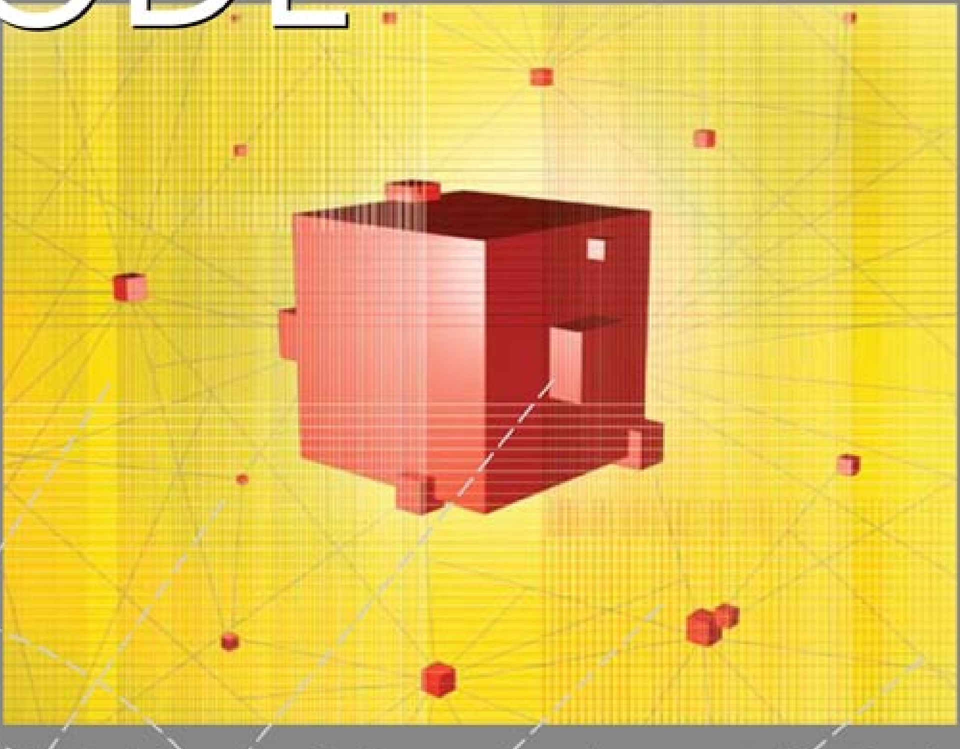
Microsoft

BEST PRACTICES

# SOLID CODE



Optimizing the Software Development Life Cycle

Donis Marshall and John Bruno

Foreword by John Robbins
*Debugging expert, Wintellect cofounder, and author*

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Library of Congress Control Number: 2008940526

Printed and bound in the United States of America.

1 2 3 4 5 6 7 8 9   QWT   4 3 2 1 0 9

Distributed in Canada by H.B. Fenn and Company Ltd.

A CIP catalogue record for this book is available from the British Library.

**Acquisitions Editor:** Ben Ryan
**Developmental Editor:** Devon Musgrave
**Project Editor:** Melissa von Tschudi-Sutton
**Editorial Production:** nSight, Inc.
**Technical Reviewer:** Per Blomqvist; Technical Review services provided by Content Master, a member of CM Group, Ltd.
**Cover Illustration by:** John Hersey

Body Part No. X15-28130

*This book is dedicated to my children: Jason, Kristen, and Adam.*
*Jason is a talented young man, Kristen is now in college,*
*and Adam (at 11) defeats me in chess regularly.*

*—Donis Marshall*

*To Christa, Christopher, and Patrick, this book is for you.*
*Your love and support inspire me every day.*

*—John Bruno*

## Recommendations for *Solid Code*

Solid Code *does a great job of hitting that super hard middle ground between the management books and the technology books. By covering ideas from how to model software to security design to defensive programming, Donis and John show you the best practices you can apply to your development to make it even better.*

*—John Robbins, Cofounder, Wintellect*

Solid Code *isn't just about code; it imparts the knowhow to deliver a solid project. This book delivers straightforward best practices, supplemented with case studies and lessons learned, from real products to help guide readers to deliver a perfect project—from design through development, ending with release and maintenance.*

*—Jason Blankman, Software Development Engineer, Microsoft Corporation*

*As a software developer of 20 years, there are a few books that I read again every couple of years. I believe that* Solid Code *will be one of the books that you will read over and over, each time finding new insight for your profession.*

*—Don Reamey, Software Development Engineer, Microsoft Corporation*

Solid Code *is an invaluable tool for any serious software developer. The book is filled with practical advice that can be put to use immediately to solidify your code base. Solid Code should definitely be on your shelf, close at hand, as you'll use it again and again!*

*—John Alexander, Microsoft Regional Director, Managing Partner, AJI Software*

Solid Code *is a must read for any IT professional, especially if you plan on using managed code. The book not only covers engineering best practices but also illustrates them with real test case studies.*

*Andres Juarez, Release Manager, Microsoft Corporation*

*This is a very well-written book that offers best practices in cultivating an efficient software development process by which typical developer mistakes can be avoided. The authors provide practical solutions for detecting mistakes and explain how software development and testing works at Microsoft.*

*Venkat B. Iyer, Test Manager, Microsoft Corporation*

*This book is excellent for developers at any level—beginner to experienced. It provides the foundation of great development practices that should be used by any size development team, and even by individual programmers.*

*John Macknight, Independent Software Developer*

# Contents at a Glance

# Table of Contents

# Foreword

Software engineering is not engineering. As a software developer, I would love nothing more than to say I am an engineer. Engineers think through and build things that are supposed to work the first time due to careful planning. So having the word “engineer” in my job title would be very cool indeed.

Let’s look at what would happen if the normal software engineering approach were applied to aerospace engineering. A plane is sitting at a gate boarding passengers, and an aerospace engineer—on a whim or forced by management—decides to replace the tail section. Because it’s just a tail section, let’s just rip it off and stick another one on right there at the gate. No problem, we can make it work! If aerospace engineering were approached like software engineering, I think the passengers would stampede to get off that plane as fast as possible. But those are the kind of changes that are made every day in major software projects the world over. The old joke is that “military intelligence” is an oxymoron, but I’d have to say that it fits “software engineering” as well. What makes this even more troubling to me is that software truly rules the world, but the approach nearly everyone takes to making it can in no way be called engineering.

Why is it that I know the physical computer I’m using right now will work, but the program I’m using, Microsoft Word, will screw up the auto numbering of my lists? While my electrical engineering friends will not be happy to hear this, hardware is easy. The electrical engineer has a limited number of inputs to work with, unlike the essentially unlimited number given to software developers.

Management also considers electrical engineering “real engineering,” so management gives the appropriate time and weight to those efforts. The software business, as a distinct field, is not a mature industry; it really hasn’t been around that long. In fact, I myself am slightly younger than the software business, so my youthful look reveals some of the problem. If I were as old as electrical engineering, I’d be writing this from the grave.

Another difficulty with software development can sometimes be the software developers themselves. Realistically, the barriers to becoming a software developer can be quite low. I’m a prime example: I was working as a full-time software developer before I had a bachelor’s degree in computer science. Because I was able to “talk the talk” in interviews, I was given a job writing software. None of my employers really cared about my lack of education because they could hire me cheaper than someone with a degree.

All real engineering fields require you to achieve ambitious certification criteria before you can add the Professional Engineer (PE) designation to your name. There’s nothing like that for the software industry. That’s due in part to the fact that no one can agree what all software developers should know because of the newness of the industry. In other fields, the PE designation appropriately carries huge weight with management. If a certified engineer says a design won't work, she won't sign off on the plans and the project won't go forward. That forces management to take the planning process much more seriously. Of course, by signing off on a project, the PE acknowledges liability for ethical and legal ramifications should things go wrong. Are you ready to sign off on the ethical and legal liability of your software's design? Until we get our industry to that point, we can't really call ourselves engineers in the traditional sense.

The good news is that even in the nearly 20 years I have been in the software development business I've seen huge changes for the better. Senior management is finally getting the message that software project failures cost companies serious amounts of money. Take a look at Robert Charette's "Why Software Fails" in the September 2005 issue of the *IEEE Spectrum* magazine (*http://www.spectrum.ieee.org/sep05/1685)* for a list of spectacular failures. With the costs so high, some senior management are finally committing real resources to get software projects kicked off, planned, and implemented right the first time. We still have a long way to go, but this buy-in for real planning from management is one of the biggest changes I've seen in my time in the industry.

On a micro level, the best change in software development is that nearly all developers are finally serious about testing their code. Now it's fortunately rare to hear about a developer who throws the code over the wall to the QA group and hopes for the best. This is a huge win for the industry and truly makes meeting schedules and quality gates achievable for many teams. As someone who has spent his career on the debugging and performance-tuning side of the business, I'm really encouraged about our industry becoming more mature about testing. Like all good change, the testing focus starts with the individual and the benefits work their way up the organization.

What's also driving change is that our tools and environments are getting much better. With .NET, we have an easy way to test our code, so that means more people will test. Also, the abstraction layers are moving up, so we no longer have to deal with everything on the computer. For example, if you need to make a Web service call, you don't have to manually open the port, build up the TCP/IP packet, call the network driver, wait for the data to return, or parse the return data. It's now just a method call. These better abstraction layers allow us to spend more time on the important parts of any software project: the real requirements and solving the user's problem.

We still have a long way to go before our field is a real engineering field, but the signs are encouraging. I think a big change will occur when we finally start treating testing as a real profession—one that is equal to or more important than development. While I probably won't see the transition to software engineering before I retire, I'm very encouraged by the progress thus far. Let's all keep pushing and learning so we can finally really be called engineers.

This book, *Solid Code*, is a great step in the direction of treating software as an engineering discipline. Bookstores' programming shelves groan under two types of development books. The first kind is the hand-waving software-management type, and the second is the gritty internals-of-a-technology type; I'm guilty of writing the latter. While those books have their uses and are helpful, the types of books we are missing are the ones that talk about real-world team software development. The actual technology is such a small part of a project; it's the team and process aspects that present the biggest challenges in getting a software project shipped. *Solid Code* does a great job of hitting that super hard middle ground between the management books and the technology books. By covering ideas from how to model software to security design to defensive programming, Donis and John, show you how the best practices you can apply to your development will make it even better. Reading *Solid Code* is like experiencing a great project lead by a top development manager and working with excellent coworkers.

The whole book is excellent; I especially loved the emphasis on planning and preparation. Many of the projects that my company, Wintellect, has had to rescue are the direct result of poor planning. Take those chapters to heart so you'll avoid the mistakes that will cost you tons of money and time. Another problem the book addresses is the tendency to leave performance tuning and security analysis for the very end of the project. As the title of Chapter 4 so succinctly points out, "Performance Is a Feature." The recommendations in those chapters are invaluable. Finally, the book's emphasis on real-world coding and debugging will pay dividends even when the code goes into maintenance mode. Even though I've been working in the field nearly 20 years, I picked up a lot of great ideas from *Solid Code*.

Every developer needs to read this book, but there are others in your company who need to read it as well. Make your manager, your manager's manager, and your manager's manager's manager read this book! The one question I always get from senior managers at any company is, "How does Microsoft develop software?" With the Inside Microsoft sections in most chapters of *Solid Code*, your management will see how Microsoft has solved problems in some of the largest applications in use today. Now start reading! It's your turn to help move our industry into a real engineering discipline!

