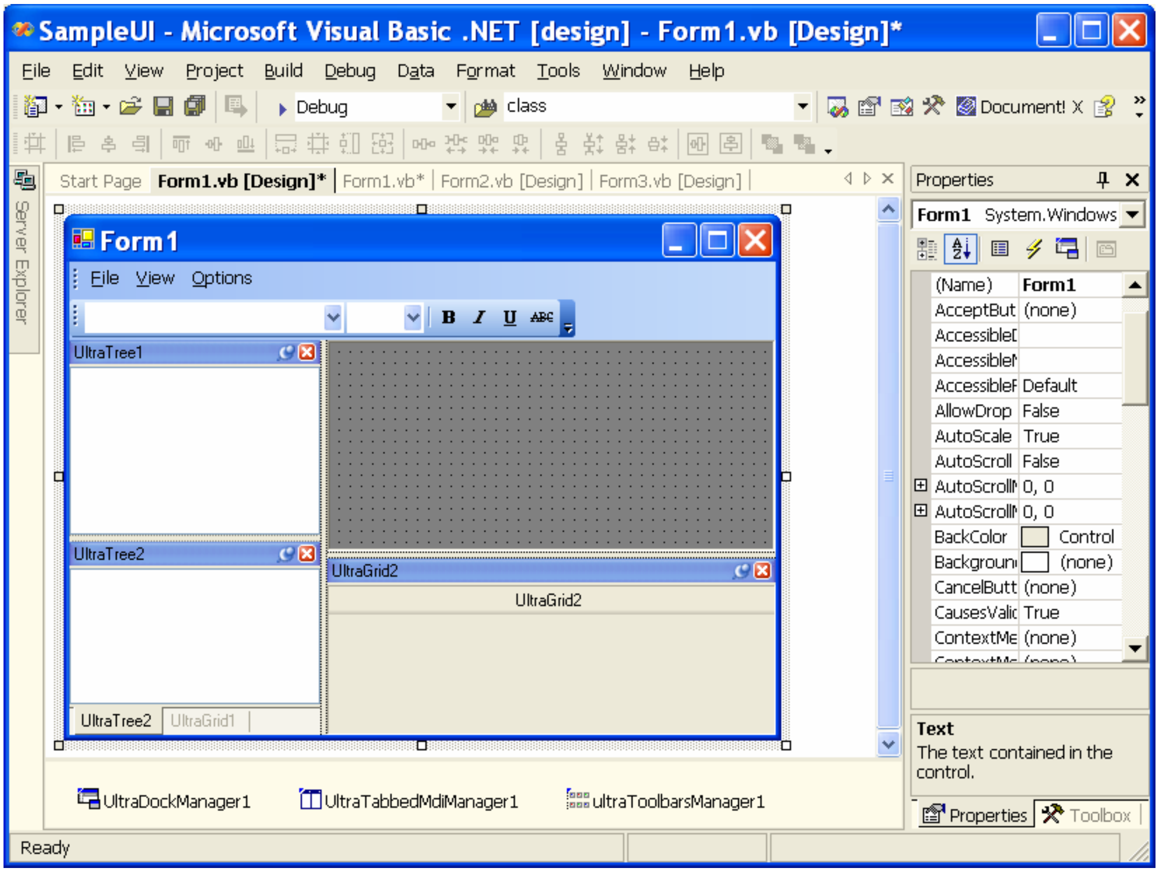
Figure 8.4: Tabbed MDI of using the TabbedMDIManager.

As you can see, creating a prototype user interface is pretty easy. The application in figure 8.4 took all of 3 minutes to create.

The next step was to add the main menu. For this, I used ToolBarsManager, or .NET Toolbar, from NetAdvantage 2004 Volume 1. Using the ToolBarsManager tool, you can create menus that look and feel exactly like any Microsoft Office application, with full theme support and graphic support for menu items. To make this easy, I found the ProcessViewer sample that shipped with the samples included in NetAdvantage 2004 Volume 1, and simply copied the ToolBarsManager from that sample onto my form. This gave me a good start to deciding what my menus actually needed. You can find the ProcessViewer sample at the following default location.

C:\Program Files\Infragistics\UltraWinToolbars\v3.00\Samples\Vb\ProcessViewer VB

After I added the ToolBarsManager tool, my form looked like figure 8.5 at design time.



(Name)	Form1
AcceptBut	(none)
AccessibleI	
AccessibleN	
AccessibleF	Default
AllowDrop	False
AutoScale	True
AutoScroll	False
AutoScrollI	0, 0
AutoScrollI	0, 0
BackColor	Control
Backgroun	(none)
CancelButt	(none)
CausesValic	True
ContextMe	(none)

Text
The text contained in the control.

Figure 8.5: Form1 after adding the ToolBarsManager.

At this point, you can see the user interface coming together. With literally no code, I have a visually appealing UI. At runtime, with the docked windows unpinned, the application looks like figure 8.6.

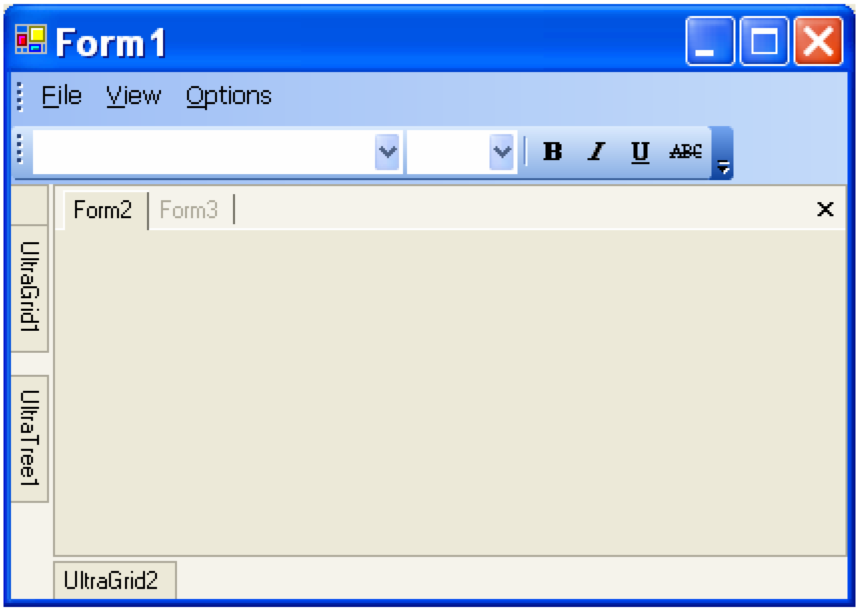


Figure 8.6: Sample UI with docked windows unpinned.

Figure 8.6 is ultimately what the MainForm.vb in Tracker looks like. To see how the data gets into the UI elements that are docked using the DockManager, we need to switch to the actual Tracker application and look at how that is implemented. Hopefully you can see at this point how truly simple it is to create a very cool UI using the UI elements that ship with NetAdvantage 2004 Volume 1. I am not trying to be a marketing person when I say that, I just get excited about cool UI's.

