CHAPTER 13

Application Design and Implementation

The contents of the book thus far have taught you the individual techniques of database access. This chapter aims to help put all you've learned into a more real-world context. You may now know how the different objects in a data provider work together and are put to use, and you may have learned how to produce a page that will do the tasks you want it to perform, but taking that next step forward and putting together a whole data-driven Web site requires an extra set of skills.

Let's start with what you do know and then think bigger. As you develop more complex data-driven pages, you start to section database functionality into functions, so that they can be called time and time again, rather than written out in full. This is good programming practice that saves you time and space, even on a single page. If you take the time to design your Web site well, you can make such common functions available to every page rather than just individual ones—more time, space, and resources saved. In fact, the benefits are exponential. Indeed, a well-designed and well-implemented application is easier to maintain, test, debug, and extend than one grown a page at a time.

In this chapter, you'll look at some of the concepts in the *development life cycle* of a data-driven Web site. The point here is to give you an idea of the issues to be aware of and the questions to ask yourself as you build your Web site, rather than tell you exactly what to do and when.

This chapter covers the following topics:

- An overview of the Web site development life cycle
- Client requirements and Web site development tools
- Database and application design
- Best practices for implementing code
- Unit testing and performance measurements
- Common Web site maintenance tasks

This chapter is intended as a pause for thought before you begin building your own Web sites; it contains no actual code or examples. Instead, it introduces many concepts and refers you to books and Web sites where you can find more details and examples. Think of this not as the *Encyclopaedia Galactica* on the subject but, rather, as the *Hitchhiker's Guide*—useful, common information only. So, with the mandatory warning—don't panic!—let's begin.

The Software Life Cycle

From initial ideas to postrelease, the main stages of software development are as follows:

Requirements gathering and analysis: This stage involves defining what functionality a Web site will have, and how people and machines will interact with the Web site. You also need to develop the tests, or metrics, for measuring the performance of your Web site.

Design: This is when you design a namespace and class structure for the Web site that's clean, flexible, and easily extendable. You'll also design an equally efficient and extendable database structure (building on the techniques you learned in Chapter 2).

Implementation: This stage involves writing code over the class structure, taking into account the good practices you've learned thus far in the book.

Testing and debugging: You'll need to test each module to see whether it passes the benchmarks you laid out for it in the analysis phase, and then track down those annoying bugs.

Maintenance: Once the Web site has gone live, that isn't the end of the story. All Web sites need to be maintained. Bugs may need to be fixed, and there will always be areas where things aren't quite right. Additionally, unless you're extremely lucky, clients will request new features after the Web site is live.

Several different life cycle methodologies are available, and there isn't any "right" way to do things. Every project has its own unique qualities.

Analysis

It would be unwise to jump straight into coding a Web site without taking some time first to consider what the Web site needs to do. If a Web site doesn't perform to a client's standards, others may think again about contracting you to build something for them, and you certainly won't get any further business from the original client either. Even if you're working on something for yourself—a portfolio piece, if you will—you want to code it well if it will be on display for others to see; this is doubly so if it's an open source project where the standard of your coding will be judged.

Client Requirements

Your first job will be to work on a list of requirements for the Web site. If this is a contracted job, you need to sit down with your clients to draft this. But what kind of requirements are you after? Well, you can broadly categorize them as described in the following sections.

Contents and Functionality

The Web site contents and functionality requirements are concerned with the main aims for the Web site, the intended users, and the tasks it should be performing. Specifically, you should consider the following questions:

- What kind of Web site is it? Is it a personal site for a friend, a reference site for a business, an e-commerce site for a retail company, or a showy site for a rock band?
- Is the Web site envisaged as an intranet site, an extranet site, or a public Web site on the Internet?
- How many pages will the Web site include? What will they do? What information is being provided in these pages to the users? Once you understand the pages that are required, you should be able to draw a site map that shows how all of the pages fit together.
- What information is being provided to the Web site's owners about the users? Will the user habits and logins need to be tracked?

While the clients are trying to describe how they would like the Web site to run, your goals are to try to identify the items and events that could be modeled in a database and to pin the clients down on the distinct functionality of the Web site that you can translate into a class structure. Now, this latter point won't actually happen because, unless the Web site is remarkably trivial, clients will continue to change their minds, adding and removing features as you try to make progress. However, the core functionality of the Web site should be possible to pin down, and you should be able to produce a site map showing how the different pages of the site fit together.

User Roles and Access

User roles and access requirements involve getting to know more about who will use the Web site and how they will use it. Specifically, you should consider the following questions:

- Will the Web site interface with any other application? How will this interaction be achieved, and to what other applications will it be available?
- Who is going to be using the Web site? Will the user base be confined to a single company if it's an intranet site or a known set of users (if it's an extranet), or can anyone access it (if it's on the Internet)?
- Of those users, can you distinguish the roles they may have? For example, in a forum Web site, guests may only read posts, members may write posts, and moderators may approve and reject posts.
- What are the use cases? Can you determine how people in various roles will try to perform a task and how the Web site will behave in response? For example, in a forum Web site, how will a member look for a post on a topic and reply to it? What happens if a guest tries the same thing?

Use cases can be tricky to write, but they become invaluable when it comes to coding pages that are applicable to them. You need to worry as much about what the page does as how to implement it. Use cases also provide a useful mechanism for double-checking against the

site map and the Web site's contents list that the client provides. Can the pages the clients want to see actually perform the tasks the use cases have demarcated?

■ **Note** Use cases are typically written in English and then more formally in a graphical notation known as Unified Modeling Language (UML). Read more about UML in *Fast Track UML 2.0* by Kendall Scott (1-59059-320-0; Apress, 2004).

Security issues also need to be considered here. Use cases should include when users without sufficient permissions for a task are denied access by the Web site. You also want to establish how the different roles for a Web site map onto its different users and the type of user authentication strategy you may use. If you're working on an intranet site, Windows authentication may be fine. If you're working on an Internet site, Forms-based authentication or even Microsoft Passport-based authentication may be more appropriate.