*John Robbins*
*Co-founder, Wintellect*

# Acknowledgements

Isaac Newton has been credited with the phrase, "If I have seen further, it is only by standing on the shoulders of giants." That statement is certainly applicable to this book, especially when considering the practices, perspectives, and experiences contained within it. More specifically, those shoulders belong to the many people who have contributed to this project. Although our names adorn the cover, we owe much of the credit to the individuals who have helped bring this book to life. We are grateful for their efforts and support throughout this project, and would like to acknowledge them individually.

For starters, we could not have done it without the team at Microsoft Press. We would like to thank Ben Ryan, Devon Musgrave, and Melissa von Tschudi-Sutton for ensuring a high-quality outcome and keeping the project on schedule. Additionally, we would also like to thank the technical editor Per Blomqvist and copy editor Cindy Gierhart for their invaluable contributions and feedback.

As mentioned, this book includes practices, perspectives, and experiences. Many of these elements would not have been included without the contributions, support, and feedback of the professionals from Microsoft and the industry. Specifically, we would like to thank the contributors and reviewers: Jason Blankman, Eric Bush, Jacob Kim, Don Reamey, Dick Craddock, Andres Juarez Melendez, Eric Schurman, Jim Pierson, Richard Turner, Venkatesh Gopalakrishnan, Simon Perkins, Chuck Bassett, Venkat Iyer, Ryan Farber, and Ajay Jha.

There is also a special acknowledgement for Wintellect. Wintellect is a consulting, debugging, and training firm dedicated to helping companies build better software faster through a concentration on .NET and Windows development. Its services include in-depth, multiday .NET on-site and open enrollment training as well as development and consulting services including emergency debugging. The company also produces Devscovery conferences—three-day multitrack events targeting the intermediate to advanced developer. For more information about Wintellect, visit *www.wintellect.com*.

John Robbins and Jeffrey Richter of Wintellect provided invaluable insights and timely feedback. Thanks!

**Donis Marshall** I have written several books. However, this is my first book with a coauthor. I have been left with one important question after the completing the book. Why did I not have a coauthor on earlier book projects? John Bruno was an incredible asset to this project. His broad knowledge and insights have made this book an important read for any technologist in the Windows arena. John also possesses a rare attribute among authors—timeliness.

**John Bruno** Writing a book is a commitment that often affects those closest to you. I would like to first thank my wife, Christa, and my two sons, Christopher and Patrick, for their patience, understanding, and sacrifice during the development of this book. Their love and support inspire me to be the best man I can be, everyday. Additionally, I am grateful to Donis Marshall for inviting me to join him on this project. I sincerely appreciate his friendship and the opportunity to work with him on such an important subject. I have been fortunate throughout my life to have known many creative and insightful people. To those of you who have always been there to inspire, encourage, challenge, and support me, I thank you.

# Introduction

Software development has evolved greatly over the past several years. Improvements in programming languages and rapid development tooling, like .NET and Visual Studio 2008, have driven the software industry to build higher-quality software, faster, cheaper, and with more frequent upgrades or refreshes. Despite this continued demand for more software and the evolution in tools and processes, building and releasing quality software remains a difficult job for all participants of software projects, especially developers. Fortunately, this title encapsulates the essence of the best-in-class engineering practices, processes, policies, and techniques that application developers need for developing robust code.

*Solid Code* explores best practices for achieving greater code quality from nearly every facet of software development. This book provides practical advice from experienced engineers that can be applied across the product development life cycle: design, prototyping, implementation, debugging, and testing. This valuable material and advice is further supplemented by real world examples from several engineering teams within Microsoft, including, but not limited to, the Windows Live Hotmail and Live Search teams.

## Who Is This Book For?

*Solid Code* has something for every participant in the software development life cycle. Most specifically, it is targeted toward application developers who are seeking best practices or advice for building higher-quality software. Portions of this book illustrate the important role of the engineering process as it relates to writing high-quality code. Other parts focus on the criticality of testing. However, most of this book focuses on improving code quality during design and implementation, covering specific topics like class prototyping, performance, security, memory, and debugging.

This book targets both professional and casual developers. Readers should have a basic understanding of programming concepts and object oriented programming in C#. There are no skill level expectations. *Solid Code* is about the practical application of best practices for managed code application development. The topics discussed within the book should resonate with managed code developers of all skill levels.

## Organization of This Book

Solid Code is organized similarly to the application development life cycle. The chapters are not separated into parts, but rather grouped according to four key principles. These principles are outlined in Chapter 1, “Code Quality in an Agile World”, and include: Focus on Design, Defend and Debug, Analyze and Test, and Improve Processes and Attitudes.

- **Focus on Design** One of the great themes of this book is the importance of thoughtful design as a means to improve overall product quality. To support this theme, practices such as class design and prototyping, metaprogramming, performance, scalability, and security are explored.
- **Defend and Debug** Although great designs are critical to building a high-quality software application, it is equally important to understand the pitfalls that hinder delivery of bug-free code. Topics such as memory management, defensive programming techniques, and debugging are all discussed in the context of this principle.
- **Analyze and Test** Even the greatest programmers produce bugs despite following the recommended best practices. Therefore, it is important to discuss code analysis and testing as methods for further improving code quality.
- **Improve Processes and Attitudes** Beyond best practices, engineering processes and culture can have a great impact on the quality of the work being produced. We explore several key topics for improving the efficiency of the team as well as their passion for quality.

## System Requirements

You will need the following hardware and software (at a minimum) to build and run the code samples for this book in a 32-bit Windows environment:

- Windows Vista, Windows Server 2003 with Service Pack 1, Windows Server 2008, or Windows XP with Service Pack 2
- Visual Studio 2008 Team System
- 2.0 gigahertz (GHz) CPU; 2.6 GHz CPU is recommended
- 512 megabytes (MB) of RAM; 1 gigabyte (GB) is recommended
- 8 GB of available space on the installation drive; 20 GB is recommended
- CD-ROM or DVD-ROM drive
- Microsoft mouse or compatible pointing device

### The Companion Web Site

This book features a companion Web site that provides code samples used in the book. This code is organized by chapter, and you can download it from the companion site at this address: *http://www.microsoft.com/learning/en/us/books/12792.aspx*.

## Find Additional Content Online

As new or updated material that complements this book becomes available, it will be published online to the Microsoft Press Online Developer Tools Web site. This includes material such as updates to book content, articles, links to companion content, errata, sample chapters, and more. This Web site is available at *http://www.microsoft.com/learning/books/online/developer* and it will be updated periodically.

## Support for This Book

Every effort has been made to ensure the accuracy of this book and companion content. Microsoft Press provides corrections for books through the Web at the following address:

*http://www.microsoft.com/mspress/support/search.aspx*

To connect directly to Microsoft Help and Support to enter a query regarding a question or issue you may have, go to the following address:

*http://support.microsoft.com*

If you have comments, questions, or ideas regarding the book or companion content or if you have questions that are not answered by querying the Knowledge Base, please send them to Microsoft Press using either of the following methods:

E-mail:

*mspinput@microsoft.com*

Postal mail:

Microsoft Press
Attn: *Solid Code* editor
One Microsoft Way
Redmond, WA 98052-6399

Please note that product support is not offered through the preceding mail addresses. For support information, please visit the Microsoft Product Support Web site at

*http://support.microsoft.com*

Chapter 4

# Performance Is a Feature

*My speed is my greatest asset.*

*—Peter Bondra, former professional ice hockey player*

Software can possess a broad array of useful features. Certain applications, such as Microsoft Office, include features that can help a user accomplish a near infinite number of tasks, many of which a normal user might never even discover. Other applications, like Notepad, may contain only the features necessary to accomplish a few simple tasks, which might leave certain users desiring more functionality. In either case, the goal of the software is the same: to provide functionality that helps users to accomplish a particular set of tasks. If we also consider how quickly users are able to accomplish that set of tasks, then performance, as suggested by the Peter Bondra quote, should also be considered an important feature of a software application.

As application developers, we spend considerable effort planning and building the key features of our applications. These features are cohesive, enhance the quality of our product, and implicitly improve overall functionality. One of the most important aspects of all features of a software product is performance. Performance is often overlooked or considered late in product design and development. This can lead to inadequate performance results for key features and overall poorer product quality. Performance is critical to the quality of any application but especially to Web applications. By contrast to desktop applications, Web applications depend on the transmission of data and application assets over a worldwide network. This presents architectural and quality challenges for Web application developers that must be mitigated during the design and construction of their applications.

Web application quality extends beyond the visible bugs that end users encounter when using the application. Network latency, payload size, and application architecture can have negative impacts on the performance of online applications. Therefore, performance considerations should be part of every Web application design. Deferring these considerations until late in the development cycle can create significant code churn after performance bugs are discovered. Application developers must understand the impact of the design choices that affect adversely performance and mitigate the risks of releasing a poorly performing Web application by applying many of the best practices discussed in this chapter.

Throughout the remainder of this chapter, we will evaluate some common problems that can negatively affect the performance of Web-based software, and we will discuss several practices that can be applied to proactively address performance bottlenecks. Although this chapter will not focus on techniques that are unique to managed code development, it will discuss several ways to apply performance best practices to your application development life cycle in order to increase the overall quality of your Web-based application, as well as the satisfaction for your application's users.

## Common Performance Challenges

Web-based applications that rely on interactions between servers and a user's Web browser inherently require certain design considerations to address the performance challenges present in the application execution environment. These factors are not specific to Web applications developed using ASP.NET; they also affect application developers who utilize Web development programming models like PHP or Java. They include the latency or quality of the connection between the client and server, the payload size of the data being transmitted, as well as poorly optimized application code, to name a few. Let's explore each of these in greater depth.

### Network Latency

To understand the impact of network latency and throughput on your Web application, we must first understand the general performance and throughput of the Internet in key regions around the world. This may prove to be an eye-opening experience for many Web application developers. The data in Table 4-1 illustrates how end users are affected by the network topology of the Internet. The data in this table was gathered during daily ping tests conducted between January through September 2008 and provide a breakdown of the average round-trip time (measured in milliseconds [ms]) and average packet loss for users in each specified region. Let us briefly review the definitions of each of these metrics before further evaluating the data in the table.

- **Average round-trip time** This refers to the average amount of time required for a 100-byte packet of data to complete a network round trip. The value in Table 4-1 is computed by evaluating the round-trip time for daily tests conducted over a period from January through September 2008.
- **Average packet loss** This metric evaluates the reliability of a connection by measuring the percentage of packets lost during the network round trip of a 100-byte packet of data. In the same way that average round-trip time is determined, the average packet loss is also computed by evaluating the results of daily tests conducted over a period of January through September 2008.

TABLE 4-1 **Internet Network Statistics by Region**

| Region | Average Round-Trip Time (ms) | Average Packet Loss (%) |
|---|---|---|
| Africa | 469 | 3.70 |
| Australia | 204 | 0.23 |

| Region | Average Round-Trip Time (ms) | Average Packet Loss (%) |
|---|---|---|
| Balkans | 202 | 0.74 |
| Central Asia | 597 | 1.24 |
| East Asia | 192 | 0.68 |
| Europe | 178 | 0.48 |
| Latin America | 270 | 1.15 |
| Middle East | 279 | 0.87 |
| North America | 59 | 0.09 |
| Russia | 243 | 2.48 |
| South Asia | 424 | 1.89 |
| South East Asia | 254 | 0.03 |

**Note** This data is based upon the results of tests being conducted between Stanford University in Northern California and network end points in 27 countries worldwide. Data obtained from each test is subsequently averaged across all end points within a particular region. The complete data set can be obtained from *http://www-iepm.slac.stanford.edu/*. Data is also available from this site in a summarized, percentile-based format, which shows what users at the 25th, 50th, 75th, 90th, and 95th percentile are likely to experience in terms of average round-trip time and packet loss. At Microsoft, teams generally assume that most of their users will experience connectivity quality at the 75th percentile or better.