Adding Data to the User Interface Elements

All of the code that you learned about in chapters 2 through 6 now comes into play. To wire up the data from the database into the controls, you simply need to call the methods in the BLL for the type of data that you need. In the MainForm.vb class, there is a call to the LoadMainMenu method in the InitializeComponent event. The LoadMainMenu method loads the workspaces in the Workspaces table into the WinTree element that is docked on the left of the MainForm.vb. A portion of the code in the LoadMainMenu method is show here.

```
Sub LoadMainMenu()

    '     Clear the current Nodes collection
    MainMenu.Nodes.Clear()

    Dim bl As New Tracker.BusinessLogic.Workspaces
    Dim ws As Info.WorkspacesInfo()

    If App.Role = RolesTypeList.User Then
        ' Get all of the that equal this users companyID
        ws = bl.GetWorkspaces(Info.WorkspaceTypeList.Company, _
                App.CompanyID)
    Else
        ' Get get all of the roles
        ws = bl.GetWorkspaces()
    End If

    Dim w As Info.WorkspacesInfo
    For Each w In ws
        ' Add the Workspace name node, and give it the StartPage tag
        Dim node As UltraTreeNode = MainMenu.Nodes.Add _
                (w.WorkspaceID, w.Description)
        node.LeftImages.Add(9)
        node.Tag = w.WorkspaceID & "~" & w.Description _
                    & "~" & w.CompanyID & "~" & "StartPage"
        node.Key = node.Tag
```

Most of this code should look familiar. We are asking the BLL to give us all of the Workspaces for the *CompanyID* that is stored in an application level variable. The *CompanyID* is based upon what user is logged in to Tracker, and is stored in a public property for use throughout the application. Once the data is filled into the *WorkspacesInfo()* type, there is a loop to get the data for each *WorkspaceInfo* item in the *WorkspacesInfo()* ArrayList. During the loop, the items are added to the WinTree control. The following code adds the top level node which represents the workspace:

```
        Dim node As UltraTreeNode = MainMenu.Nodes.Add _
                (w.WorkspaceID, w.Description)
        node.LeftImages.Add(9)
        node.Tag = w.WorkspaceID & "~" & w.Description _
                    & "~" & w.CompanyID & "~" & "StartPage"
        node.Key = node.Tag
```

In the Tag property of the node, as much information as possible about this Workspace is stored. This information is used by the forms that are loaded based on what item the user clicks on the WinTree. To add the sub nodes, you simply create another TreeNode object, and add it to the parent node, as the following code demonstrates.

```
' now add the rest of the nodes
' at this point you can determine if the person
' is not a manager, you can not display certain nodes.
Dim subNode As UltraTreeNode
subnode = node.Nodes.Add()
With subNode
    .LeftImages.Add(10)
    .Text = "Tasks"
    .Tag = w.WorkspaceID & "~" & w.Description _
            & "~" & w.CompanyID & "~" & "Tasks"
    .Key = .Tag
End With
```

Once this code executes, you get something that looks like figure 8.7.

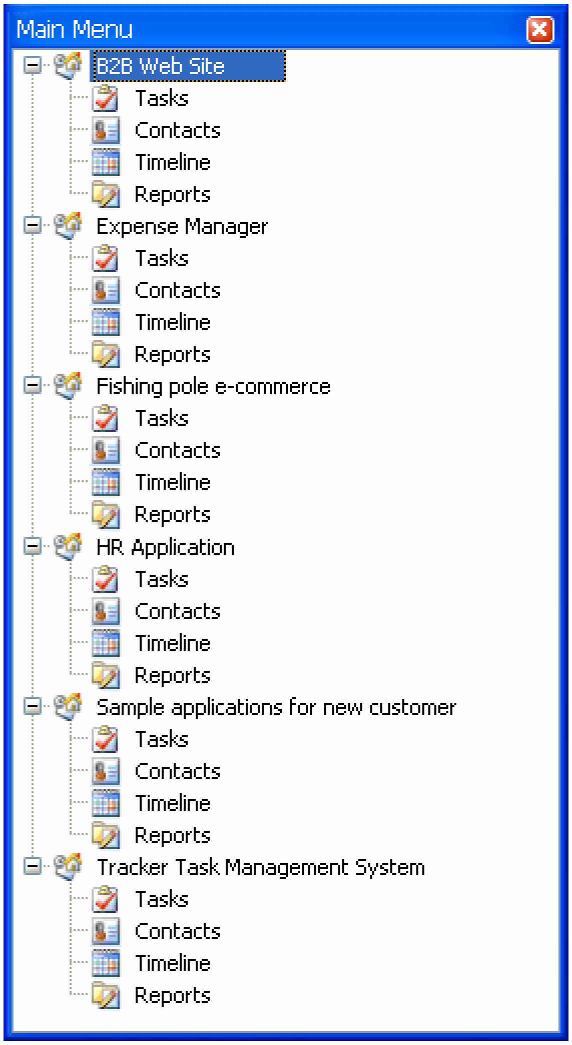


Figure 8.7: WinTree UI element with Workspaces listed.

When Tracker starts, the only thing that happens is the MainMenu (the WinTree) is loaded. When an item on the main menu is selected, the information that was stored in the Tag property of the node is used to determine what form to load as a MDI child, and what parameters to use to load the remaining docked windows.

Examining the Main Menu Events

When a node on the MainMenu is selected, the AfterActivate event handler of the WinTree fires. In this method, the *Tag* property of the node is read, and the information contained is used to determine what the *WorkspaceID*, *Description*, *CompanyID* and the friendly name of the node. The code that reads this information is as follows:

```
Dim nodeTag() As String = Split(e.TreeNode.Tag.ToString, "~")
```

The data in the Tag is delimited by the "~" character, so using the Split function, each item of data is stored in the *nodeTag()* array. Once this occurs, there is a check to make sure the node is not already the currently selected node in the WinTree. The method to load the remaining docked windows is called with this code:

```
' if the workspace has switched, then load the hot contacts,
' attachments and timesheet for this workspace
If App.CurrentWorkspaceID <> nodeTag(0) Then
    StartLoadingContacts()
    StartLoadingAttachments()
    StartLoadingTimesheet(nodeTag(0))
End If
```

Each of these methods will load their respective user interface elements on the form. The cool thing about this is the methods all run on their own threads. This means that if there is a slow connection, or there is a lot of data, the user interface is not blocked while the data is loaded. To see how this works, you can examine the following code, which represents the *StartLoadingAttachments* method.

```
    Private _bllAttach As New Tracker.BusinessLogic.IssueAttachments
    Private _attachInfo As Tracker.Info.IssueAttachmentsInfo()
    Private _attachThread As Thread
    Private _attachThreadStart As New _
            ThreadStart(AddressOf LoadAttachments)
    Private _callData bindAttachments As New _
            MethodInvoker(AddressOf Me.BindAttachments)
    Private _gridDoubleClickFlag As Boolean
    Private _issueAttachmentPath As String
    Private _issueAttachmentID As String

    Sub LoadAttachments()
        Dim formTag() As String = Split(Me.Tag, "~")
        _attachInfo = _bllAttach.GetIssueAttachmentsByWorkspace _
                (App.CurrentWorkspaceID)
        Me.BeginInvoke(_callData bindAttachments)
    End Sub

    Sub StartLoadingAttachments()
        Try
            _attachThread = New Thread(_attachThreadStart)
            With _attachThread
```

```
                .IsBackground = True
                .Name = "LoadAttachmentsGridThread"
                .Start()
            End With
        Catch ex As Exception
            ShowError(ex)
        End Try
    End Sub

    Public Sub BindAttachments()
        AttachmentsGrid.DataSource = _attachInfo
    End Sub
```

Each of the methods that load any data on the MainForm.vb follows the same pattern. A thread is created, and the data is retrieved on that thread. Once the data is retrieved, it is loaded into its docked window UI element.