Available Resources and Performance Targets

You need to ascertain what resources the Web site will have available to it and whether those resources will be able to perform as the clients would have them do. Specifically, you need to consider the following:

- Who will be hosting the Web site and where? Will the developers get physical access to the servers (hopefully not, as developers normally can't stop fiddling)? What specification are they using? Which database is running? What is the connection speed?
- What's in the budget for the day-to-day running of the Web site?
- Will this Web site use proprietary or open source software (besides ASP.NET)?
- Roughly how many users are expected to use the Web site at any one time? How quickly does the client expect the number of users to rise, and how fast does the client think the amount of data to be stored and retrieved in the database will increase?

The majority of clients will already have a hosting solution in place and presumably ASP.NET, too.

■ **Note** If your clients are already using Java or PHP, now may be a good time to go and get another book, say on Java database programming, and put this one down. Either that, or start extolling the virtues of ASP.NET.

The key in this part of the requirements analysis is to establish the kind of performance and scalability targets your clients want to achieve and whether their hosting solution will accommodate that. In concert with this, you also need to devise the tests, or metrics, that will allow you to measure whether your code has attained those standards. For example, if you're

working on a project with a SQL Server database containing 200,000 records that will need to be referenced and may be updated every day between 10 p.m. and 6 a.m, an obvious benchmark is the speed with which this database scan can be done.

If your clients are relying on you to provide a hosting solution as well as the Web site itself, the onus is on you to work with their budget and their performance targets to find the solution that best suits them. Typically, you need to establish a compromise between the price you pay for a database, the size of the connection you can use, the level of support provided for the servers if they're located off site, and so on. (You'll find a quick guide to choosing a database in the "The Right Data Source" section, coming up shortly.) If the client's budget stretches to it, you could investigate using and extending a commercially available piece of software as the backbone of the Web site.

Future Needs

Understanding how the client may want to extend the Web site in the future will have an influence on how you design the Web site, and you need to answer questions regarding how new content and functionality are to be added, as well as how maintenance is to be performed after the initial delivery. Specifically, you should consider the following questions:

- Is the Web site a one-off, or will it be refactored as a different Web site with the same purpose but different content, as is the case with `http://www.dotnetjunkies.com` and `http://www.sqljunkies.com`?
- How will the Web site continue to grow? Will it be a case of existing functionality being refined, or will completely new aspects of the Web site be added?
- How will content and data be added to the Web site and maintained? Will a user add it through a Web page or with a Windows application? Will data be updated automatically from system to system (for example, as one application pulls current stock prices from Wall Street)?
- How will Web site upgrades be applied? Will they be applied from a CD directly on the server? Or will upgrades be done online—perhaps via FTP, via some source versioning control system, or even via a set of Web services?

You should have resolved all the bugs you can find in the Web site before you release it to the client, but that won't mean there won't be more. Consider the case of Microsoft Windows. That's been in continuous development for more than ten years, and you can still find bugs all over the place. Maintainability isn't just about considering how the client will keep the Web site and the data it consumes current. It's also about how you can make bug fixes and small alterations to the Web site with the least interference to its uptime.

Maintainability is also about leaving yourself with the path of least resistance when it comes to upgrading a Web site with new features or reapplying the site design to some new purpose. The key is to design an open class structure so that new functionality can be "plugged into" the Web site without disrupting anything else. It's also vital to make sure the code is well documented and commented so that other developers who may work on the site later can see how it works and write their upgrades in a way to blend with your code. You may also be able to reuse code you've already written for other projects or use code online.

Last, but not least, it may not be a bad idea here to profile the clients yourself and see if there's anything they've missed or may want to add to the Web site in the second version. Forewarned is forearmed, as they say. And don't be afraid to ask clients for clarifications to their requirements list, especially when they revise them in the middle of the whole process. If they can bother you, you can bother them—especially if it means you go one way or another with the Web site's design.

We'll take a closer look at some maintenance issues in the "Maintenance" section later in this chapter.

The Right Tools

With a set of requirements from a client, you can start deciding on the basic building blocks for the Web site. You've already decided on ASP.NET as the framework of choice and effectively standardized on Internet Information Services (IIS), but what about the database containing the Web site's data, and even the tools you'll use to create the Web site? If the client has predetermined the database, that's fine; it leaves you with just your development tools to pick.

The Right Data Source

Although it may sound strange to ask, but does the Web site actually need to use a relational database as its data source? Would it suffice to use an alternative data source such as a set of XML files, Excel spreadsheets, `.csv` files, or perhaps an alternative type of database, XML-based or object-oriented?

Looking on such open source Web sites as SourceForge (`http://www.sourceforge.net`) or even GotDotNet workspaces (`http://workspaces.gotdotnet.com`), it's not too difficult to find applications with the same goal but different approaches to data storage. Take the example of two blogging engines, dasBlog and .Text. Both are designed to make blogging easy for users, but dasBlog uses XML files to store recent entries, and .Text uses SQL Server. dasBlog is more portable and easier to host. .Text can provide blogging for many users once the blog is set up. dasBlog works for one user per installation.

As this example implies, all the decisions you make are trade-offs—one requirement against another. You're inevitably forced to make financial decisions based on your time and the client's budget, and to balance the issues of performance, scalability, maintainability, and availability based on the requirements and the tools you're using. Your choice of database, if you have one, reflects that balance. The following are some of the factors that may influence your decision:

Price: SQL Server 2005 Express Edition is free to download, Access is part of Microsoft Office, and SQL Server 2005 varies in price from expensive to very expensive depending on which version you want to use. MySQL 5.0 is free to download, but you'll need to pay for phone support, if you need it.

Performance: Access (`*.mdb`) files are easy to create and use but are slow compared to actual database servers such as SQL Server 2005 and MySQL 5.0. MySQL's emphasis is on fast data retrieval, but SQL Server beats it for performance at enterprise levels.

Maintenance: Can the database server be backed up easily? Does it support automatic failover to a different database server if the main database server fails? A server failure can happen any time, and you need to be prepared for the worst-case scenario.

Data provider: Where possible, use a database with its own data provider rather than using the generic `OleDb` or `Odbc` data providers supplied by Microsoft. For example, although we've used the `Odbc` data provider to connect to MySQL 5.0 in the majority of the examples in this book, that was only for demonstration purposes. If you're working with MySQL 5.0, you should use its native data provider, Connector/NET, which you can download from `http://dev.mysql.com/downloads/connector/net/1.0.html`.