There are a few key points to take away from the data presented in this table:

**Network reliability is poor in certain regions** The general throughput of data on the Internet varies according to region. This means that even if your Web application is available 100 percent of the time and performing perfectly, an end user in Asia might be affected by suboptimal network conditions such as high latency or packet loss and not be able to access your application easily. Although this seems to be a situation beyond an application developer's control, several mitigation strategies do exist and will be discussed later in this chapter. That said, it is definitely useful to understand the general network behavior across the Internet when you consider what an end user experiences when using your Web application.

**Average round-trip time is high** We also notice that the average round-trip time for a piece of data to travel from a point within North America to a point within another region is quite high in certain cases. For example, a single Transmission Control Protocol (TCP) packet of data traveling on the Internet between North America and Central Asia has an average round-trip time of 597 ms. This means that each individual file required by a Web application will incur 597 ms of latency during transfer between the server and the client. Thus, as the number of required requests increases, the performance of the application gets worse. Fortunately, the number of round trips between the client and the server is something every Web application developer can influence.

**Packet loss is high** In conjunction with average round-trip time, packet loss also increases significantly for users outside North America. Both of these factors are related to general throughput on the network, so they usually go hand-in-hand. These results demonstrate that, as packet loss increases, additional round trips are required between the browser and the server to obtain the packets of data lost in transmission. Hence, higher packet loss means decreased performance of your Web application. Even though developers cannot control the amount of packet loss a user is likely to experience, you can apply certain tactics to help mitigate the effects, such as decreasing payload size, which will be discussed later in this chapter.

## Payload Size and Network Round Trips

The term "payload size" loosely refers to the size of data being transmitted over the network to render the requested page. This could include the dynamic ASPX page content as well as static files such as JavaScript files, images, or cascading style sheets (CSS). The number of TCP requests required to retrieve the data is referred to as the "network round trips." Web application performance is most negatively affected by a combination of the payload size and the required round trips between the browser and the server. Let's take a look at a few examples of how typical Web application designs might contribute to poorly performing Web applications.

**Compression is not enabled** Compressing static and dynamic files are not necessarily part of your Web server's default configuration. Compression is strongly recommended for Web applications that use high amounts of bandwidth or when you want to use bandwidth more effectively. Many Web application developers might not be aware of this feature and could be unknowingly sending larger amounts of data to the client browser, thereby increasing the size of the payload. When enabled, compression can significantly reduce the size of the file being transmitted to the client browser. Compression requires additional CPU utilization when compressing dynamic content such as .aspx files. Therefore, if the CPU usage on your Web servers is already high, enabling Internet Information Services (IIS) dynamic compression is not recommended. However, enabling IIS static compression on file types such as JavaScripts, CSS, or HTML files does not increase CPU usage and is, therefore, highly recommended. Hosting static files with a Content Delivery Network service provider generally includes compression with the service offering.

**Using multiple small static image files** Most Web application developers naturally use references to individual images or iconography throughout their code. This is how most of us were taught to write our HTML. The reality is that each of these files, no matter how small we make them, results in a separate round trip between the browser and the server. Consider how bad this might be for image-rich Web pages where rendering a single page could generate dozens of round trips to the server!

These are just two simple examples of how typical Web applications could be delivering unnecessarily large payloads as well as initiating numerous round trips. The challenge facing Web application developers is to both reduce the amount of data being transmitted between the server and the client as well as optimize their Web application's architecture to minimize the number of network round trips. Fortunately, a number of tactics can help Web application developers accomplish this, all of which we'll explore later in this chapter. For now, we will continue reviewing some of the more common performance problems facing Web application developers.

## Limited TCP Connections

We've discussed how an individual HTTP request is made for each resource (such as JavaScript files, CSS, or images) within a Web page, which can negatively affect the rendering performance of the page. However, it may come as a surprise to you to learn that the HTTP/1.1 specification suggests that browsers should download only two resources at a time in parallel for a given hostname. This implies that, if all content necessary to render a page is originating from the same hostname (e.g., *http://www.live.com*), the browser will retrieve only two resources at a time. Thus, the browser will utilize only two TCP connections between the client and the server. This phenomenon is illustrated in Figure 4-1, and although configurable in some browsers and ignored by newer browsers like Internet Explorer 8, it very likely affects users of your Web application.

**Note** Even though Internet Explorer allows the number of parallel browser sessions to be configured, normal users are unlikely to do this. For more information on how to change this setting in Internet Explorer, see the following Microsoft Knowledge Base article: *http://support.microsoft.com/?kbid=282402*.

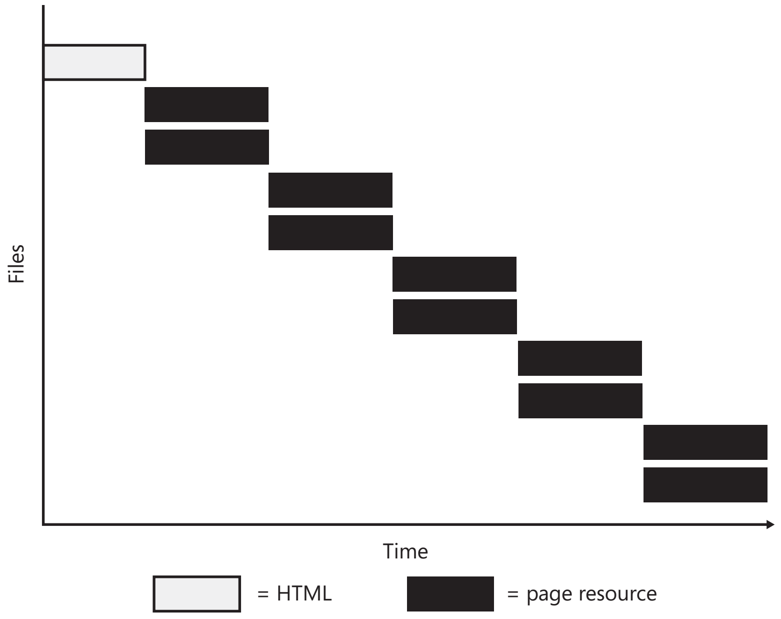


**FIGURE 4-1** Theoretical example of resources downloading in parallel for a single hostname.

**Note** Several figures in this chapter intend to illustrate how parallel downloading of resources theoretically works in a Web browser. In reality, resources are often of varying sizes and therefore will download in a less structured way than illustrated here. To understand this phenomenon in greater detail, download and run HTTPWatch (*http://www.httpwatch.com*) against your Web application.

As you probably realize, the TCP connection limitation could have profoundly negative effects on performance for Web applications that require a good deal of content to be downloaded. It is critically important for Web application developers to consider this limitation and properly address this phenomenon in the design of their applications. We will evaluate mitigation strategies for this later in this chapter.

## Poorly Optimized Code

Performance challenges for Web application developers are not solely related to network topology or data transmission behavior between client browsers and Web servers. It is true that connectivity and data transmission play a big role in the performance of Web applications, but application architecture and application coding play a big role as well. Oftentimes Web application developers will choose a particular implementation within their application architecture or code without fully realizing the impact of the decision on a user of the application. Some examples of common implementations that have a negative impact on Web application performance include the overuse of URL redirects, excessive Domain Name System (DNS) lookups, excessive use of page resources, and poor organization of scripts within a Web page. Let us review each of these in greater detail.

**Overuse of redirects** These are typically used by developers to route a user from one URL to another. Common examples include use of the *<meta equiv-http="refresh" content="0; url=http://contoso.com">* directive in HTML and the *Response.Redirect("http://fabrikam.com")* method in ASP.NET. While redirects are often necessary, they obviously delay the start of the page load until the redirect is complete. This could be an acceptable performance degradation in some instances, but, if overused, it could cause undesirable effects on the performance of your Web application's pages.

**Excessive DNS lookups** DNS lookups are generally the result of the Web browser being unable to locate the IP address for a given hostname in either its cache or the operating system's cache. If the IP address of a particular hostname is not found in either cache, a lookup against an Internet DNS server will be performed. In the context of a Web page, the number of lookups required will be equal to the number of unique hostnames, such as *http://www.contoso.com* or *http://images.contoso.com*, found in any of the page's JavaScript, CSS, or inline code required to render that page. Therefore, multiple DNS lookups could degrade the performance of your Web application's pages by upwards of *n* times the number of milliseconds required to resolve the IP address through DNS, where *n* is equal to the number of unique hostnames found in any of the page's JavaScript, CSS, or inline code requiring a DNS lookup. While there are exceptions to this rule that we will explore when discussing the use of multiple hostnames to increase parallel downloading, it is generally not advisable to include more than a few unique hostnames within your Web applications.

**Poorly organized JavaScript and CSS** Web application developers may not have given a lot of thought to how code organization affects performance of Web applications. In many cases, developers choose to separate JavaScript code from CSS for maintainability. While this practice generally makes sense for code organization, it actually hurts performance because it increases the number of HTTP requests required to retrieve the page. In other cases, the location of script and CSS within the structure of the HTML page can have a negative effect on gradual or progressive page rendering and download parallelization.

It is important for Web application developers to understand how these simple choices can affect their Web application's performance, so they can take the appropriate mitigation steps when designing their applications. Let's review an example of how to analyze Web page performance and begin discussing mitigation strategies for the common problems we have been discussing thus far.

## Analyzing Application Performance

The key to a fast Web application is to understand the application's behavior from the user's perspective. Naturally, this requires a combination of analysis tools and an investment of time to evaluate the resultant data from the analysis tools. Analyzing Web applications is far from a simple task. Developers must evaluate many facets of the application's behavior, including but not limited to such items as the network traffic, the sequence of events that occurs during a page load, and the different rendering behaviors caused by client-side technologies like JavaScript and CSS. Unfortunately, Microsoft does not offer an end-to-end toolset that works in conjunction with Visual Studio to allow for a holistic analysis of Web application performance. There is, however, a collection of stand-alone tools available, both from Microsoft as well as other vendors, for conducting such an analysis, many of which we will discuss in this chapter.

As we discussed earlier in this chapter, when Web application pages are requested, the browser governs the flow of content from the server to the user and performs rendering based on several different factors. Much of the downloading of content is serial, meaning that, while the browser is retrieving a piece of content, it is delaying the retrieval of other content. To understand this and other interactions between the browser and the server, application developers should familiarize themselves with the diversity of tools that are available for analyzing these interactions. There are several tools that are freely available and very effective at analyzing certain parts of the browser and server interaction, including but not limited to Fiddler, Network Monitor, Visual Round Trip Analyzer, HTTPWatch, Firebug, and Y!Slow. The following information represents an overview of these products. A more detailed review of these tools is beyond the scope of this chapter.

**Fiddler** This is one of the most widely used tools among Web application developers at Microsoft. Fiddler is a freely available HTTP debugging proxy application that captures all HTTP information between the client browser and the server and allows application developers to inspect and manipulate incoming and outgoing data. This tool was not designed strictly for performance analysis but rather for the broader purpose of enabling detailed inspection of the Web application's HTTP traffic. However, it is quite useful for performance analysis and understanding the detailed HTTP interactions between the browser and the server. This enables developers to gain insight into HTTP transaction details like the number of requests for a given page load, header values, and many other page load characteristics. Most Web application developers would be pleasantly surprised by the power of this tool and are encouraged to spend some time playing with it.

**Network Monitor** This application has been available from Microsoft for several years and is primarily a protocol analyzer, or packet sniffer. It allows application developers to inspect network traffic at a very low level and analyze application behavior at essentially the packet level. Network Monitor is a great tool for conducting network-level analysis, but it is rather complex to understand and requires knowledge of networking, packet sniffing, and related technologies. It is not the tool you would use all that frequently, but it does provide a depth of information that other tools do not.