The last bit of code on the main form I want to highlight is the loading of the MDI child. Remember that the TabbedMDIManager is used to handle the display of the MDI children, so the code just creates new instances of whatever form it needs to, and shows the form. Since there are several MDI forms that could be loaded, I did not want to use a Select Case or If … Then statement to check which form to load. Instead, I used the value of the *Text* property of the selected node, and load the form using Reflection. This is accomplished with the following code.

```
' Since the form is not in the current MdiChildren
' collection, load it up using reflection

Dim asm As [Assembly] = [Assembly].LoadFrom _
        ([Assembly].GetExecutingAssembly.CodeBase)

Dim formObject As Type = asm.GetType("Tracker." & _
        e.TreeNode.Text.ToString & "Main")

Dim formActivator As Object = Activator.CreateInstance(formObject)
f = DirectCast(formActivator, Form)
f.Tag = e.TreeNode.Tag
f.Text = TopLabel.Text
f.MdiParent = Me
f.Show()
```

This few lines of code will handle all of the MDI child loading for the MainForm.vb class. In order for the form that is being loaded to know what data to load, the *Tag* property of the newly instanced form is set to the *Tag* property of the selected node.

```
f.Tag = e.TreeNode.Tag
f.Text = TopLabel.Text
```

This way, the form that was just created can in turn use that information to determine what data to load and how to display it. The following code is from the Load event of the TasksMain MDI child:

```
    Private Sub TasksMain_Load(ByVal sender As Object, _
            ByVal e As System.EventArgs) Handles MyBase.Load

        ' Save the workspace id from the form tag to a local variable
        Dim formTag() As String = Split(Me.Tag, "~")
        _workspaceID = formTag(0)
        ' Load the tasks on a thread
        StartLoading()
    End Sub
```

Notice the first thing that happens is the *Tag* property of the form is retrieved, and set to a variable on the form. Each of the forms in the *MdiChildren* folder of the Tracker project follows this pattern, making it easy for the form to know what to load.

Synchronizing the MDI Children with the Main Form

An important aspect of writing MDI applications using docked windows on a main controller form is how to make sure the data in the docked window changes based on what form is active. For example, in Tracker, the docked windows are displaying information based on that node is selected in the MainMenu WinTree element. Since this is on the main form, it is easy to simply re-load the data if the user selected another workspace.

In a tabbed MDI environment, the user can switch tabs at random, so the main form needs to make sure that the docked windows correctly represent the information in the activated form. To make this happen, each MDI child has code in its *Activated* event handler, for TasksMain, it looks like this:

```
Private Sub TasksMain_Activated(ByVal sender As Object, _
        ByVal e As System.EventArgs) Handles MyBase.Activated

    ' set the current node on the parent form
    ' This will reload the main form with the correct data based on
    ' what MDIChild is now active
    CType(Me.MdiParent, MainForm).ActivateNode(Me.Tag)
End Sub
```

What is happening is the *ActivateNode* event handler on the MainForm.vb is being called from the MDI child. When this occurs, the code that would execute if you actually clicked a node on the MainMenu WinTree fires, thus synchronizing the workspace information represented on the active MDI child and the workspace specific data in the docked windows of the main form.

Summary for Today and What is Next

This chapter covered the basics of the main form of the Tracker application. You learned about how the form was designed, and how the data is wired up to the form. You should also understand how the information is communicated from the nodes that are selected on the main menu and their corresponding MDI children using the Tag property of the various controls. This is important, since you do want to somehow manage what was selected with a global variable, which would get messy. Another key aspect covered in this chapter is how Tracker loads data on threads other then the main application thread. This is important when it comes to writing a responsive user interface. In the next chapter, you are going to learn about the TasksMain form and how to customize the WinGrid to do something other than list data in a single row.

9

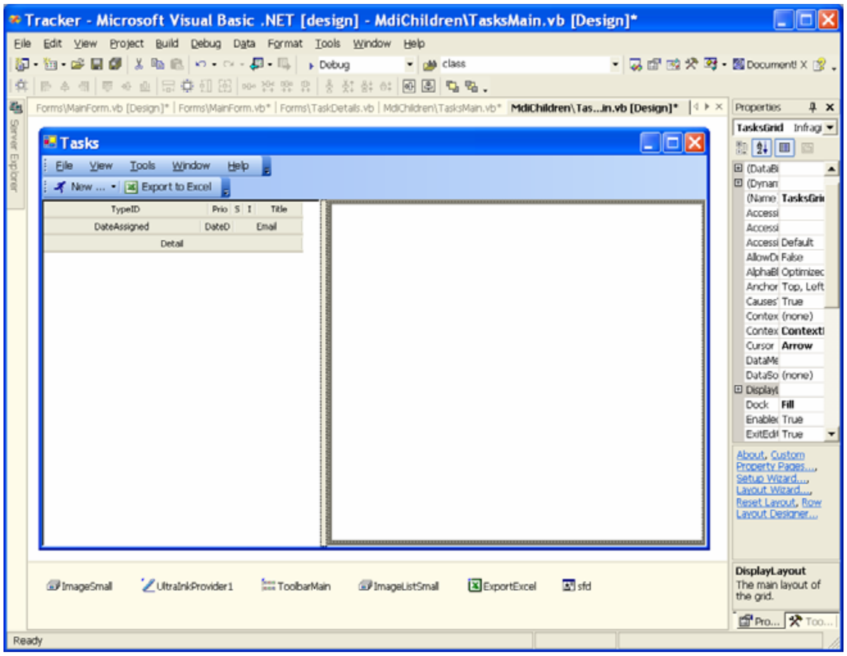
Chapter 9: Enhancing the User Interface

In the last chapter, you learned a lot about how data is handled in the Tracker application. In this chapter, you are going to learn about several more key UI components of Tracker. You will learn how the WinGrid on the TasksMain form was created, how the Toolbars work, and finally how the various modal dialogs in the application work.

Understanding the TasksMain Form

The TasksMain form represents the data from the Issues table in the Tracker database for the currently selected workspace. There are two main user interface elements on the TasksMain page, the WinGrid and the Web Browser COM component. At runtime, when a user selects an item in the WinGrid, data is retrieved from a cell in the grid, and the specific task data is retrieved from the data source. Once the data is retrieved back from the data source, it is serialized into an Xml document, and transformed on the fly into an HTML document using an XSL stylesheet for display in the Web Browser control. Before we look at the code for how that all works, let's examine how the TasksGrid UI element was created.