Ease of use: All databases work fine with ASP.NET, but some are easier to administer than others. For example, you've used both SQL Server Management Studio and MySQL Query Browser in this book. While these tools are certainly an improvement over the previously available tools, they still have their limits. Similarly, how easy is it to migrate a database across to a new installation or even a new version of the database?

Functionality: Relational databases all have the same core functionality because they're all built according to the 12 rules of relational database design originated back in the early 1970s. What you need to look for are the extensions to the core that the database vendor has decided to add that may help you in building your Web site. Does it handle XML, transactions, and stored procedures? What extensions to the SQL standard does it support? How does its error handling work? What about user roles, security, and backup tools? Establish what functionality you need from the database and shop around for the one that does the best job providing it.

Support: Last, but not least, how well is this database supported by the vendor if something goes wrong and you have no choice but to ask for help? Newsgroups and FAQs can be helpful, but phoning technical support is sometimes the only solution.

The Right Development Tools

You've used the free (for the time being) Visual Web Developer 2005 Express Edition for the ASP.NET development work in this book. It's a perfectly acceptable tool, works particularly well with C# and VB .NET, and is a subset of the functionality afforded by Visual Studio 2005.

If you're new to programming and aren't used to any development tool yet, stick with Visual Web Developer for the time being and download evaluation copies of a few others. Although Visual Studio 2005 has by far the best support for .NET development of all these tools—with features such as the class browser, integrated data explorer, and IntelliSense—it's still a very expensive piece of software.

Design

As Web developers, you may tend to regard Web sites as groups of pages. However, it's far wiser to approach them as traditional applications. Bearing this in mind, it's safe to say that Web sites are subject to the following usual tenets of software engineering:

- A software application—whether it's Web-based, desktop-based, or even designed for a mobile platform—is implemented by dividing the tasks it must perform into smaller and smaller tasks until they can't be reduced any further. These atomic tasks are represented as a line of code in the application, and where atomic tasks are often called in the same sequence, they're grouped into methods.

- A software application of any size is subject to bugs and the client's desire to upgrade it. Dividing the functionality of an application into classes and namespaces that can be debugged and upgraded individually lessens the downtime for the application as a whole. It also has the added bonus that, because several people can work on individual classes and namespaces simultaneously as long as the public application programming interfaces (APIs) are as agreed, development time will be quicker.

The purpose of analyzing the requirements for an application is to identify the common tasks it will execute, how well they should execute, and the various ways in which you can segregate its functionality into manageable chunks—what you'll implement as namespaces and classes. In the design phase, your goal is to translate those requirements into a class and namespace structure and a supporting database design that will make your Web site work efficiently and be easy to maintain. You can then use the metrics you created for the Web site to check that the various chunks of the Web site work as well as required.

You should split this phase of development into two parts: designing the table structure for the database that underlies the Web site, and then designing its namespace and class structure.

Database Design

As with every other subject in this chapter, whole books have been written about database design. Many people make a very good living from just designing and administering databases. Here, I'll touch on the salient points.

Database design in general is all about efficiency. If you can correctly identify and model the various objects and events the Web site needs to keep details on, the Web site will have the following benefits:

- It will be easier to write.
- It will run faster because fewer queries need to be made to the database.
- It will be easier to maintain and upgrade because the SQL will be easier to work with.
- It will be easier to add new tables into the database if you need to do so.

The database design should be completed before any work starts on building the pages of the Web site. Discovering that the database hasn't been completed or is incorrect can cause problems if you've already started coding the Web site. It may even require you to throw away the pages that you've already written because they don't work with the changed database.

At this basic level, you're most concerned with modeling the Web site's data correctly. As you learned in Chapter 2, a lot of good database design is simply a result of paying attention to the details. However, it all bears repeating here.

Your core job at this stage is to correctly identify the individual types of objects and events you'll model in the database underlying your Web site. For example, in a blogging Web site, you need to model messages, users, and comments. Don't forget the following simple tips toward this end:

- Each table in the database should contain details about one object or event. Don't try to match up two objects that are roughly the same—for example, buildings and companies. The ploy may work at first, but it would require you to split them later when the Web site is up and running, which would mean you would need to take the Web site offline while this was being done.
- Pay attention to the names of the tables and columns as they're conceived. Using plurals and words such as *or* and *and* could indicate that a column needs to be split into its own table that has a one-to-many or many-to-many relationship with the original. Try to make every column and table name unique.

Of course, you have a more formal way to achieve a streamlined database design, called *normalization*.

Normalization

The basic rules of normalization have been around since 1972. They've been tweaked and added to since then, but the same goal remains: to improve the performance of a database by eliminating duplicate data and therefore the chance of any errors occurring when information is added, updated, or deleted. A normalized database will also make searching easier because it won't have to deal with redundant copies of data.

Consider the sample database we've used in this book. We chose to include the storage format for each Player as text inside each row in the Players database, but as you saw in Chapters 8 and 9, this made some aspects of working with this particular column quite troublesome. It's also a waste of time and a source of potential errors to enter the storage format by hand for each Player. Why not just enter the storage formats once, and then refer to which one of them the Player uses? This would also remove the possibility of a user misspelling a storage format—mistakes that make accurate searches trickier. By using normalization, you identify duplicate data that could be stored in separate tables.

Using normalization to improve the design of a database is a three-step process. The result of each step is known as a *normal form*. (There are extra steps beyond the first three, but those are rarely used in the real world.)

First Normal Form

The goal of the first normal form (1NF) is to eliminate repeating groups of data in a table. You create a separate table for each related set of data and identify each table with a primary key.

> *A relation is in 1NF if and only if all underlying simple domains contain only atomic values.*

In more practical terms, a database table is in 1NF only if all values in all columns can't be split any further into separate columns—in other words, they're *atomic*. Furthermore, atomic values shouldn't be repeated over various columns.

For example, Figure 13-1 shows the Player table as it was originally. The original Player table isn't in 1NF for a number of reasons, but the most obvious reason is that the PlayerManufacturer column contains two pieces of information that should be split into two separate columns: the name of the Player and its Manufacturer. The PlayerFormats column also contains more than one piece of information depending on how many Formats the Player supports.