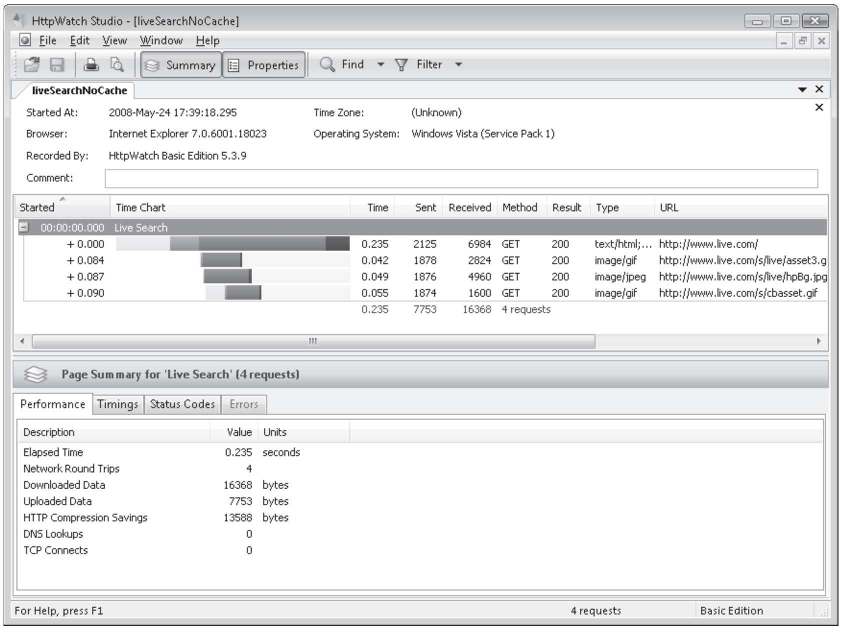
**Visual Round Trip Analyzer** As a complement to Network Monitor, Microsoft recently released a tool for analyzing page performance and behavior over the network called Visual Round Trip Analyzer (VRTA). Although previously available as an internal Microsoft tool, VRTA is a solid (and free) addition to the commercially available set of performance analysis tools. VRTA works in conjunction with Network Monitor to capture the HTTP traffic between the client and the server, and it renders an informative, graphical representation of the transaction. This analysis includes information about the number, type, and download pattern of all file types in the transaction as well as their respective sizes. It further provides information about how well the page was leveraging the available bandwidth, as well as recommendations for where improvements can be made to the page. Generally speaking, this tool builds on top of the powerful things already being done by Network Monitor but distills the output in a way that presents actionable results for application developers.

**HTTPWatch** Similar to Fiddler, HTTPWatch from Simtec Limited captures all HTTP traffic between the client browser and the server and provides a useful interface for analyzing the captured information. Unlike Fiddler, HTTPWatch provides a more powerful graphical representation of the page rendering behavior. This allows an application developer to easily acquire a deep understanding of the interaction between the browser and the server by simply exploring each step of the page rendering process. Figure 4-2 (shown later in this chapter) illustrates an analysis of Microsoft's Live Search home page.

In addition to those just described, there are other tools that are also helpful for developers when analyzing Web page performance. Those include the freely available Firebug, which is an add-on for the Firefox Web browser; the developer toolbar for Internet Explorer, which helps with page troubleshooting and debugging; and Y!Slow, which is a tool built by the performance team at Yahoo!. Each of these tools shares functionality similar to the tools mentioned above and will likely complement any Web application developer's analysis toolset. Application developers are encouraged to investigate each of the tools discussed and to choose the tool or tools that best help to augment their analysis efforts. A list of these tools and their respective Web sites has been provided in Appendix B of this book.

## Analyzing the Performance of Live Search

To further illustrate how developers can analyze their Web applications using the tools mentioned previously, we will review Microsoft's Live Search application. Using HTTPWatch, which runs as an Internet Explorer add on, we clear the browsers cache and use the recording functionality to capture the results of a main page load from *http://www.live.com*. HTTPWatch generates the analysis shown in Figure 4-2.


HttpWatch Studio - [liveSearchNoCache]
File Edit View Window Help
Summary Properties Find Filter
liveSearchNoCache

Started At: 2008-May-24 17:39:18.295 Time Zone: (Unknown)
Browser: Internet Explorer 7.0.6001.18023 Operating System: Windows Vista (Service Pack 1)
Recorded By: HttpWatch Basic Edition 5.3.9
Comment:

| Started | Time Chart | Time | Sent | Received | Method | Result | Type | URL |
|---|---|---|---|---|---|---|---|---|
| 00:00:00.000 | Live Search | | | | | | | |
| + 0.000 | | 0.235 | 2125 | 6984 | GET | 200 | text/html;… | http://www.live.com/ |
| + 0.084 | | 0.042 | 1878 | 2824 | GET | 200 | image/gif | http://www.live.com/s/live/asset3.g |
| + 0.087 | | 0.049 | 1876 | 4960 | GET | 200 | image/jpeg | http://www.live.com/s/live/hpBg.jpg |
| + 0.090 | | 0.055 | 1874 | 1600 | GET | 200 | image/gif | http://www.live.com/s/cbasset.gif |
| | | 0.235 | 7753 | 16368 | 4 requests | | | |

Page Summary for 'Live Search' (4 requests)

| Description | Value | Units |
|---|---|---|
| Elapsed Time | 0.235 | seconds |
| Network Round Trips | 4 | |
| Downloaded Data | 16368 | bytes |
| Uploaded Data | 7753 | bytes |
| HTTP Compression Savings | 13588 | bytes |
| DNS Lookups | 0 | |
| TCP Connects | 0 | |

For Help, press F1 4 requests Basic Edition

**FIGURE 4-2** HTTPWatch analysis of *http://www.live.com* without browser caching.

In the lower window, under the performance tab, HTTPWatch generates some statistics about the page load. Metrics such as the elapsed time, number of network round trips, size of the downloaded data, and the HTTP compression efficiency provide some indication about how this page is performing. Note that some of the features in this window may not be available in the Basic Edition of HTTPWatch, which is available for free. Specifically, we note the following to be true.

- The elapsed page load time is 0.235 seconds.
- The total number of network round trips was four.
- The amount of data downloaded was 16.3 kilobytes (KB), which includes all relevant content, JavaScript, CSS, and image assets.
- The amount of data uploaded was 7.7 KB, which includes the transmission of cookies and request header values.
- HTTP compression saved 13.5 KB from being transferred to the client, which is an approximate 45 percent reduction.
- DNS was served from a local machine cache, which saved remote DNS lookups.
- TCP connects indicate that Keep-Alives are enabled on the Web servers.

This data helps us to understand what is happening between the browser and the server quite well. However, to better understand what the user is experiencing, we need to observe the interaction between the server and the browser through the illustration in the upper window. This Gantt chart–style illustration depicts the behavior of the application from the initial server request to the end of the page load, where each bar represents an instance of an HTTP request for a particular application asset or assets, like HTML, images, or JavaScript. Notice that the first bar shows how much time elapsed before the main content of the page was retrieved, and the subsequent bars show the point at which certain image assets are being rendered. In this case, the end point of the first bar indicates when the user actually sees the content get rendered, which is 0.235 seconds after the request was issued. As previously noted, the total page content was delivered to the browser in 0.235 seconds, which consisted of four total network round trips.

Based on the brief analysis of this data, we can conclude that this is an example of a page that is well optimized for performance. This is evident from the low number of HTTP requests, the size of the data being downloaded, and the use of several other best practices, all of which we will discuss later in this chapter. As an experiment, download a free copy of HTTPWatch and use it yourself against a few of your favorite pages. You may be surprised by what you find. Although the capabilities of the free version of HTTPWatch will be limited, you will quickly obtain a visual representation of your page performance.

Although this was a simple example, it does provide interesting data points that help depict the page load characteristics of the Live Search Web application. Tools like HTTPWatch and Fiddler provide developers the ability to evaluate the detailed HTTP information being transferred between the server and the browser, so each page load behavior can be better understood, and performance problems can be prevented. When combined with packet sniffing tools like Network Monitor, developers can quickly gain insight into the end-to-end page load characteristics from the network layer to the Web browser. In general, this toolset will allow Web application developers to get a better understanding for what their users are experiencing, so that performance issues or bottlenecks can be avoided before the application is released.

## Tactics for Improving Web Application Performance

Earlier in this chapter, we discussed several of the architectural challenges that face developers when building high-performance Web applications. Many of these challenges stem from the basic interaction model between Web browsers and Web servers. They include such issues as network latency and the quality of the connection, payload size and round trips between client and server, as well as the way code is written and organized. These issues generally transcend multiple development platforms and affect every Web application developer, whether they are developing ASP.NET and managed code Web applications or using an alternative technology like PHP or Java. It is important for developers to understand these issues and incorporate performance considerations in their application designs. Performance bugs that are discovered late in the release cycle can create significant code churn and add risk to delivering a stable and high-quality application.

There are several best practices for improving the performance of a Web application, which have been categorized into four basic principles below. These principles are intended to help organize very specific, tactical best practices into simple, high-level concepts. They include the following:

- **Reduce payload size** Application developers should optimize Web applications to ensure the smallest possible data transfer footprint on the network.
- **Cache effectively** Performance can be improved when application developers reduce the number of HTTP requests required for the application to function by caching content effectively.
- **Optimize network traffic** Application developers should ensure that their application uses the bandwidth as efficiently as possible by optimizing the interactions between the Web browser and the server.
- **Organize and write code for better performance** It is important to organize Web application code in a way that improves gradual or progressive page rendering and ensures reductions in HTTP requests.

Let us review each of these principles thematically and discuss more specific, tactical examples for applying performance best practices to several facets of your Web application.

## Reduce Payload Size

As reviewed earlier in this chapter, one of the primary challenges to delivering high-performance Web applications is the bandwidth and network latency between the client and the server. Both will vary between users and most certainly vary by locale. To ensure that users of your Web application have an optimal browsing experience, application developers should optimize each page to create the smallest possible footprint on the network between the Web server and the user's browser. There are a number of best practices that developers can leverage to accomplish this. Let us review each of these in greater detail.

**Reduce total bytes by using HTTP compression** Web servers like IIS, Apache, and others offer the ability to compress both static and dynamic content using standard compression methods like gzip and deflate. This practice ensures that static content (JavaScript files, CSS, and HTML files) and dynamic content (ASP and ASPX files) are compressed by the Web server prior to being delivered to the client browser. Once delivered to the client, the browser will decompress the files and leverage their contents from the local cache. This ensures that the size of the data in transit is as small as possible, which contributes to a faster retrieval experience and an improved browsing experience for the user overall. In the example illustrated in

Figure 4-2, compression reduced the payload size by 13.5 KB, or by approximately 45 percent, which is a modest reduction.

**Minify JavaScript and CSS** Minification is the practice of evaluating code like JavaScript and CSS and reducing its size by removing unnecessary characters, white space, and comments. This ensures that the size of the code being transferred between the Web server and the client is as small as possible, thus improving the performance of the page load time. There are several minifier utility programs available on the Internet today such as YUI Compressor for CSS or JSMin for JavaScript, and many teams at Microsoft, for instance, share a common minifier utility program for condensing JavaScript and CSS. This practice is very effective at reducing JavaScript and CSS file sizes, but it often renders the JavaScript and CSS unreadable from a debugging perspective. Application developers should not incorporate a minification process into debug builds but rather into application builds that are to be deployed to performance testing environments or live production servers.