At design time, the TasksMain form looks like figure 9.1.

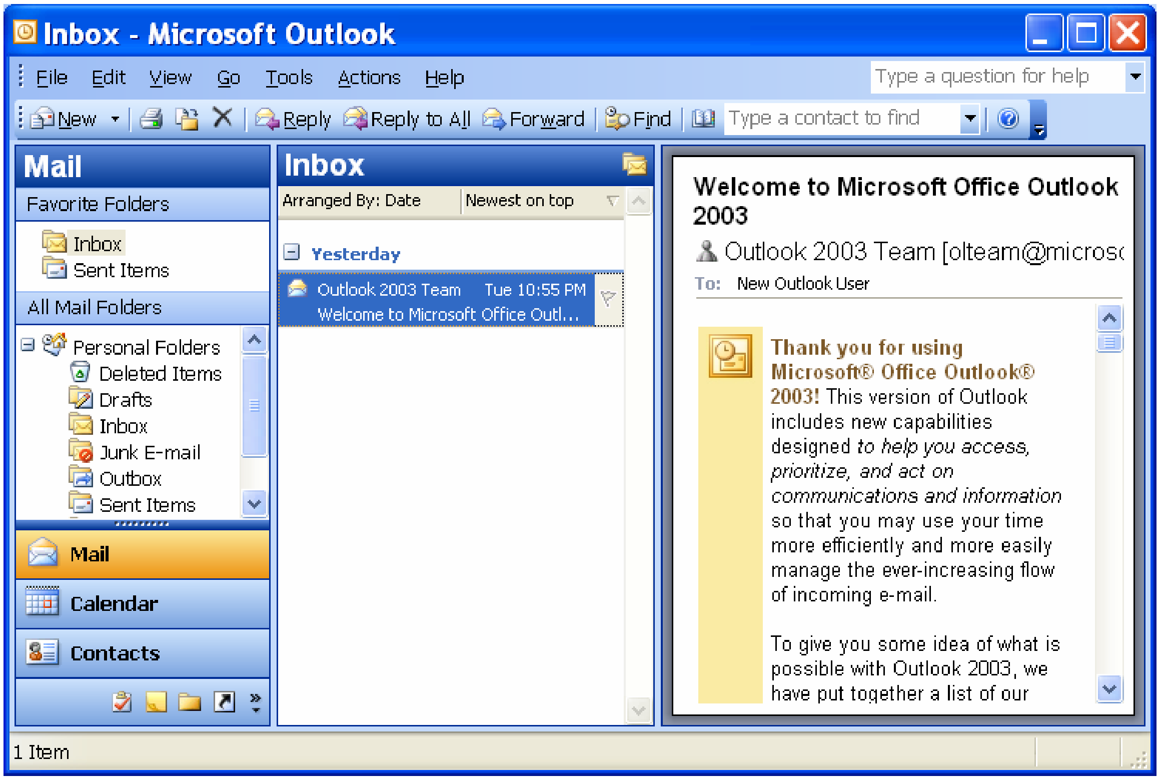

Tracker - Microsoft Visual Basic .NET [design] - MdiChildren\TasksMain.vb [Design]*
File Edit View Project Build Debug Data Format Tools Window Help
Debug
class
Document! X
Forms\MainForm.vb [Design]* | Forms\MainForm.vb* | Forms\TaskDetails.vb | MdiChildren\TasksMain.vb* | MdiChildren\Tas...in.vb [Design]*
Server Explorer
Tasks
File View Tools Window Help
New ...
Export to Excel
TypeID
Prio
S
I
Title
DateAssigned
DateD
Email
Detail
Properties
TasksGrid Infragi
About, Custom Property Pages..., Setup Wizard..., Layout Wizard..., Reset Layout, Row Layout Designer...
Pro...
Too...
ImageSmall
UltraInkProvider1
ToolbarMain
ImageListSmall
ExportExcel
sfd
Ready

⊞ (DataBi	
⊞ (Dynan	
(Name	TasksGri
Accessi	
Accessi	
Accessi	Default
AllowDr	False
AlphaBl	Optimizec
Anchor	Top, Left
Causes'	True
Contex	(none)
Contex	Context
Cursor	Arrow
DataMe	
DataSo	(none)
⊞ Displayl	
Dock	Fill
Enablec	True
ExitEdi	True

DisplayLayout
The main layout of the grid.

Figure 9.1: TasksMain form at Design time.

This form was created by using a panel docked left, a splitter bar, and a panel docked fill. I then added the WinGrid UI element to the left panel, and set its Dock property to fill. The Web Browser control was then added to the panel on the right of the form, and I strategically tried to make it look like the Reading Pane of Outlook 2003 by centering it on the panel and setting its Anchor property to Top, Left, Right and Bottom. If you are not familiar with Outlook 2003, take a gander at figure 9.2.



Welcome to Microsoft Office Outlook 2003

Outlook 2003 Team [olteam@microsc

To: New Outlook User

Thank you for using Microsoft® Office Outlook® 2003! This version of Outlook includes new capabilities designed *to help you access, prioritize, and act on communications and information* so that you may use your time more efficiently and more easily manage the ever-increasing flow of incoming e-mail.

To give you some idea of what is possible with Outlook 2003, we have put together a list of our

Figure 9.2: Outlook 2003 User Interface with Reading Pane.

I think this UI is really cool, so I wanted to mimic it somewhat.

Designing the TasksGrid

To design the TasksGrid, I needed to somehow let it know that I wanted to have a single row of data represented on three rows in the grid. This was not as hard as you might think. If you right click on the WinGrid at design time, you have several options on customizing the grid. In figure 9.3, I have cropped the relevant grid specific options.

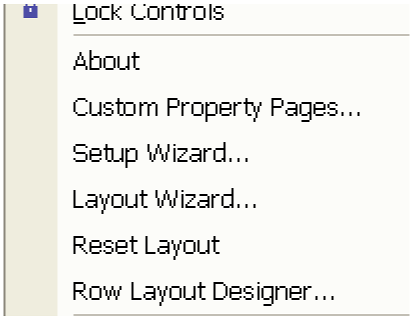


Figure 9.3: Context menu for the WinGrid UI element.

In the Tracker architecture the data is treated as unbound at design time. In other words, you do not have the schema of the data available to the various designers in the Visual Studio .NET IDE. You cannot use any of the wizards offered by the grid, since they depend in schema information to create a grid layout for you. That is not a big deal, since you can still customize the layout using the Custom Property Pages option.

Using the Groups and Columns tab of the Custom Property Pages dialog of the WinGrid, you can add as many groups and levels as you want for your grid to display. You can then add columns, which map to column names in the data source. For the TasksGrid, the Groups and Columns I created look like figure 9.4.