Table Name: Player

PlayerID	PlayerManufacturer	PlayerCost	PlayerStorage	PlayerFormats
1	iPod Shuffle (Apple)	99.00	Solid State	wav, mp3, aac
more rows...				

Figure 13-1. *The Player table not in 1NF*

To obey the first half of the 1NF rule, the Player table should look like Figure 13-2.

Table Name: Player

PlayerID	PlayerName	ManufacturerName	PlayerCost	PlayerStorage	Format1	Format2	Format3
1	iPod Shuffle	Apple	99.00	Solid State	wav	mp3	aac
more rows...							

Figure 13-2. *Splitting atomic values*

The only problem is that this table still isn't in 1NF. The second half of the rule states that atomic values shouldn't be repeated over more than one column in the same table. The three columns Format1, Format2, and Format3 violate this rule. Figure 13-3 shows how to achieve 1NF by splitting this information into two tables.

Table Name: Player

PlayerID	PlayerName	ManufacturerName	PlayerCost	PlayerStorage
1	iPod Shuffle	Apple	99.00	Solid State
more rows...				

Table Name: Format

FormatID	PlayerID	PlayerName	FormatName
1	1	iPod Shuffle	wav
2	1	iPod Shuffle	mp3
3	1	iPod Shuffle	aac
more rows...			

Figure 13-3. *1NF achieved*

By creating the Format table, you now have a FormatName column containing individual Format names. Both tables are in 1NF.

Note how the tip for accurately naming columns can help in identifying whether a table is in 1NF. A pluralized name indicates that perhaps the column should be separated from the table. Any sign of an *or* or an *and* would indicate that a column may be better as two or more columns.

Second Normal Form

The second step in the normalization process, second normal form (2NF), seeks to identify relationships between entities in a database and to model them correctly.

> *A relation is in 2NF if and only if it's in 1NF and every nonkey attribute is fully dependent on the primary key.*

The first part of this definition is a given. You shouldn't be moving onto 2NF if you haven't made your database design 1NF already. The second part of the definition warrants attention, however. In plainer English, if a column isn't uniquely related to the primary key in a table, it should be elsewhere.

Let's take a look at the Player and Format tables, which are both in 1NF. The primary key for the Player table is the PlayerID, so you need to establish which of the values for the four other columns are linked directly to the PlayerID. The answer is that only the PlayerName and PlayerCost column are; both ManufacturerName and PlayerStorage aren't dependent on the Player having a certain PlayerID. As you've seen, many Players may have the same Manufacturer or Storage, so to satisfy 2NF, you'll have to split them into a different table. Neither column relies on the other either, so you'll need to create one table each for them, as shown in Figure 13-4.

Table Name: Player

PlayerID	PlayerName	ManufacturerID	PlayerCost	StorageID
1	iPod Shuffle	1	99.00	1
more rows...				

Table Name: Manufacturer

ManufacturerID	ManufacturerName
1	Apple
more rows...	

Table Name: Storage

StorageID	StorageName
1	Solid State
more rows...	

Figure 13-4. *Putting the Player table in 2NF*

■ **Note** The Player table that we've been using is not fully in 2NF. The PlayerStorage column was deliberately left incorrect to highlight the problems that you will have if the database is not correctly designed.

Like the Player table, the Format table isn't in 2NF. The FormatName column is fully dependent on the FormatID column, which is the primary key, but the PlayerID and PlayerName columns aren't; the wav Format will probably be supported by more than just the iPod Shuffle, for example. You need to introduce a third table to express the many-to-many relationship between Formats and Players, and thus satisfy 2NF, as shown in Figure 13-5.

Table Name: Format

FormatID	FormatName
1	wav
2	mp3
3	aac
more rows...	

Table Name: WhatPlaysWhatFormat

WPWFPlayerID	WPWFFormatID
1	1
1	2
1	3
more rows...	

Figure 13-5. *Putting the Format table in 2NF*

The Format table is now in 2NF because the FormatName is dependent on the FormatID column, and the WhatPlaysWhatFormat table is in 2NF because it has a compound key comprising both columns.

Third Normal Form

The third and final step, third normal form (3NF), looks specifically at tables with compound primary keys.

> *A relation is in 3NF if and only if it's in 2NF and every nonkey attribute is nontransitively dependent on the primary key.*

In plain English, this rule basically says that a column in a table must depend on all the elements in a primary key. So if a table has a simple primary key (that is, the primary key is only one column) and is in 2NF, it's also in 3NF. If a table has a compound primary key, however, you'll need to take a closer look.

For example, what if the WhatPlaysWhatFormat table had two more columns, as in the slightly contrived Figure 13-6?

Table Name: WhatPlaysWhatFormat

WPWFPlayerID	WPWFFormatID	ImplementationUrl	PlayerUrl
1	1	http://www.apple.com/ipod/wav	http://www.apple.com/ipod
1	2	http://www.apple.com/ipod/mp3	http://www.apple.com/ipod
1	3	http://www.apple.com/ipod/aac	http://www.apple.com/ipod
more rows...			

Figure 13-6. *Removing the PlayerUrl column will leave this table in 3NF.*

Both the ImplementationUrl and the PlayerUrl columns are dependent on the primary key in this table, so it's still in 2NF. But whereas the ImplementationUrl column depends on both WPWFPlayerID and WPWFFormatID columns for context, the PlayerUrl column relies only on the WPWFPlayerID column. This partial dependence is also referred to as *transitive*

dependence, which means the table isn't in 3NF. The PlayerUrl column should be moved to the Player table for the whole database to be in 3NF.

Normalizing a database will make it faster to sort and search through by reducing the chance of redundant data in the database. In particular, the database will need fewer indexes, which will improve the performance of `INSERT`, `UPDATE`, and `DELETE` queries.

However, normalization can increase the complexity of joins required to retrieve the data. This, in some cases, may hinder performance. Sometimes, databases are denormalized to reduce the complexity of joins and to get quicker query response times (one of the reasons why fourth and fifth normal forms aren't often used). As always, you need to strike a balance between flexibility and speed. This is just another juggling act you'll encounter as you implement the Web site.