**Re-palletize images** Another way to reduce the payload size of a Web page is to reduce the size of the images that are being transmitted for use within the page. When coupled with the use of CSS Sprites, which will be discussed later in this chapter, this technique can further optimize the transmission of data between the Web server and the user's computer. Adobe published a whitepaper[1] that provides insight into how reducing the color palette in iconography and static images can have a dramatic savings on the size of an image. By simply reducing the color palette in an image from 32 bit to 16 bit to 8 bit colors, it is possible to reduce the image size by upwards of 40 percent without degrading the quality of the image. This can produce dramatic results when extrapolated out to hundreds of thousands of requests for the same image.

## Cache Effectively

As we have seen, Web application performance is improved significantly by incorporating various strategies for reducing the payload size over the network. In addition to shrinking the footprint of the data over the wire, application developers can also leverage page caching strategies that will help reduce the number of HTTP requests sent between the server and the client. Incorporating caching within your application will ensure that the browser does not unnecessarily retrieve data that is locally cached, thereby reducing the amount of data being transferred and the number of required HTTP requests.

**Set expiration dates** A Web server uses several HTTP headers to inform the requesting client that it can leverage the copy of the resource it has in its local cache. For example, if certain cache headers are returned for a specific image or script on the page, then the browser will not request the image or script again until that content is deemed stale. There are several examples of these HTTP headers, including *Expires*, *Cache-Control,* and *ETag*. By

1 *http://www.adobe.com/uk/education/pdf/cib/ps7_cib/ps7_cib14.pdf*

leveraging these headers effectively, application developers can ensure that HTTP requests sent between the server and the client will be reduced as the resource remains cached. It is important for developers to set this value to a date that is far enough in the future that expiration is unlikely. Let's consider a simple example.

```
Expires: Fri, 14 May 2010 14:00 GMT
```

**Note** The preceding code is an example of setting an *Expires* header on a specific page resource like a JavaScript file. This header tells the browser that it can use the current copy of the resource until the specified time. Note the specified time is far in the future to ensure that subsequent requests for this resource are avoided for the foreseeable future. Although this is a simple method for reducing the number of HTTP requests through caching, it does require that all page resources, like JavaScript, CSS, or image files, incorporate some form of a versioning scheme to allow for future updates to the site. Without versioning, browsers and proxies will not be able to acquire new versions of the resource until the expiration date passes. To address this, developers can append a version number to the file name of the resource to ensure that resources can be revised in future versions of the application. This is just one example of ways to apply caching to your application's page resources. As mentioned, leveraging *Cache-Control* or *ETag* headers can also help achieve similar results.

**Note** Each of these HTTP headers requires in-depth knowledge of correct usage patterns. I recommend reading *High Performance Web Sites* (O'Reilly, 2007), by Steve Souders, or *Caching Tutorial for Web Authors and Web Masters*, by Mark Nottingham[2] before incorporating them in your application.

## Optimize Network Traffic

The network on which application data is being transferred between the server and the Web browser is one element within the end-to-end Web application pipeline that developers have the least control over, in terms of architecture or implementation. As developers, we must trust that network engineers have done their best to implement the fastest and most efficient networks so that the data we transmit is leveraging the most optimal route between the client and the server. However, the quality of the connection between our Web applications and our users is not always known. Therefore, we need to apply various tactics that both reduce the payload size of the data being transmitted as well as reduce the number of requests being sent and received. Application developers can accomplish this by incorporating the following best practices.

**Increase parallel TCP ports** If your Web application requires a large number of files to render pages, then increasing the number of parallel TCP ports will allow more page content

[2] *http://www.mnot.net/cache_docs/*

to be downloaded in parallel. This is a great way to speed up the time it takes to load the pages in your Web application. We discussed earlier how the HTTP/1.1 specification suggests that browsers download only two resources at a time in parallel for a given hostname. Web application developers must utilize additional hostnames within their application to allow the browser to open additional connections for parallel downloading. The simplest way to accomplish this is to organize your static content (e.g., images, videos, etc.) by unique hostname. The following code snippet is a recommendation for how best to accomplish this.

```
<img src="http://images.contoso.com/v1/image1.gif"/>
<embed src="http://video.contoso.com/v1/solidcode.wmv" width="100%"
height="60" align="center"/>
```

By leveraging multiple hostnames, parallel downloading of content by the browser will be encouraged. Figure 4-1, shown previously, illustrated how page content is downloaded when a single hostname is used. Figure 4-3 contrasts that by illustrating how the addition of multiple hostnames affects the downloading of content.

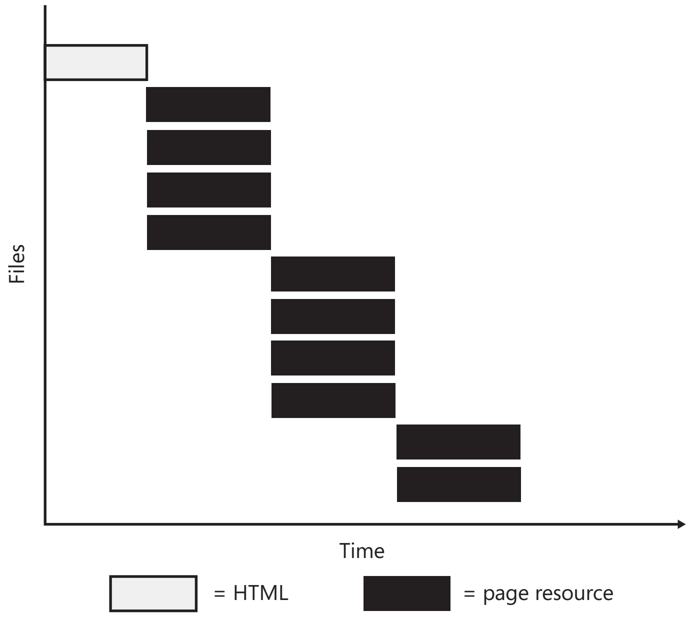


**FIGURE 4-3** Theoretical example of resources downloading in parallel for multiple hostnames.

**Enable Keep-Alives** Keep-Alives is the way in which servers and Web browsers use TCP sockets more efficiently when communicating with one another. This was brought about to address an inefficiency with HTTP/1.0 whereby each HTTP request required a new TCP socket connection. Keep-Alives let Web browsers make multiple HTTP requests over a single connection, which increases the efficiency of the network traffic between the browser and the server by reducing the number of connections being opened and closed. This is accomplished by leveraging the *Connection* header that is passed between the server and the browser. The following example is an HTTP response header, which illustrates how Keep-Alives are enabled for Microsoft's Live Search service.

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
X-Powered-By: ASP.NET
P3P: CP="NON UNI COM NAV STA LOC CURa DEVa PSAa PSDa OUR IND",
policyref="http://privacy.msn.com/w3c/p3p.xml"
Vary: Accept-Encoding
Content-Encoding: gzip
Cache-Control: private, max-age=0
Date: Tue, 30 Sep 2008 04:30:19 GMT
Content-Length: 8152
Connection: keep-alive
```

In this example, we notice a set of HTTP headers and their respective values were returned. This header indicates that an HTTP 200 response was received by the browser from the request to the host Uniform Resource Identifier (URI), which is *www.live.com*. In addition to other interesting information, such as the content encoding or content length, we also notice there is an explicit header called *Connection*, which indicates that Keep-Alives are enabled on the server.

**Reduce DNS lookups** Previously, we discussed how DNS lookups are the result of the Web browser being unable to locate the IP address for a given hostname in either its cache or the operating system's cache. These lookups require calls to Internet-based DNS servers and can take up to 120 ms to complete. This can adversely affect the performance of a Web page if there are a large number of unique hostnames found in any of the JavaScript, CSS, or inline code required to render that page. Reducing DNS lookups can improve the response time of a page, but it must be done judiciously as it can also have a negative effect on parallel downloading of content. As a general guideline, it is not recommended to utilize more than four to six unique hostnames within your Web application. This compromise will maintain a small number of DNS lookups while still leveraging the benefits of increased parallel downloading. Furthermore, if your application does not contain a large number of assets, it is generally better to leverage a single hostname.

**Avoid redirects** Web page redirects are used to route a user from one URL to another. While redirects are often necessary, they delay the start of the page load until the redirect is complete. In some cases, this could be acceptable performance degradation, but overuse could cause undesirable effects on the user experience. Generally, redirects should be avoided if possible, but understandably they are useful in circumstances where application developers need to support legacy URLs or certain vanity URLs used to make remembering a page's location fairly easy. In the sample redirect below, we see how the Live Search team at Microsoft redirects the URL *http://search.live.com* to *http://www.live.com*. On my computer, this redirect added an additional 0.210 seconds to the total page load time, as measured with HTTPWatch.

```
HTTP/1.1 302 Moved Temporarily
Content-Length: 0
Location: http://www.live.com/?searchonly=true&mkt=en-US
```

In this example, we notice that the HTTP response code is a 302, which indicates that the requested URI has moved temporarily to another location. The new location is specified in the *Location* header, which informs the browser where to direct the user's request. The browser will then automatically redirect the user to the new location.

**Leverage a Content Delivery Network** Earlier in this chapter, we discussed how incorporating caching in Web applications can reduce the number of HTTP requests. In addition to caching, leveraging the services of a Content Delivery Network (CDN) provides a complementary solution that also improves the speed at which static content is delivered to your users. CDNs like those offered by Akamai Technologies or Limelight Networks allow application developers to host static content, such as JavaScript, CSS, or Flash objects on globally distributed servers. Users who request this content required by a particular Web page are dynamically routed to the content that is closest to the originating request. This not only increases the speed of content delivery, but it also offers a level of redundancy for the data being served. Although there can be a high cost associated with implementing a CDN solution, the results are far and away worthwhile for Web applications that typically use a large volume of static content and require global reach.

**Incorporate CSS Sprites** CSS Sprites group several smaller images into one composite image and display them using CSS background positioning. This technique is recommended for improving Web application performance, as it promotes a more effective use of bandwidth when compared with downloading several smaller images independently during a page load. This practice is very effective because it leverages the sliding windows algorithms used by TCP. Sliding windows algorithms are used by TCP as a way to control the flow of packets between computers on the Internet. Generally, TCP requires that all transmitted data be acknowledged by the computer that is receiving the data from the initiating computer. Sliding window algorithms are methods that enable multiple packets of data to be acknowledged with a single acknowledgement instead of multiple. Therefore, sliding windows will work better for transmission of fewer, larger files rather than several smaller ones. This means that application developers who build Web applications that require a number of small files like iconography, or other small static artwork, should cluster images together and display those using CSS Sprites. Let us explore an example of how CSS Sprites are utilized.

Consider the following code snippet from Microsoft's Live Search site in conjunction with Figure 4-4. Notice that Figure 4-4 is a collection of four small icon files that have been combined into a single vertical strip of images. The code below loads the single file of three images known as *asset4.gif* and uses CSS positioning to display them as what appears to be singular images. This methodology ensures that only one HTTP request is made for the single image file, instead of four. Although this is a method of rendering images that is very different from what most Web developers have been taught, it promotes a much better performing experience than traditional image rendering. Therefore, this practice is recommended for Web applications that require several small, single images.