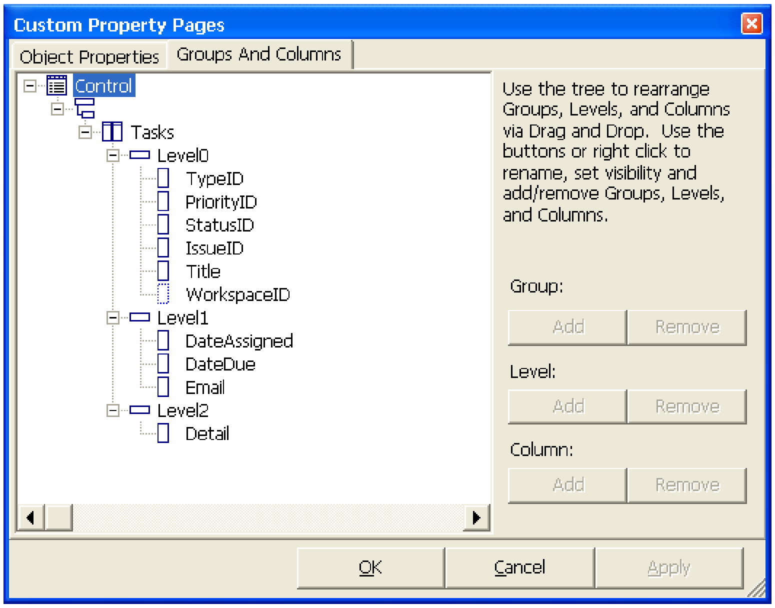


Figure 9.4: Groups and Columns design for the TasksGrid UI element.

Each Level in the layout represents a row in the grid, but a single row from the data source. Once these columns were added, the grid looks like figure 9.1 at design time. This is a good start to how the grid should look, but it needed more customization. I wanted the certain columns hidden, and I wanted certain columns to have specific width properties. To do this, you would use the *InitializeLayout* event of the WinGrid. In the *InitializeLayout* event handler, you can set column and row specific properties before the grid is displayed on the form. This sort of customization is very powerful, allowing you complete control over how the grid is presented. The code for the *TasksGrid_InitializeLayout* event handler is listed below.

```
Private Sub TasksGrid_InitializeLayout(ByVal sender As System.Object, _
    ByVal e As Infragistics.Win.UltraWinGrid.InitializeLayoutEventArgs) _
    Handles TasksGrid.InitializeLayout

    With Me.TasksGrid.DisplayLayout.Bands(0)
        .Columns("StatusID").Header.Fixed = True
        .Columns("StatusID").Header.Column.MaxWidth = 13
        .Columns("TypeID").Header.Fixed = True
        .Columns("TypeID").Header.Column.MaxWidth = 13
        .Columns("PriorityID").Header.Fixed = True
```

```
            .Columns("PriorityID").Header.Column.MaxWidth = 13
            .Columns("IssueID").Header.Fixed = True
            .Columns("IssueID").Header.Column.MaxWidth = 45
            .Columns("DateAssigned").Header.Fixed = True
            .Columns("DateAssigned").Header.Column.MaxWidth = 80
            .Columns("DateAssigned").Header.Column.MinWidth = 80
            .Columns("DateDue").Header.Column.MinWidth = 80
        End With

        With e.Layout.Override
            .SelectedRowAppearance.BackColor = Color.White
            .SelectedRowAppearance.ForeColor = Color.Blue
            .RowAppearance.BackColor = Color.White
            .SelectedCellAppearance.BackColor = Color.LemonChiffon
            .SelectedCellAppearance.ForeColor = Color.DarkBlue
            .RowPreviewAppearance.BorderColor = Color.White
            .RowSizing = UltraWinGrid.RowSizing.AutoFixed
            .ActiveRowAppearance.BackColor = Color.LemonChiffon
            .CellClickAction = CellClickAction.RowSelect
            .BorderStyleRow = UIElementBorderStyle.Dotted
        End With

        Me.TasksGrid.BackColor = Color.White

    End Sub
```

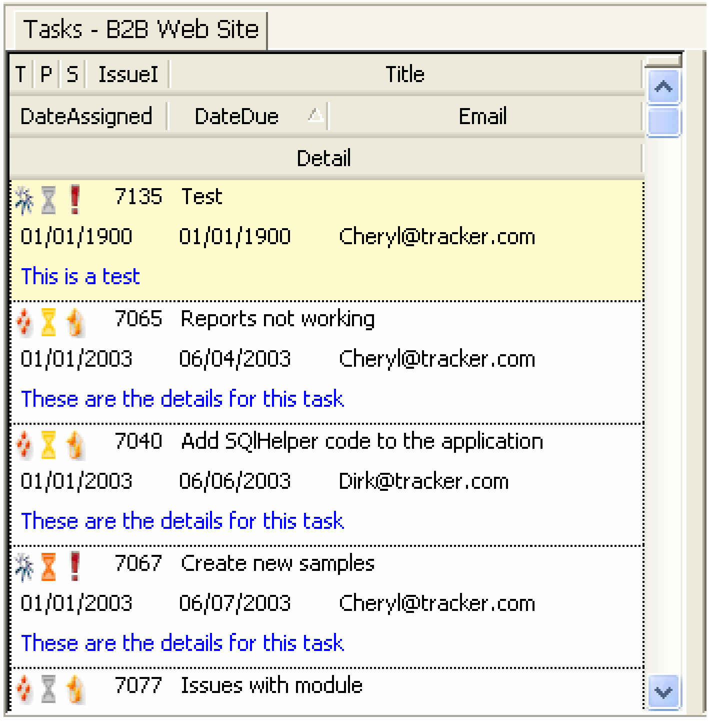
Notice the code to set the column specific properties, such as *MaxWidth* and *Header.Fixed*. These properties will make sure the columns never exceed a certain width, even if the user attempts to make them larger. The *Override* properties can be set using the properties window at design time, but I found it easier to write the code, which also helped me better understand the Presentation Layer Framework that the WinGrid is built upon. After this code was written, I wanted to display images in the grid based on properties of the data being displayed.

The three columns on the first level of the grid are *StatusID*, *TypeID* and *PriorityID*. If you remember back in Chapter 3, *Implementing the Tracker Database*, you saw how the *IssueStatus*, *IssueType* and *IssuePriority* tables contain descriptions for certain statuses and type of task in the Issues table. In order to display a meaningful image for each of these descriptions, an ImageList control on the form contains the various images that map to the column type. At runtime, during the *InitializeRow* event, the data in the cell is checked, and the correct image is loaded in the column. A portion of that code is listed here:

```
    Private Sub TasksGrid_InitializeRow(ByVal sender As System.Object, _
            ByVal e As Infragistics.Win.UltraWinGrid.InitializeRowEventArgs) _
            Handles TasksGrid.InitializeRow

        Select Case e.Row.Cells("TypeID").Value
            Case "1"
                e.Row.Cells("TypeID").Appearance.Image = 0
            Case "2"
                e.Row.Cells("TypeID").Appearance.Image = 1
            Case "3"
                e.Row.Cells("TypeID").Appearance.Image = 2
        End Select
```

If you recall back in chapter 3 when discussing the tables, I mentioned that it would be smarter to put the image index in the database, and not hard-code it in the *InitializeRow* event. It is a catch 22 situation, but I think an improvement in this code would be to dynamically load the image based on information stored in the table.