Other Modeling Considerations

Normalization is a great tool for laying the groundwork in your database design, but just because your tables are all in 3NF doesn't mean you've modeled the data the best you can for the Web site. The following are a few more database design considerations.

Naming: The database server doesn't care how tables and columns are named as long as they're identified unambiguously in SQL queries. However, for your own benefit, you should try to name them all sensibly and uniquely, as you did in the sample database. You only have to look at the number of columns that could be called ID in the sample database to see that it's always a good idea to give them all unique names so you don't get confused as you write your code. Here, you used the simple method of prefixing the name column with the name of the table: PlayerID, ManufacturerID, and FormatID.

Foreign keys: You've identified the various tables in the database and the relationships they have between each other, but when you're setting up the foreign key constraints, you need to consider how a child table should be affected when a row in the parent table is updated or deleted. Are the database's defaults fine, or should the implications cascade down into the child table?

Data types: You've identified the various columns in each table of the database, but what type should they be and what length? Should ID columns be automatically generated integers or globally unique identifiers (GUIDs)? How long should each string be? Should the database store the image or just the image's file name?

Views: Will it be of any benefit to you to create distinct views over the database for querying? It does mean that you can keep your SQL queries simple, but at what cost? Each time a view is queried, the query to create that view must first be executed, so there are actually two queries being used here for the sake of one.

When you've considered these issues, you can start to determine how to speed up database access and secure the database against both the malevolent and the stupid.

Application Design

While design of any kind is somewhat in the eye of the beholder and subject to a person's own preference for doing things a certain way, it's good to know that database designs won't vary

that much among architects given the same list of requirements. The design of an application's class structure, on the other hand, can be much more divergent.

Not only must you marry functionality to pages, you must also see that the classes you design reflect the database design you've just achieved and obey the second software tenet. For those of you who missed it earlier, this means that to make the Web site as quick to implement, easy to debug and maintain, and straightforward to upgrade as possible, you need to divide the tasks of the Web site into clearly defined sections that you can then map to namespaces and classes. So, for example, you can split such easy-to-define sections of a Web site as forums, polls, shopping carts, product browsing, and so on. However, it isn't so obvious how to split potential namespaces into individual classes.

Fortunately, a great deal of practice and trial and error by a great many people have shown that you can think of a Web site (and indeed most applications) as split across several tiers.

Tier Definition

Splitting the functionality of the Web site into separate modules—polls, forums, carts, newsletters, and so on—means that you can add new modules to the Web site at any time and that individual modules can be shut down for maintenance without affecting the rest of the Web site. To make things easier, you can split the tasks each module performs into the following three tiers:

Data access tier: Code in the data access tier deals solely with sending commands to and retrieving data from the database, checking that the current user has permissions, and handling any database-related errors. If code in either of the two other tiers needs to interact with the database, it must do so through the data access tier.

Business rules tier: The business rules tier is where the majority of the work happens. Here, the Web site will determine how to react to a user's request, log the request for future reference, start the process of creating a new page by requesting information from the data access tier, and then interpret that new data into something a user will be able to understand. For example, when a user requests the home page of a Web site, the business rules tier may take care of such tasks as retrieving user preferences, determining what's new on the Web site since the user last visited, and telling the presentation tier (described next) how to react accordingly.

Presentation tier: Code in the presentation tier deals exclusively with the Web site's user experience and the generation of pages when requested. It will use the business rules and data access tiers to retrieve the content for a page, and then use its own code to assemble it on the page. No code from the business rules tier or data access tier should directly alter the user interface of the Web site.

Splitting a code module into tiers and then into classes within those tiers gives you great flexibility. Suppose you were writing an e-commerce cart module and were told to provide front ends for both Web and mobile users. You could write separate classes for the module's presentation tier: one for browsers and one for mobile devices. They would both interface with the business rules tier in the same way, but each would optimize the presentation of content for their respective device.

In the same way, you could write individual modules for the different ways to pay for the items in the shopping cart. To the users, this would be invisible, but to the owners of the e-commerce store, this would be invaluable. Should they choose to barter Visa transactions

with a different merchant bank, for example, you could take down and revise just the Visa class in the e-commerce module, leaving the cash, Mastercard, and other modules up and running. Compare this to a situation where all transactions are run through a single class or as a single monolithic piece of code. To update the Visa functions, you would need to take down the whole shopping cart.

Similarly, you could write new classes for working against a new database rather than rewriting the old ones.

As long as the public interface to each class remains the same (it may expand to include new public calls, too), so you don't have to worry about other classes calling functions that no longer exist, you could work with as many different databases, credit cards, banks, and interfaces as you like. Good application design gives you this kind of flexibility, and separating an application into these three tiers is the first step to achieving it.

■ **Note** Depending on how complex your Web site or a module for your Web site is, its code may split logically into only one or two tiers. Don't worry about this and go searching for the third tier. Three is just the standard number; there are always exceptions to the rule.

Structure Definition

Now that you understand how a Web site may be split into data, business rules, and presentation tiers, a question arises. Say you're building a community Web site and, from the site map, have identified a number of modules you'll eventually want to add to it, as shown in Figure 13-7. What translates into a namespace and the classes inside it?

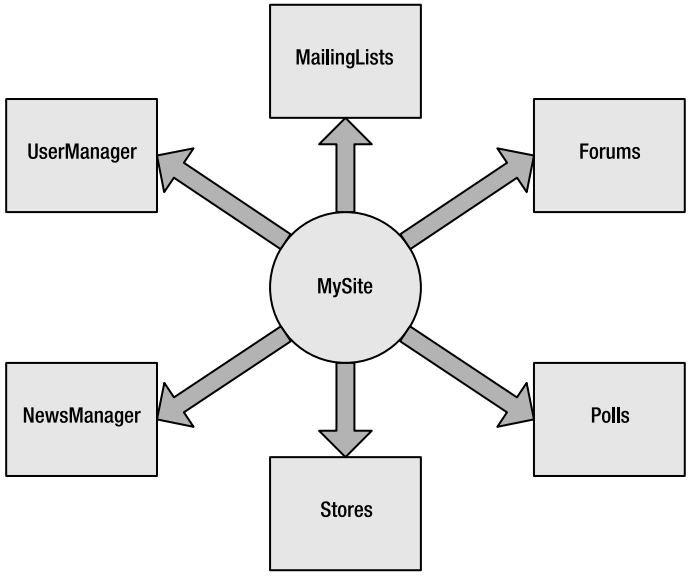

Figure 13-7. *A Web site and its modules, but how to divide it?*

Do you have three namespaces for each tier of the Web site and a class for each relevant section of forums, polls, and so on inside those namespaces? Or are the individual modules modeled better as namespaces with the tiers split inside them into classes? The latter is the best route. Taking the long view, if you design a uniform class structure and API for working with the core of the Web site for each, you get a nice plug-in API from which it's easy to add or remove functionality. If you keep the core of the Web site separate, you can come up with a namespace structure looking something like Figure 13-8.