```
<style type="text/css">
input.sw_qbtn
{
background-color: #549C00;
background-image: url(/s/live/asset4.gif);
background-position: 0 -64px;
background-repeat: no-repeat;
border: none;
cursor: pointer;
height: 24px;
margin: .14em;
margin-right: .2em;
vertical-align: middle;
width: 24px; padding-top:24px;line-height:500%
}
</style>
```

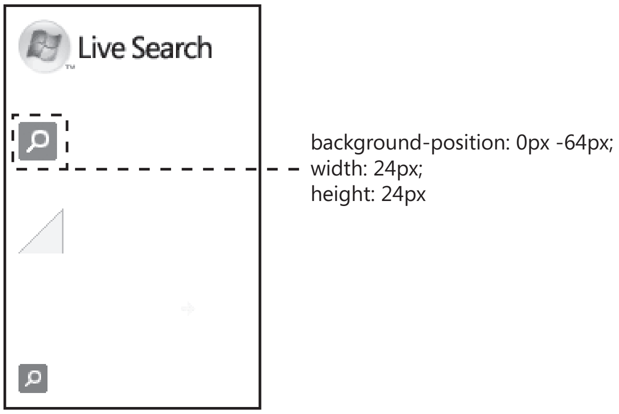


**FIGURE 4-4** Example of a Live Search CSS Sprite.

## Organize and Write Code for Better Performance

Thus far, we have discussed some examples of architectural and coding best practices for developing high performance Web applications. Let us review a few additional practices related to the organization and writing of application code that will also help improve the performance of your Web application pages.

**Make JavaScript and CSS files external** Application developers have two basic options for incorporating JavaScript and CSS in their Web applications. They can choose to separate the scripts into external files or add the script inline within the page markup. Generally, in terms of pure speed, inserting JavaScript and CSS inline is faster in terms of page load rendering, but there are other factors that make this the incorrect choice. By making JavaScript and CSS external, application users will benefit from the inherent caching of these files within the Web browser, so subsequent requests for the application will be faster. However, the downside to this approach is that the user incurs the additional HTTP requests for fetching the file or files. Clearly there are tradeoffs to making scripts external for initial page loads, but in the long run where users are continuously returning to your application, making scripts external is a much better solution than inserting script inline within the page.

**Ensure CSS are in the top of the page** Progressively rendering a Web page is an important visual progress indicator for users of your Web application, especially on slower connections. Ideally, we want the browser to display the page content as quickly as it is received from the server. Unfortunately, some browsers will prohibit progressive rendering of the page if Style Sheets are placed near the bottom of the document. They do this to avoid redrawing elements of the page if their respective styles change. Application developers should always reference the required CSS files within the *HEAD* section of the HTML document so that the browser knows how to properly display the content and can do so gradually. If CSS files are present outside the *HEAD* section of the HTML document, then the browser will block progressive rendering until it finds the necessary styles. This produces a more poorly performing Web browsing experience.

**Place JavaScript at the bottom of the page** When Web browsers load JavaScript files, they block additional downloading of other content, including any content being downloaded on parallel TCP ports. The browser behaves in this manner to ensure that the scripts being downloaded execute in the proper order and do not need to alter the page through *document.write* operations. While this makes perfect sense from a page processing perspective, it does little to help the performance of the page load. Therefore, application developers should defer the loading of any JavaScript until the end of the page, which will ensure that the application users get the benefit of progressive page loading.

> **Note** It is worth mentioning that the release of Internet Explorer 8.0 addresses this problem. However, application developers should be cognizant of the browser types that are being utilized to use their application and design accordingly.

Throughout this chapter, we have discussed a variety of common challenges and circumstances that lead to poorly performing Web applications. We have also seen how, with each challenge, there is a corresponding mitigation strategy or technique for ensuring that Web application pages perform well. While these techniques and strategies provide a tactical means to improve the performance of your Web applications, performance best practices must also be incorporated into day-to-day engineering processes and procedures. The key to implementing engineering best practices, such as those associated with performance, is to ensure that they are properly complemented by a sound set of engineering processes that incorporate them into the normal rhythm of software development. Let's review how best to accomplish this.

## Incorporating Performance Best Practices

Driving performance best practices within day-to-day engineering processes helps to ensure that overall quality remains a top priority across application development teams within the organization. To do so effectively, organizations should establish a performance excellence program that aligns the goal of releasing high-performance applications with the objectives of the business that is driving the software creation. This practice helps to ensure that application performance goals are properly prioritized and aligned with the goals of the software application from the business perspective.

### Establish a Performance Excellence Program

The key goal of any performance excellence program is to drive best practices into engineering processes so that software developers remain focused on building high performance Web applications. This can be accomplished by establishing a simple process that should consider the business drivers of the software application, wrap specific metrics around application performance, and drive results through the implementation of best practices. This process can be broken down into these five steps.

**Establish usage scenarios and priorities** This practice will help to prioritize the scenarios that are important to the success of a particular Web application. The goal of this step is to determine the most important usage scenarios for a particular application so that performance improvement efforts are prioritized appropriately. Application developers are likely to rely on program managers or business partners to help identify and prioritize the specific usage scenarios. Customer feedback will also provide valuable insight into scenario prioritization.

**Analyze competition** To understand how to set adequate performance goals for the previously established usage scenarios, application developers must also understand the performance of competitive applications for similar scenarios. This not only helps to establish a baseline understanding of what users may expect from your application's performance, but it also helps to learn how your competition measures up. The simplest way to accomplish this is to utilize the tools we discussed earlier in this chapter.

**Set performance goals** Once the scenarios and competition are well understood, the next step is to establish goals that will help guide your engineering efforts. These goals can be as simple or as complex as you choose, but it is recommended that goals be aligned with some of the key metrics we have discussed in this chapter, including but not limited to total byte size of the page, time to load, number of files downloaded, and number of HTTP requests.

**Implement best practices** To achieve the goals that were previously established, application developers should minimally ensure that the aforementioned best practices have been implemented where applicable. The specific practices applied will very likely vary by application as not all recommendations will be applicable. The application development project team should determine which best practices are likely to provide the greatest benefit to their application and incorporate them. This chapter has provided clear guidance on how to accomplish this.

**Measure and test** After goals are established and best practices are implemented, the key is to continuously measure and test your Web applications to ensure that they are both adhering to performance best practices and are meeting the previously established performance goals of the software. This is perhaps one of the most important steps in the process because it focuses on ensuring that the quality of the application performance remains high. Organizations should focus on continuous performance testing in the same vain that they focus on testing other aspects of application quality. While performance bugs may not be as evident as other bugs, they have an equally negative impact on users and should be avoided prior to releasing the application.

Once established, a performance excellence program will ensure that proper focus is always given to this engineering tenet during the application development life cycle. This should be accomplished by establishing a process that considers the business drivers of the software application, incorporates specific metrics around application performance, and drives results through the implementation of best practices and continuous testing. In the next section, we will explore how Microsoft's Live Search team leverages some of the aforementioned processes and practices and ensures continued excellence in the performance of its products.

## Inside Microsoft: Tackling Live Search Performance

Live Search is a Web-based search engine launched by Microsoft in September of 2006. The application offers users the ability to search a variety of different types of information, including but not limited to Web sites, news, images, videos, music, and maps. The team recognizes that, to be competitive in the search market, a well-performing product is a necessity to winning with users who have come to expect near immediate results from the competition. This passion and commitment is evident in the way the team approaches the performance of the application; however, this was not always the case.

Shortly after Live Search launched in 2006, the team realized that its performance was suboptimal. Although team members had spent a lot of time testing the application, they did not adequately test the page rendering performance and subsequently released a product that did not perform as well as they would have liked. Fortunately, these issues were quickly recognized, and the team began making changes to improve the performance of the site.

The team realized that several of the application's performance issues were related to user interface and architectural design decisions whose implications were not fully understood. The application's design was promptly re-evaluated, and the team moved to implement many of the best practices mentioned in this chapter, including specific practices like combining scripts and redesigning the page to reduce the size and number of images. The results of the team's efforts included an equally attractive but improved page loading experience for the users, as well as a newly discovered dedication to performance for the team. Eventually, the Live Search team became one of the most performance-focused Web application teams within Microsoft, and its focus is evident in the way it incorporates performance best practices into its end-to-end application development process.

### Web Performance Principles

The Live Search engineering team has developed a great deal of experience over the years in the delivery of high performance Web applications to a global audience. As a team, it recognizes the depth and complexity of the engineering challenges it faces and has spent many release cycles perfecting its practices. The team adheres to a set of guiding principles that govern how it considers performance when delivering its software application to market. These principles include:

**Set performance budgets for key scenarios** The team believes strongly in setting budgets for certain page load characteristics like the number of get requests, the time to load the page, or the total byte size of the page. These goals help to drive rigorous application design practices or feature tradeoffs to ensure that pages that enable key usage scenarios continue to perform well. Oftentimes these budgets require very creative design tradeoffs that could change the way a specific feature is developed. The team believes, though, that this principle is the first line of defense against developing poorly performing Web application pages.

**Continuously analyze and test application performance** In conjunction with setting goals and budgets for page load characteristics, the team also believes strongly in running performance test cases before features are checked into the source library and after features are complete. While this practice does not necessarily prevent features from being checked in that exhaust the budget, it does introduce a certain rigor into the development process that provides early insight into poorly performing code changes. Additionally, this practice also ensures that performance bugs are being logged early and often so that developers have time to address the issues with the code before the application is released.

**Experiment and understand user behavior** Experimentation, or A/B testing, is a more advanced approach to understanding user behavior on Web sites. It generally requires a mechanism that allows certain features or application changes to be released to a small subset of users so that behavior can be observed through instrumentation and used to drive feature decisions. The Live Search team has leveraged this methodology to increase its understanding of how performance affects user behavior. The team subsequently incorporates the knowledge gained from these experiments back into the features of the application. The team has used this approach to learn the impact of page size, load times, and even the number of search results displayed to the user. While this requires an investment in a mechanism to enable this type of testing, the results are clearly valuable to product development and improvement.

**Understand usage patterns and optimize performance accordingly** The team has learned that, if it can anticipate user behaviors, it can improve the performance of the pages that the user subsequently visits. The team accomplishes this by preemptively downloading scripts asynchronously prior to a user actually visiting the page that requires those scripts. For example, if usage data indicates that a user who wishes to search for images will generally want to preview those images before clicking on them, then the application is built to proactively download the scripts required to render the image preview, before the user even gets to that page. This approach does not interfere with the use of the initial page, and it speeds up the loading of the subsequent page, which is beneficial to the user. Although not listed as a best practice, this approach clearly demonstrates a certain creativity and level of dedication to ensuring application performance is maximized for the user.

## Key Success Factors

Since becoming keenly focused on application performance, the Live Search team has found the above-mentioned set of principles to have had a very positive impact on the quality of the code and the overall application performance. These principles collectively have helped the team to incorporate performance excellence into its engineering processes and continuously innovate on its services while still achieving a high level of quality and performance. Although the team continues to learn about the usage of its application and how best to optimize performance for its key usage scenarios, it has found the following lessons and practices have yielded the best results.