That is all there was to creating the very cool looking TasksGrid, which at runtime, looks like figure 9.5.



Tasks - B2B Web Site

T	P	S	IssueI	Title

DateAssigned	DateDue	Email

Detail

7135 Test
01/01/1900 01/01/1900 Cheryl@tracker.com
This is a test

7065 Reports not working
01/01/2003 06/04/2003 Cheryl@tracker.com
These are the details for this task

7040 Add SQlHelper code to the application
01/01/2003 06/06/2003 Dirk@tracker.com
These are the details for this task

7067 Create new samples
01/01/2003 06/07/2003 Cheryl@tracker.com
These are the details for this task

7077 Issues with module

Figure 9.5: TasksGrid at Runtime.

Creating the HTML Output for the Selected Task in the WinGrid

To create the HTML page that is displayed when a task row is selected in the grid, the *IssueID* cell value is retrieved, and a method is called to create the HTML page. To retrieve the *IssueID* from a cell in a WinGrid, you can use code that looks like the following:

```
Private Sub TasksGrid_AfterRowActivate(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles TasksGrid.AfterRowActivate

    Dim oRow As Infragistics.Win.UltraWinGrid.UltraGridRow
    oRow = TasksGrid.ActiveRow
    Try
        If oRow Is Nothing Then oRow = _
            Me.TasksGrid.GetRow _
            (Infragistics.Win.UltraWinGrid.ChildRow.First)

        If Not oRow.Cells Is Nothing Then
            _issueID = oRow.Cells("IssueID").Text
            _workspaceID = oRow.Cells("WorkspaceID").Text
```

To create the HTML, the *CreateIssuePage* method is called. This method, listed below, retrieves the current task from the Issues table in the database. It takes the *IssuesInfo()* type and serializes it to disk as an XML file. This XML file is then married with the Issues.xsl file using the *Transform* method of the *XslTransform* class. This transformation takes the XML and XSL and creates and HTML file. The last step is the simply load the HTML file into the Web Browser control using the Navigate method. I personally think this is pretty slick.

```
Function CreateIssuePage(ByVal issueID As Integer)
    ' This code created the HTML page displayed in the
    ' browser window

    ' Get the issue detail
    Dim i As Tracker.Info.IssuesInfo()
    i = _bll.GetIssuesByType(Tracker.Info.IssuesTypeList.ID, issueID)

    ' Create the writer and serializer to save to disk
    Dim sw As StreamWriter = New StreamWriter(Application.StartupPath & "\Issue.xml")
    Dim ser As XmlSerializer = New XmlSerializer(i.GetType())
    ser.Serialize(sw, i)
    sw.Close()

    Try
        ' Transform the XML
        Dim xpd As XPathDocument = _
            New XPathDocument(Application.StartupPath & "\Issue.xml")

        Dim xslTrans As XslTransform = New XslTransform
        Dim xsltArgList As New XsltArgumentList
        Dim xtw As XmlTextWriter = _
            New XmlTextWriter(Application.StartupPath & "\Issues.htm", Nothing)

        xslTrans.Load(Application.StartupPath & "\Issues.xsl")
        Dim parser As New DateParser
        xsltArgList.AddExtensionObject("urn:DateParser", parser)
        ' This code has the obsolete attribute, will work with 1.0 and 1.1 FCL
        xslTrans.Transform(xpd, xsltArgList, xtw)
        xtw.Close()
    Catch ex As Exception
        MessageBox.Show(Exceptions.HandleError(ex))
    End Try
    ' Load the newly created HTML file into the browser control
    wb.Navigate("file://" & Application.StartupPath & "/Issues.htm")
    End Function
```

One of the problems I had was displaying the date fields in a readable format. In the XML file, they look like this:

```
<DateAssigned>1900-01-01T00:00:00.0000000-05:00</DateAssigned>
<DateDue>1900-01-01T00:00:00.0000000-05:00</DateDue>
<DateStarted>1900-01-01T00:00:00.0000000-05:00</DateStarted>
<DateCompleted>1900-01-01T00:00:00.0000000-05:00</DateCompleted>
<DateAdded>1900-01-01T00:00:00.0000000-05:00</DateAdded>
```

In order for the dates to display correctly when transformed into HTML, an *ExtensionObject* is used with a custom class in the application to format the date.

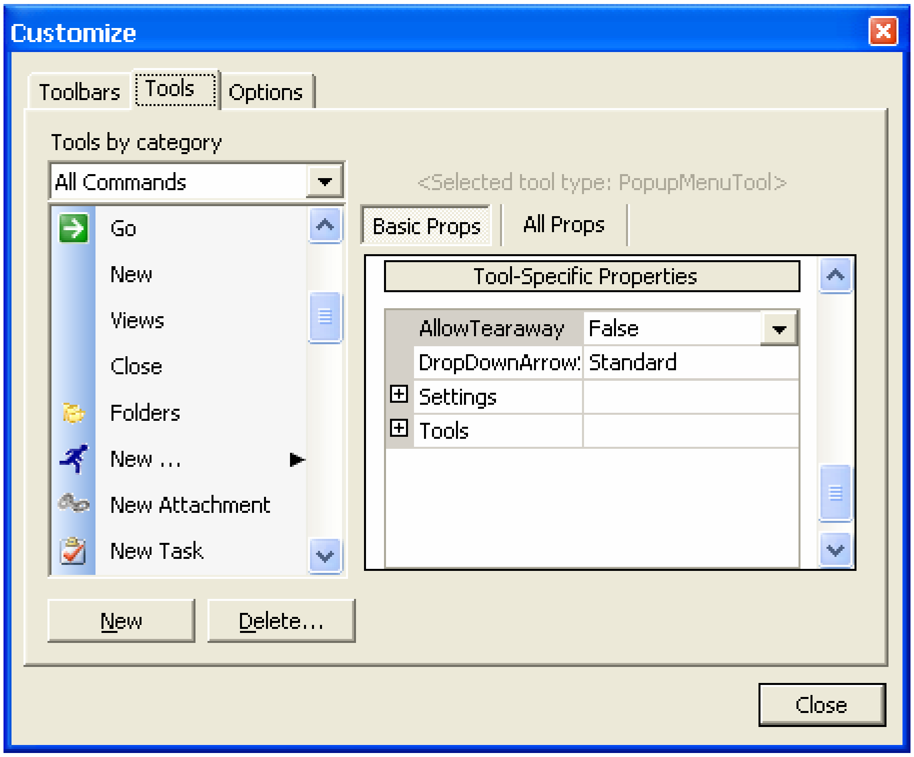
```
xsltArgList.AddExtensionObject("urn:DateParser", parser)
```

This class simply formats the date during the XSLT transformation. I need to thank my good friend and fellow INETA Academic Committee member Jeff Julian for this code. You can check out more of Jeff's code wizardry at http://geekswithblogs.com.

```
Public Class DateParser
    Public Function ParseDate(ByVal inDate As String) As String
        Return (DateTime.Parse(inDate).ToShortDateString())
    End Function
End Class
```

Implementing Toolbar Functionality

The ToolBarsManager tool adds an extremely professional look to your application. Using the ToolBarsManager tool is also extremely simple. Once you add a ToolBarsManager to your form, it is just a matter of right-clicking on the UI element and selecting the Customize option from the context menu to start adding tools to the toolbar. Figure 9.6 demonstrates the Customize dialog for the ToolBarsManager.