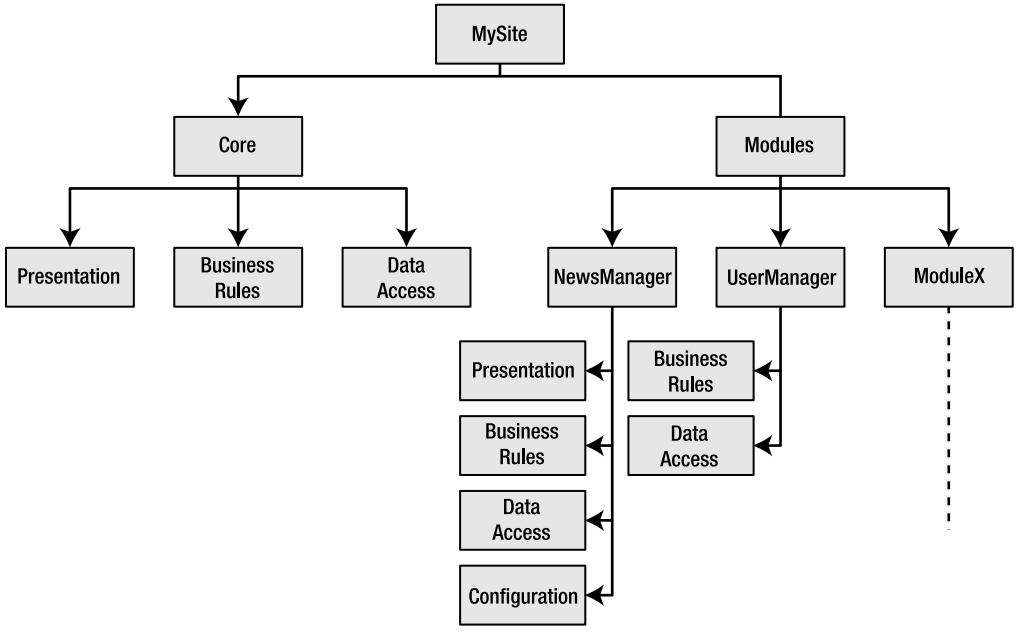

Figure 13-8. *A clean namespace structure for MySite*

Depending on how detailed the client requirements are and how complex the Web site is, you may want to encapsulate the three tiers within classes rather than namespaces. Note that you'll need to tailor the creation of namespaces to the functionality of the module, rather than arbitrarily creating presentation, business rules, and data access namespaces and attempting to wedge all functionality into those. In Figure 13-8, for example, the UserManager module won't return any on-screen information per se, so it won't need a presentation tier. Meanwhile, the NewsManager module acquires an additional namespace for configuration to keep track of news feeds being used. In the same way, a shopping cart may have a separate namespace for negotiating sales with merchant banks, with each bank being dealt with in a separate class.

With any luck, the requirements list you've already procured will give you plenty of clues as to what the classes within each namespace will encompass. If not, take the opportunity to go back to the client and ask for more details on any functionality that isn't clear.

Remember that some classes will reflect the objects and events you're modeling in the database, and some will simply be utility classes, which may not be immediately foreseeable. Work through the use cases for each module in turn. Understand which pieces of data they will require and the way in which the sequence of events within them will flow.

It would be nice to say that every enumeration and class will come forth in a Zen-like way from the pattern of the Web site, but they won't initially. Just don't get frustrated when this happens. You can't identify every single facet of the design until you've implemented and looked at it again in hindsight. That's why you prototype and postmortem your projects.

■ **Note** Like every subject in this chapter, good class design is a topic that warrants its own book. Good ones in this case are siblings: the *C# Class Design Handbook* by Richard Conway (1-59059-257-3; Apress, 2003) and the *Visual Basic .NET Class Design Handbook* by Damon Allison, Andy Olsen, and James Speer (1-59059-275-1; Apress, 2003).

Techniques such as using design patterns to drill down into individual classes also apply at this stage. However, even though these techniques are handy, a lot of this step comes down to your own experience and what you can learn from the experiences of other developers. Remember that however you choose to design (and implement) your Web site, the whole point of breaking it up into components is to give it a flexible, efficient structure that's easy to program and extend.

Implementation

Lest we forget that this book is aimed squarely at introducing you to developing pages using a database to supply the content, the following sections on implementing your design focus on the basic issues of writing code for working with the database, or issues related to the data tier. How you choose to present information on a page isn't the issue here, and neither are any business rules you may be adding.

Prototypes

It's a good idea to prototype a Web site to prove and refine an initial design. Although it won't help you identify and deal with every issue concerning elements that seemed reasonable on paper but don't translate across into code, it will catch many problems. It also has the added advantage of giving you something to show your clients and get their feedback on, which can't be a bad thing. Inevitably, on seeing even the first iteration of their commission, customers often have opinions and questions that may affect the final product, so it's worth building a prototype at least once to validate your design and pacify clients, if nothing else.

Exactly how much detail you include in a prototype is up to you, but from a data access point of view, you definitely want to validate the following pieces of the design:

- The Web site core
- The database design
- The various SQL queries to the database that are most likely to be made and the classes you'll create to wrap these queries

Security always seems to get put aside in favor of the noticeable functionality of the Web site, but try to prototype the user registration and login to make sure that each user role has the correct set of permissions for access to the database. You could even use the metrics you've designed previously to test the prototype and see how far away you are from your performance goals.