**Understand end-user perceived performance** The Live Search team has learned that poor end-user perceived performance has little to do with server latency or server health but rather the number of get requests, the number of serialized get requests, and the way in which the user receives the page. Therefore, the team spends a lot of time and energy optimizing the way in which it delivers the pages to the users and less time worrying about how quickly the server is processing the page request.

**Incorporating performance test tools** As previously mentioned, the team has incorporated performance analysis and testing into several different places within the engineering process. To enable that testing, the team built a number of custom test tools that leverage applications like Network Monitor, Fiddler, HTTPWatch, Firebug, and others to monitor certain page load characteristics in its development and production environments. These tools continuously evaluate the application and ensure that bugs get logged when issues are discovered and appropriately assigned to developers to address.

**Learn and live the best practices** The team strongly believes that application developers should learn and incorporate the performance best practices whenever possible. More specifically, the team believes that the most impactful changes that can be made to any Web application include making fewer requests, consolidating scripts, reducing image sizes, using CSS Sprites, enabling HTTP compression, and incorporating edge caching using a CDN.

The Live Search team clearly believes strongly in the importance of incorporating performance excellence and best practices into its engineering processes. This is evident from both the way the team governs its engineering processes with respect to performance as well as the way the application performs in the production environment. The team continues to raise the bar with respect to Web application performance best practices among all Web-focused teams within Microsoft.

## Summary

As we have discussed in this chapter, performance is a critically important aspect of any application and represents yet another facet of the overall quality of software applications. Developers must understand common Web performance problems, their respective mitigation strategies, and the importance of establishing and maintaining a performance excellence program within their day-to-day software engineering processes. Incorporating these processes and best practices into the application development life cycle will definitely yield higher quality, better performing user experiences for any Web-based software application.

## Key Points

- Understand common Web performance challenges.
- Analyze and evaluate your application’s performance.
- Apply the key Web performance best practices.
- Establish a performance excellence program within your organization.
    - Analyze the performance of your competition.
    - Set performance goals.
    - Implement performance best practices during application or feature design.
    - Continuously analyze, test, and improve performance.

Chapter 7
# Managed Memory Model

*The first rule of management is delegation. Don't try and do everything yourself, because you can't.*

*—Anthea Turner*

In managed code, garbage collection is delegated to the Common Language Runtime (CLR). The Garbage Collector (GC) is a component of the CLR and responsible for managing managed memory. This chapter is a practical discussion of the Garbage Collector and the memory mode of the .NET Framework. In managed code, memory is allocated on demand for dynamic objects with the new operator. However, the Garbage Collector is responsible for freeing the memory for that object when necessary. There is no *delete* operator as in the C++ language.

In C++, the developer was responsible for managing dynamic memory. Dynamic memory is allocated at run time. The *new* and *delete* operators exist for this reason. The *new* operator allocates memory, while the *delete* operator frees memory. There are also advanced techniques for allocating dynamic memory in native code. *HeapCreate*, *HeapAlloc*, *HeapFree*, *HeapDestroy*, and related functions are used to create and obtain memory from a native heap. To access virtual memory directly, there is *VirtualAllocEx* and *VirtualFreeEx*. For memory mapped files, the application programming interfaces (APIs) *CreateFile*, *CreateFileMapping*, and *MapViewOfFileEx* are available. Pointers are the common thread through the various options to allocate memory at run time. Historically, mismanagement of pointers has been the reason for untold problems, such as memory leaks and memory corruption. The C++ dynamic memory model overly involved the developer. The primary goal of the developer is to create a solution to a problem, not to manage pointers. Managed code allows developers to focus on solving problems rather than on the intricacies of memory management.

Memory management for managed code is the responsibility of the developer and the Garbage Collector. The developer is responsible for allocating objects, while the Garbage Collector is responsible for freeing objects. When objects are created at run time, they reside on the managed heap. You refer to an object with a reference, which is an abstraction of a pointer. The reference abstracts the developer writing managed code from managing pointers, which prevents pointer-related problems. In this way, the developer delegates to the CLR and the Garbage Collector to manage the managed heap and pointers. This delegation is an important shift in responsibility in the managed environment from the native environment.

Two basic assumptions dominate the memory management model of .NET. Large objects are long-lived objects. Similarly sized objects are more likely to communicate with each other.

For these reasons, the Garbage Collector uses a concept called generations to group objects by size and age. As a practice, architect your program to match these assumptions. This will adversely affect the performance of garbage collection and, consequently, your application.

Memory utilization in .NET revolves around the managed heap. When you allocate a new object, it is placed on the managed heap. When unreachable, the Garbage Collector will free that object. That will reclaim the memory for that object on the managed heap.

## Managed Heap

The managed heap is partitioned into generations and the Large Object Heap. Generations 0, 1, and 2 are used to group objects by size and age. The ephemeral generations exclude the oldest generation. At the moment, the ephemeral generations include Generations 0 and 1. The reason for the distinction is that the ephemeral generations and the oldest generation sometimes can behave differently. Generation 0 is the smallest generation, Generation 1 is medium sized, while Generation 2 is the largest. For this reason, it is more likely that larger objects will appear in Generations 1 and 2. Generation 0 is simply not large enough to hold many larger objects.

The managed heap is partitioned into large objects and everything else. Large objects are greater than 85,000 bytes (85 KB) and reside on the Large Object Heap. This is not documented and is subject to change. Everything else resides on Generation 0, 1, or 2.

The Garbage Collector is responsible for freeing memory during a garbage collection. There are three events that initiate garbage collection. First is an allocation that, if successful, would exceed the memory threshold of Generation 0. Objects are always allocated to Generation 0. You cannot directly place an object on Generation 1 or 2. Because objects always start their life at Generation 0, it holds the youngest objects. Second is allocating a large object when there is insufficient memory available on the large object heap. The third event is calling the *GC.Collect* method. This will force garbage collection on demand.

The Garbage Collector collects the generations in order: Generation 0, 1, and then 2. Whenever a generation is collected, the younger generations of that generation are also collected. If Generation 1 is collected, then Generation 0 is also collected. Collecting Generation 2, which is considered a full collection, will also collect the ephemeral generations. This approach means that younger generations are collected more frequently than the older generations. This is designed for efficiency since the older objects tend to reside on the larger generations. Collecting, reclaiming, and compacting the memory for larger generations is more costly than for smaller generations. Another advantage to this model is the ability to collect a portion of the heap. Partitioning the managed heap generation supports this behavior. You can collect one or more generations and avoid a full collection of the managed heap, which is, naturally, expensive.

## Garbage Collection

Natural garbage collection in the managed environment is non-deterministic. It occurs at some point in time and is not entirely predictable. Natural garbage collection is not forced with a call to *GC.Collect*.

When does natural garbage collection occur? Allocations for new objects are added to Generation 0. If that addition exceeds the threshold for Generation 0, garbage collection occurs. The Garbage Collector will attempt to reclaim enough memory from Generation 0 to support the new allocation. If enough memory is not reclaimed, Generation 1 is collected, and then, if necessary, Generation 2. Objects surviving a garbage collection are promoted to the next generation. For example, surviving objects on Generation 0 are then promoted to Generation 1 after garbage collection. This means that older objects tend to migrate to Generation 2. This furthers the policy of grouping objects by age.

The Garbage Collector manages each generation similar to a stack. This makes allocations both quick and efficient. Each generation has an allocation pointer, which delineates the end of the last object and the beginning of the free space. This is where the next object will be allocated. At that time, the new object is stacked upon the previous object, and the allocation pointer is adjusted. The allocation pointer will now point to the end of the new object. For this reason, the oldest objects are at the base of the generation, while the newest objects are toward the top. See Figure 7-1.

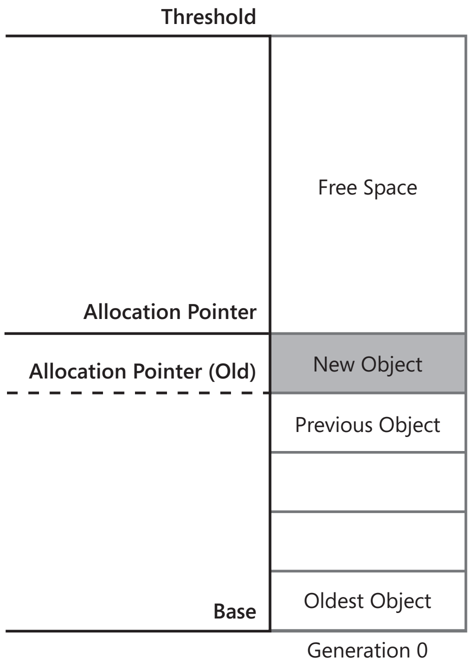


**FIGURE 7-1** An example layout of Generation 0 after a new allocation.

When garbage collection occurs, objects on the affected generations are invalidated. A memory tree is rebuilt beginning with the root objects and their object graphs. The root objects are composed of the global, static, and local variables. The object graph includes all the other objects that are referenced either directly or indirectly by the root object. Creating the memory tree marks those objects that are reachable. Objects not in the tree are considered unreachable and available for collection. Unreachable objects have no reference variable or a field referring to them. The Garbage Collector compacts the reachable objects on the managed heap. Compacting the heap prevents fragmentation and maintains the stack model.

Although unadvisable, the *GC.Collect* method of the .NET Framework Class Library (FCL) can be used to force garbage collection. The parameterless version of the function performs a full collection. The single argument version of the function targets a specific generation, which is identified by the parameter. *GC.Collect* can interfere with the normal practice of the Garbage Collector. First, forced garbage collection is expensive. Second, calling *GC.Collect* frequently can harm the performance of your application.

## Managed Wrappers for Native Objects

Managed classes sometimes wrap native objects. The managed class is an interface between the managed application and the native resource. In this way, the managed class abstracts the native resource. There are plenty of examples of this in the .NET Framework Class Library: the *FileStream* class abstracts a native file, the *Socket* class abstracts the Berkeley sockets interface, the *Bitmap* class abstracts a bitmap, and so on.

Problems can occur when there is a disparity between the size of the managed class and the native resource that it represents. For example, a managed wrapper could be a few kilobytes in size, while the native resource represented by the wrapper is several megabytes in size. The Garbage Collector will track the memory for the managed wrapper. However, the memory for the native resource is unseen. You could have plenty of managed memory available, while unknowingly running out of native memory. This creates a situation where an application crashes for lack of memory, while the Garbage Collector believes there is plenty. Native memory is the invisible elephant in the room. As instances of the manager wrapper are allocated, the elephant is getting bigger, while the room appears nearly empty.

The *GC.AddMemoryPressure* and *GC.RemoveMemoryPressure* methods help the Garbage Collector account for native memory. This is especially useful for classes that wrap heavy native resources. *GC.AddMemoryPressure* applies artificial memory pressure to the managed heap, while *GC.RemoveMemoryPressure* reduces memory pressure. Each method has a single parameter, which is the amount (bytes) of pressure to apply or relieve. In the constructor for the wrapper class, call *GC.AddMemoryPressure* and apply memory pressure equal to the amount of native memory required for the native resource. This will force additional garbage collections, where instances of the wrapper object and native resource can be released. In the *Finalize* or *Dispose* method, call *GC.RemoveMemoryPressure* to remove the additional pressure.

The following class demonstrates the proper way to implement a managed wrapper for a native resource that uses a disproportional amount of native memory.