Tools by category: All Commands

AllowTearaway	False
DropDownArrow:	Standard

Figure 9.6: Customizing the Tools on the ToolBarsManager UI element.

Once you add tools to the toolbar, you write code that responds to the *ToolClick* event handler, and check the *Key* property of the Tool that was clicked to fire the correct code. The following snippet is from the *ToolbarMain_ToolClick* event handler of the TasksMain form.

```
Private Sub ToolbarMain_ToolClick(ByVal sender As System.Object, _
    ByVal e As Infragistics.Win.UltraWinToolbars.ToolClickEventArgs) _
    Handles ToolbarMain.ToolClick

    Select Case e.Tool.Key

        Case "GroupBy"
            Select Case Me.TasksGrid.DisplayLayout.ViewStyleBand
                Case ViewStyleBand.OutlookGroupBy
                    Me.TasksGrid.DisplayLayout.ViewStyleBand _
                            = ViewStyleBand.Vertical
                Case Else
                    Me.TasksGrid.DisplayLayout.ViewStyleBand _
                        = ViewStyleBand.OutlookGroupBy
            End Select

        Case "Export"
            With sfd
```

```
                .Title = "Enter name of file to save"
                .Filter = "*.xls"
                .ShowDialog()
                If .FileName <> "" Then
                    Me.ExportExcel.Export(Me.TasksGrid, .FileName)
                    MessageBox.Show("File exported to " & .FileName, _
                        App.MessageCaption, MessageBoxButtons.OK, _
                        MessageBoxIcon.Information)
                End If
            End With
```

The ToolBarsManager also fully supports merging the tools on the toolbar with the toolbar on the parent form of the MDI child. If a corresponding tool with the exact Key property does not exist on the form you are merging with, the ToolBarsManager on the parent form will add the tools to its ToolBarsManager. You can set the *MergeOrder* property to specify the order in which the tools should display, giving you complete control over how the toolbars look at all times.

One nice bit of code in the ToolbarMain on the TasksMain form is the context menu for the TasksGrid. This menu displays filtering options for the grid using the *ColumnFilters* property of the Row object. This allows the grid to display certain data based upon the filter criteria without having to retrieve it from the data source. The context menu shown in figure 9.7 is part of the ToolbarMain, and pops up in the right-click on the grid by setting the *ContextMenuUltra* property on ToolbarMain user interface element in the Properties window for the TasksGrid.

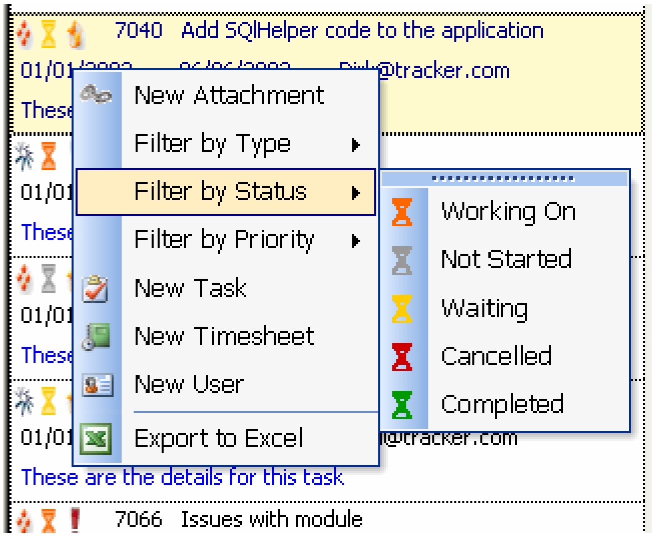


Figure 9.7: TasksGrid Context Menu.

When an item is selected, the grid is filtered based on information stored in the Tag property of the Tool on the ToolBarsManager. The following code shows the implementation of the *ColumnFilter* in a WinGrid UI element.

```
            Case "Bug", "Tasks", "Suggestions", "Requests"
                With Me.TasksGrid
                    .Rows.ColumnFilters.ClearAllFilters()
                    Dim strFilter = e.Tool.SharedProps.Tag & "*"
                    .Rows.ColumnFilters("TypeID").FilterConditions. _
                        Add(Infragistics.Win.UltraWinGrid. _
                        FilterComparisionOperator.Like, strFilter)
                End With

            Case "Working On", "Not Started", _
                    "Waiting", "Cancelled", "Completed"

                With Me.TasksGrid
                    .Rows.ColumnFilters.ClearAllFilters()
                    Dim strFilter = e.Tool.SharedProps.Tag & "*"
                    .Rows.ColumnFilters("StatusID").FilterConditions. _
                    Add(FilterComparisionOperator.Like, strFilter)
                End With

            Case "Critical", "High", "Medium", "Low"

                With Me.TasksGrid
                    .Rows.ColumnFilters.ClearAllFilters()
                    Dim strFilter = e.Tool.SharedProps.Tag & "*"
                    .Rows.ColumnFilters("PriorityID").FilterConditions. _
                     Add(FilterComparisionOperator.Like, strFilter)
                End With

            Case "View All Tasks"
                With Me.TasksGrid
                    .Rows.ColumnFilters.ClearAllFilters()
                End With
```

As you can see, this is a very powerful feature that allows total control over what data gets displayed in the WinGrid.

The last item of importance is the dialog boxes which handle the adding and editing of single items from the various MDI children in the application.

Understanding the Dialog Forms

In each of the MDI children and the docked windows, if you double-click on a data item, a modal dialog will pop up with information that you can edit for that specific data item. For example, if you double-click on a user in the Hot Contacts docked window, the User Details form shown in figure 9.8 will pop up filled with that users data.