Stored Procedures

When you send a SQL query to a database, the database checks that its syntax is correct, making sure that the tables, columns, constraints, and so on that it references actually exist. Finally, it figures out how best to execute the query, and then does it. Depending on the database server, this whole process is repeated for each new query (SQL Server 2005 actually caches the execution plan for the SQL query to speed things, whereas MySQL 5.0 performs the same steps every time). Even with a small number of concurrent users, those three steps can quickly mount up and burden the database. Fortunately, there's a solution.

Most relational database management systems now offer the ability to create and save stored procedures. As you learned in Chapter 10, a stored procedure is a SQL query saved in the database server, so that rather than the database working through three steps, the stored procedure requires only one step (the execution of the stored procedure). It's not hard to see that using stored procedures can offer the promise of a significant performance increase. However, not everything works better as a stored procedure. There's a small overhead associated with actually retrieving and then executing the stored procedure. In some cases, simple SQL queries (for example, selecting data from a single table) may actually perform better when they come from the code directly. In general, though, stored procedures are a good thing to use if the database server supports them.

Stored procedures also give you an extra level of flexibility when it comes to retrieving data. When you get to the inevitable decision of whether to use a `DataSet` or DataReader to build your pages, you can also consider using a stored procedure's output parameters to retrieve single values straight into a variable, without the need for either a `DataSet` or DataReader.

Stored procedures also give you a more structured way to introduce error handling and transactions into a SQL query. By using SQL aggregate functions, you can use stored procedures to perform quite a few calculations that you may have thought only ASP.NET could do for you.

Code Issues

Designing a Web site well will help a project along, but it's making informed implementation decisions on the spot based on the design and your experience that really counts. The following are some of the most common data-related issues that will crop up:

DataReader vs. DataSet: By now, you know the advantages and disadvantages of both approaches. Browse to `http://msdn.microsoft.com/library/en-us/vbcon/html/vbconDecidingOnDataAccessStrategy.asp` for details on some additional issues; refer to Chapters 5, 6, 7, and 8 of this book; and make the choice.

SqlDataSource: As you saw in Chapter 9, you can accomplish quite complex editing using a `SqlDataSource`. However, the `SqlDataSource` doesn't fit into the tiered application model described earlier in this chapter. It connects to the database directly from the presentation tier, skipping the data access and business rules tiers completely. That doesn't mean you shouldn't use it, but you should consider its limitations before you do.

Data provider: It sounds obvious, but you need to consider the actual data provider you use in your code. For example, MySQL has a solid, stable ODBC driver, and you can take advantage of that using the `Odbc` data provider, which is also stable. However, is this the best solution? As noted earlier, MySQL has a native data provider that is a better choice.

Data modeling: If speed is of the essence, then the amount of data retrieved from a database is surely one of the key factors in keeping the speed up, so finding ways of packing information into smaller pieces is always handy. Reducing the maximum length of a column is always a risk, but take the case of an ISBN for a published book. This is a ten-digit string consisting of nine integers and a tenth-check digit that's either another integer or an X. The check digit is always calculated in the same way, so do you store the ISBN as a ten-character string that's held by 10 × 8 bits = 80 bits, or as a nine-digit integer that can be held comfortably in 32 bits, and create the check digit programmatically when required?

Security: It's easier to write code that does the job required and then add security measures after the fact, but coding securely is an art in itself; in fact, it's something you should always try to do. You've already seen some good practices in earlier chapters. Hide your database connection strings in `Web.config` to keep them from prying eyes. Try not to use the query string for values being sent to and from the database. Work with multiple database servers rather than just one. If you're using a DataReader, close it with `Close()` as quickly as possible.

Of course, you'll have to make many more trade-offs and decisions as you implement your design. But rather than tread in places that have been well covered by books devoted to the topic, let's move on to the next phase of the software life cycle: testing and debugging your Web site.

Testing and Debugging

Nothing is more annoying than installing a new application, starting to work in it, and seeing a dialog box pop up to inform you that an error has occurred and the application will now shut down. You've now lost all the work you were doing and have given yourself a nasty injury after kicking the desk in frustration. If the application had been debugged more thoroughly, that may not have happened.

This phase of the software life cycle is certainly as crucial as the others, yet it's usually written about less because it's almost impossible to describe how to debug specific errors; further, writing for the generic case isn't very helpful. However, without going into the debugging process itself, you can try to factor a handful of useful techniques into your development process.

■ **Note** For a complete guide to .NET debugging strategies, see *Debugging Strategies for .NET Developers* by Darin Dillon (1-59059-059-7; Apress, 2003) or (if you can find it) *Visual Basic .NET Debugging Handbook* by Jan Narkiewicz and Thiru Thangarathinam (1-86100-729-9; Wrox Press, 2002). The latter is out of print but packed with gems of information.

Unit Testing

As mentioned earlier, one of the benefits of writing code as individual modules is that you can write them in parallel. But it goes further than that. You can debug and test them individually as well.

Back in the analysis phase of the software life cycle, you devised a set of benchmarks and use cases that could measure the performance of your Web site as it was built. By isolating the relevant user scenarios and benchmarks, and writing some code (a *test harness*) that reflects the metrics you designed to test the Web site against those benchmarks, you can prove the utility of this module before you move on to the next. This process is known as *unit testing*.

One of the more frustrating aspects of this process is that as requirements change and targets are realigned, modules that you've already written and tested will need to be altered and retested. Indeed, this cycle of rebuilding and retesting is quite short, so a few utilities now allow you to automate the unit testing process. The best of these is NUnit (`http://www.nunit.org`).

NUnit is the standard unit testing framework for .NET applications and was itself written in C#. By itself, NUnit comes with a choice of a rudimentary command-line interface or a rudimentary graphical interface, which both do the job, albeit not in a particularly attractive way. NUnit on its own is great for testing business rules and data access code, but it needs help to test Web pages because they can't be "run" inside the NUnit framework. In this case, you need to use NUnitASP (`http://nunitasp.sourceforge.net/download.html`), which essentially hooks into ASP.NET and gives NUnit a view of the intrinsic objects (Context, Response, Request, and so on) to use. You can find a great introduction to using NUnitASP at `http://www.theserverside.net/articles/showarticle.tss?id=TestingASP`.

■ **Note** NUnit is a free, stand-alone application that's great if you continue to use Visual Web Developer. If you use Visual Studio 2005, however, you'll be pleased to know that you can run NUnit as an add-in, using TestRunner for Visual Studio .NET (`http://www.mailframe.net/Products/TestRunner`). Visual Studio 2005 also has its own testing environment (and a whole lot more) called Visual Studio Team System. Team System is currently in the final stages of its beta process and should be available soon. For more details about Team System, see `http://msdn.microsoft.com/vstudio/teamsystem/default.aspx`.

Unit testing applies to only single modules, but you need to account for the way modules interact with each other, as well. Do they successfully share session and user information, for instance? If one module makes a call to another, is that call being made for the purpose the method was originally intended, or is it being forced into the engine of another car? If the latter is the case, you may want to investigate why this is being done and how better to achieve the desired results before continuing.

Measuring Performance

As is the case with most development, the strategies you use to measure the performance of your Web site are formed by the experience you've had using different techniques in previous projects and whether they have worked for you. You can then define the target metrics for a Web site and the tests you know will be able to prove that those benchmarks have been achieved.

But where do you begin? What methods can you use to stress test your Web site and retrieve results? Can you continue to work on your Web site without buying an IDE? Of course you can. Microsoft even tells you how. Refer to the ASP.NET performance page at `http://msdn.microsoft.com/asp.net/using/understanding/perf/default.aspx`. This page links to many articles, each covering a different aspect of performance you may not have considered and the various ways to measure and improve it. In particular, look first at the articles entitled "ASP.NET Performance Monitoring, and When to Alert Administrators," "MyTracer Monitors and Traces ASP.NET Apps," and "Real-World Load Testing Tips to Avoid Bottlenecks When Your Web App Goes Live." For a great resource on testing and improving the performance of a database, look no further than `http://www.sql-server-performance.com`. It focuses specifically on SQL Server, but a lot of the information applies to other databases as well.

■ **Note** You can also find several good books that cover performance issues and testing. See *Performance Tuning and Optimizing ASP.NET Applications* by Jeffrey Hasan and Kenneth Tu (1-59059-072-4; Apress, 2003) and *Test-Driven Development in Microsoft .NET* by James W. Newkirk and Alexei A. Vorontsov (0-73561-948-4; Microsoft Press, 2004).

Maintenance

Once your site has been implemented, tested, and deployed, that isn't the end of the story. There will always be some element of maintenance to your Web site. Unlike the other steps in the life cycle, the maintenance step will go on forever (well, until the Web site is switched off).

The main part of maintenance will be fixing any bugs that occur in the operation of the site. Most bugs affect the end user's experience of using the site and will, normally, need to be addressed as soon as possible.

However, that won't be the only part of the maintenance task. Your client will request changes to existing functionality and even the addition of new functionality. You'll need to handle and manage these feature requests.

Bug Fixes

All software, once completed, will have some bugs in it. No matter how thoroughly you test the Web site before it goes live, there will be problems. Like death and taxes, it's one of the few things in life that can be guaranteed.

When a bug is discovered, you need some way of being informed of the issue. In most cases, this will be an e-mail message informing you that an end user has experienced a problem. Or maybe you're using the `Error` event of the `Application` in `Global.asax` to automatically log the error (into a log file, by e-mail, or even directly into the database). However you're notified of the error, you need to deal with it.

The first step in bug fixing is to reproduce the error. Once you can do that, you can then investigate how to fix the problem. If you can't reproduce the error, then you'll need to communicate with the user who raised the error (if there was one) to get more details.

Fixing the problem may require changes to the database, as well as changes to the pages of the Web site. As noted earlier, any changes to the database structure have the potential to

cause havoc in the Web site. A code change in a page will normally affect only that page, but if you're making changes to the database, you need to make sure that the database change hasn't affected anything else.

The key part of bug fixing is testing. All changes to fix bugs in the Web site must be tested as rigorously as the Web site was tested before it went live. It is very easy for a simple change to impact parts of the Web site that you didn't think would be affected.

Feature Requests

When you released the Web site, you met all of the client's requirements. However, at some point, clients will find something that doesn't work quite as they wanted and request that you change the functionality. You'll also get requests for new features that weren't part of the original requirements (and you should be able to charge the client for this new work).

Whether the change is a small or a large change to the Web site, just as with bug fixes, you must consider the impact on the rest of the Web site. Some changes will be quite minor, requiring very little work in the analysis, design, and implementation stages. Others may be quite fundamental and require large changes to the Web site that can take a considerable amount of time to implement.

As with bug fixing, you should always fully test your changes to the Web site. Even a relatively minor change can impact parts of the Web site that you wouldn't expect and cause problems.

Issue Tracking

When maintaining a Web site, you can stick to the e-mail and post-it note solution, but that can very quickly degenerate into chaos. You need a better way to track issues.

Microsoft is planning to release an application that supports issue tracking, Team Foundation Server, as an add-on for Visual Studio. It integrates directly into the Visual Studio IDE, and as well as issue tracking, it also provides a complete source control system and allows the end-to-end tracking of issues (from the issue being raised to the individual code changes to fix the issues). For more information, see `http://msdn.microsoft.com/vstudio/teamsystem/team/default.aspx`.

Another option for issue tracking is FogBugz, available from `http://www.fogbugz.com`. Although it doesn't integrate into the Visual Studio IDE, it is a perfectly capable issue-tracking system.

Summary

In this chapter, you looked at some of the issues you'll face each time you start a new project. Working with a client on the specifications can be fruitless if you don't have a specific agenda. The more information you have, the more responsive your design can be to the resources you have available and the tasks you're trying to achieve. You can use the three-tier design to make your Web site more flexible and easier to develop. You can iron out more wrinkles in both design and specifications by building prototypes to prove the concept and showing those to the client.

When it comes to implementing the design, you need to keep in mind the good practices you've learned in this book and apply them evenly to improve security and performance.

But this is just the beginning. As you become more experienced, you'll quickly start to build an appreciation of what does and doesn't work. Web site development is a continually evolving skill. There will always be that "new way" of doing something that is slightly better than the way you done something in the past. Learn from your mistakes and don't repeat them!