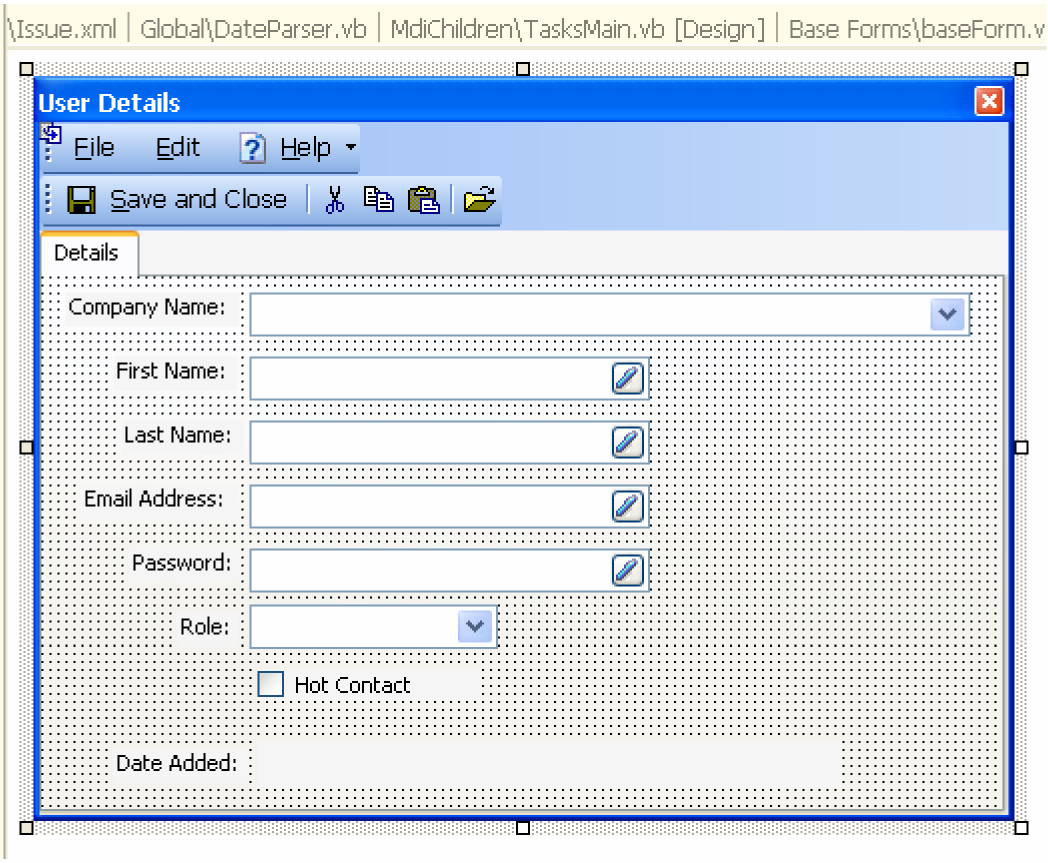


Figure 9.8: User Details form.

In the Tracker project, each of the forms in the Inherited Forms folder is based on the baseForm.vb class in the Base Forms folder. Since I knew that each form would need a main menu and a tab element, I did not want to set properties and colors on each form that I needed to add. By simply inheriting from the baseForm.vb, I had the basic look and feel of the form, and I just needed to add the controls for the form that corresponded to the data that needed to be edited. The base form is shown in figure 9.9.

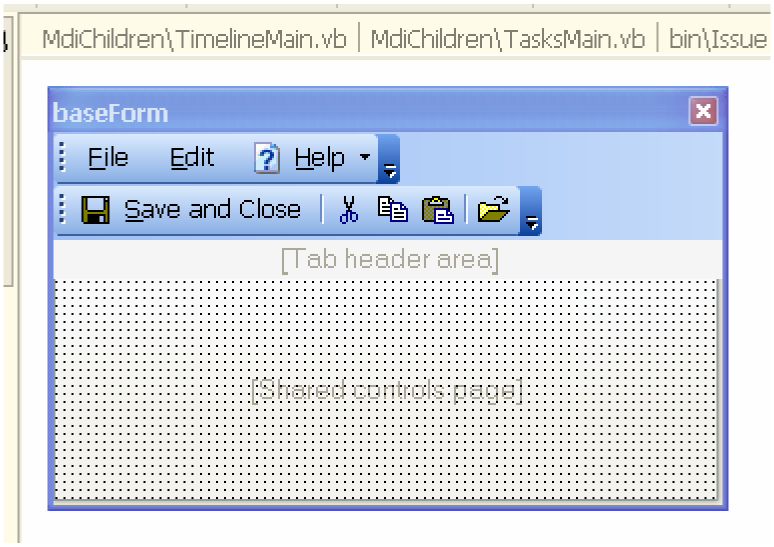


Figure 9.9: Base Form for all Dialogs in Tracker.

The key to making this customizable once it is inherited is to change the *Modifiers* property on the controls that can be changed by the newly created form to *Public*. This way, you can alter the look and feel of the ToolBarsManager element and the .NET tab user interface element on the sub-classed form.

Summary for Today and What is Next

This chapter covered some of the cooler features of the Tracker application, mostly the highly extensible and customizable WinGrid UI element. Using the designers of the WinGrid and some code, you can take ordinary data and display it in extraordinary ways using the WinGrid. You also learned how the ToolBarsManager works, and how the dialog boxes are implemented by using the inheritance features of the .NET Framework. In the next chapter, we are going to discuss some enhancements to Tracker, and what I would like to do for Tracker version 2.0.

10

Chapter 10: Where to go from Here

If you have gotten this far in the e-Book, you are probably swimming with ideas on how to write applications using the Infragistics tools, WS-Security, the Application Blocks for .NET and Visual Basic .NET. In this chapter, we'll look at a few items in Tracker that can be improved upon in code, and how the UI can be improved.

How Tracker can be Improved

Like any version .5 or 1.0, there are lots of holes. Tracker is no different. I will outline what I think are issues with Tracker (this of course does not include any bugs that you might find!).

Offline Conflict Resolution

Though I do not think the "disconnected" world is actually a reality just yet, it is not very hard to implement an offline application using the .NET Framework. The Tracker database and DAL are ready for offline data access by using the *TimeStamp* data type in SQL Server for each row in each table to handle any data conflicts that may arise if a user is offline for an extended period of time. Adding a user interface to Tracker to handle those conflicts would not be that difficult.

Removing all Modal Dialogs

I would like to change the modal dialogs for adding and editing data to non-modal forms. I hate modal forms, but after a while I was committed to using them. This is very simple to change, I would just need to make sure that each form is only looking at data within the form itself, and not at any stray global variable I may have set.

Better Menus

There should be a right-click everywhere in Tracker to do anything. At the moment, there are only a few right-click context menus. This would be accomplished by going through the ToolBarsManager on each form, and determining what needs to be added, and where the code should go.

Better Administration

I really cannot say better administration, since there is no administration. There needs to be forms added that handle all of the various lists and supporting tables in the database. There also needs to be a way for an administrator to manage the developers on his team.

Better Reports

Thanks to the built in printing functionality of the WinGrid, and the immensely robust WinChart charting capabilities, there are some reports in the Tracker application. I think there should be more though, and it is not too hard to add reports, so this is not a big job.

Track History for each Task

The History table in the Tracker database is not used for anything, so adding the functionality to track changes on a per item basis in Tracker would be immensely useful.

Handle Attachments better

The path for the attachments is hard coded in the data. This is not very useful if the path every changes.

Add more Field to the Users Table

I am not sure why, but it seems I forgot to add phone, fax and cell phone fields in the Users table. The best idea would be to add a PhoneNumbers table, and just let a user have an unlimited amount of contact numbers.

Where to go for Tracker Help and Updates

For updates and assistance on the Tracker application, please continue to visit the Infragistics DevCenter at:

http://DevCenter.Infragistics.com/RefApps/Tracker/Tracker.aspx