```
public class Elephant
{
    public Elephant()
    {
      // Obtain native resource and allocate native memory

      GC.AddMemoryPressure(100000);
    }

    ~Elephant()
    {
      // Release native resource and associated memory

      GC.RemoveMemoryPressure(100000);
    }
}
```

## GC Class

The GC class, which is in the System namespace, is an interface between the user and the Garbage Collector. Table 7-1 lists each method with a description.

**TABLE 7-1 GC Methods**

| GC Method | Description |
|---|---|
| *GC.Collect* | Forces a garbage collection cycle. The default *GC.Collect* forces a full garbage collection, which is essentially Generation 2. For a more granular garbage collection, use the one-parameter *GC.Collect* method. The parameter stipulates the generation that should be collected (i.e., 0, 1, or 2). |
| *GC.WaitForPendingFinalizers* | Suspends the current thread until the finalization thread has called the finalizers of the objects waiting on the *FReachable* queue. Call this method after *GC.Collect* to provide ample time for the finalization thread to finish its work before the current thread resumes. |
| *GC.KeepAlive* | Keeps an otherwise unreachable object from being collected during the next garbage collection cycle. |
| *GC.SuppressFinalize* | Removes a reference to a finalizable object from the Finalization queue. Remaining overhead related to the finalizer is avoided. *GC.SuppressFinalize* is usually called in the *Dispose* method. Because the object has been disposed, finalization is no longer required. |

| GC Method | Description |
|---|---|
| *GC.AddMemoryPressure* | Applies additional memory pressure to the managed heap. This is typically used to compensate for native resources in managed code. |
| *GC.RemoveMemoryPressure* | Removes memory pressure from the managed heap. Like *GC.AddMemoryPressure*, this is typically used to compensate for native resources in managed code. |
| *GC.CollectionCount* | Returns the number of times garbage collection has occurred for the specified generation. |
| *GC.GetGeneration* | Returns the generation of the provided object. |
| *GC.GetTotalMemory* | Returns the number of bytes allocated on the managed heap. |
| *GC.ReRegisterForFinalize* | Reattaches a finalizer to an object. This is usually called on objects that have been resurrected to assure proper finalization. |
| *GC.RegisterForFullGCNotification* | Registers the application to be notified when a full collection is likely to happen and after it has occurred. |
| *GC.CancelFullGCNotification* | Unregisters the application from receiving notifications about impending full garbage collections. |
| *GC.WaitForFullGCApproach* | Notifies an application if a full garbage collection is impending. |
| *GC.WaitForFullGCComplete* | Notifies an application that a full garbage collection has completed. |

## Large Object Heap

The Large Object Heap holds large objects. Most large objects are arrays rather than the assemblage of non-array members of a class. Larger objects are longer lived and typically migrate to Generation 2. Promoting large objects from Generation 0 and eventually to Generation 2 is expensive. Placing really large objects immediately on the Large Object Heap is much more efficient. The Large Object Heap is collected during a full garbage collection, which is Generation 2. During garbage collection, memory for large objects on the Large Object Heap is freed. However, the Large Object Heap is never compacted. Sweeping and consolidating large objects on the Large Object Heap would be expensive. Therefore, that step is skipped. Garbage collection for the Large Object Heap entails these steps:

- Memory for unreachable objects is released.
- Memory from adjacent and unreachable objects is combined into a free block.
- Memory for unreachable objects at the end of the Large Object Heap is released back to Windows.

Because the Large Object Heap cannot be compacted, it can become fragmented. Allocating and releasing disparate-sized large objects on the Large Object Heap makes fragmentation more likely. You are unable to place large objects in the free space from unreachable smaller large objects—unless combined with contiguous space from another free object. The Garbage Collector is forced to search the individual free spans for holes large enough for the pending allocation. Collectively, the free spaces of the Large Object Heap may have enough memory to honor the request but not in a contiguous area.

If you use disparate-sized objects, one possible solution is a buffer of like-sized large objects that can be reused. This keeps the large objects in contiguous memory and could prove to be more efficient. You conserve memory, when the number of instances would otherwise exceed the pool, minimize fragmentation, and reduce the number of full collection operations. Full collections are especially expensive. The downside is when the simultaneous instances are consistently less than the size of the pool. That would waste memory resources and require fine-tuning the pool.

The following code demonstrates how to create and manage a buffer of large objects. In our example, the buffer contains 10 large objects, as shown below.

```
        static BigObject[] bigobjects = {  new BigObject(),
                                           new BigObject(),
                                           new BigObject(),
                                           new BigObject(),
                                           new BigObject(),
                                           new BigObject(),
                                           new BigObject(),
                                           new BigObject(),
                                           new BigObject(),
                                           new BigObject()};
```

The *BigObject* class below contains a byte array of 200,000 elements. For this reason, the byte array but not the *BigObject* class is placed on the Large Object Heap. The code for the class is minimally implemented because the concepts are simple. If an object in the buffer is available for use, the *bAvailable* field is set to *true*. The *Initialize* method initializes an object and makes the status available. The *Reset* method is called to reset an object from the object pool that is already being used. The reinitialized object is then returned.

```
    public class BigObject
    {
        // other data

        public void Initialize()
        {
            // perform initialization
            bAvailable = true;
        }

        public BigObject Reset()
        {
            Initialize();
            bAvailable = false;
            return this;
        }

        public void Update()
        {
        }

        public bool bAvailabled=true;
        byte[] data = new byte[200000];    }
```

I run the application and create 15 objects. This exceeds the pool limit. Therefore, 10 objects are actually created. The additional five objects reuse objects that are already in the pool. Using Windbg, I have listed instances of the byte array. Windbg is a debugging tool that is discussed more thoroughly in Chapter 9, "Debugging." In the following listing, *MT* refers to the method table of a class. A method table is an array of methods that belong to a particular class. Instances of the same type share the same method table. For this reason, you can list all instances of the same type from the address of the method table. In this way, the method table is more of a cookie of a particular type of object than an address. They are shown in bold in the following listing. As expected, there are exactly 10 instances of the large byte array, not 15. Five of the instances reuse large objects from the object pool.

```
!dumpheap -mt 7912dae8
 Address       MT     Size
014aad34 7912dae8     1036
014ab140 7912dae8     1036
014ab54c 7912dae8     1036
014ab958 7912dae8     1036
02486bc0 7912dae8   200016
024b7920 7912dae8   200016
024e8680 7912dae8   200016
025193e0 7912dae8   200016
0254a140 7912dae8   200016
0257aea0 7912dae8   200016
025abc00 7912dae8   200016
025dc960 7912dae8   200016
0260d6c0 7912dae8   200016
0263e420 7912dae8   200016
```

## Finalization

Finalization occurs during garbage collection. The finalizer is invoked during finalization to clean up resources related to the object. Non-deterministic garbage collection is performed on a generation or Large Object Heap when the related threshold is exceeded. Because of this, there may be some latency between when an object becomes unreachable and when the *Finalize* method is called. This may cause some resource contention. For example, the action of closing a file in the *Finalize* method may not occur immediately. This may cause resource contention because, although the file is not being used, it remains unavailable for a period of time.

### Non-Deterministic Garbage Collection

Place cleanup code for the non-deterministic garbage collection in the *Finalize* method, which is implicitly called in the class destructor. The class destructor cannot be called on demand. As mentioned, there may be some latency in the *Finalize* method running. The class destructor is the method of the same name of class with a tilde (~) prefix.

```
class XClass {
    // destructor
    ~XClass() {
        // cleanup code
    }
}
```

For certain types of resources, non-deterministic garbage collection is inappropriate. You should not release resources that require immediate cleanup. Also, managed objects should not be cleaned up in a *Finalize* method. Order of finalization is not guaranteed. Therefore, you cannot assume that any other managed object has not been already finalized. If that has occurred, referring to that object could raise an exception.

Non-deterministic garbage collection is neither simple nor inexpensive. The lifetime of objects without a *Finalize* method is simpler. For these reasons, the *Finalize* method should be avoided unless necessary. Even an empty destructor (which calls the *Finalize* method), harmless in C++, enlists the object for the complete non-deterministic ride—a very expensive ride. For the purposes of this chapter, objects with destructors are called finalizable objects.

The additional cost of having a *Finalize* method begins at startup. At startup, objects with a *Finalize* method have a reference placed on the Finalization queue. This means, when the object is otherwise unreachable, there is an outstanding reference being held on the queue, which will prevent immediate garbage collection.

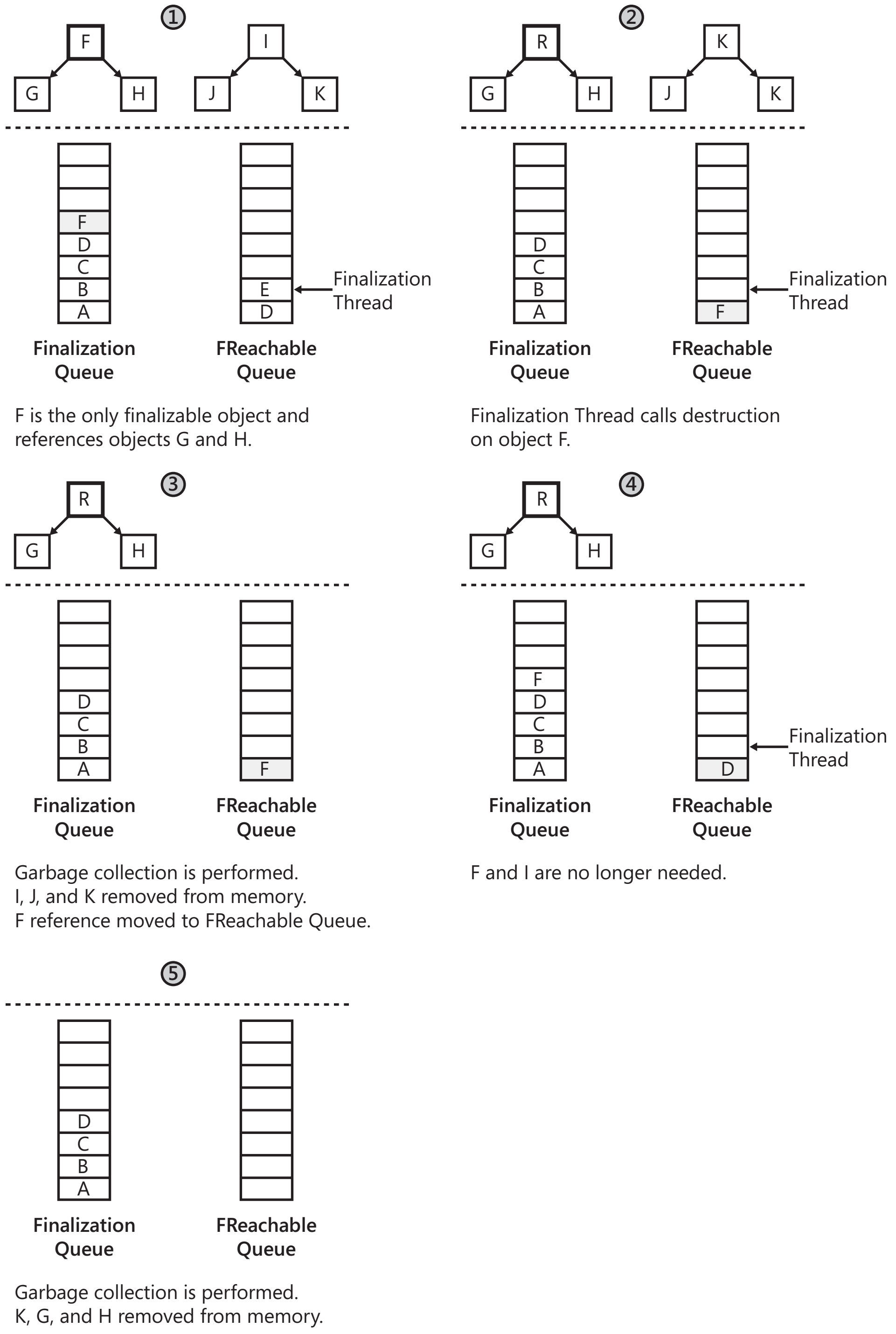
When the finalizable object is no longer reachable and there is a garbage collection event, the object is not removed from memory. At this time, a normal object that is unreachable would be removed from memory. However, the finalizable object is moved from the Finalization to FReachable queue. This keeps the finalizable object in memory. The current garbage collection cycle then ends.

The FReachable queue holds finalizable objects that are waiting for their *Finalize* methods to be called. Finalizer thread is a dedicated thread that services the FReachable queue. It calls the *Finalize* method on the finalizable objects. After the *Finalize* method is called, the reference to the finalizable object is removed from the FReachable queue. At that time, there are no outstanding references to the finalizable object.

During the next garbage collection, finalizable objects that have been removed from the FReachable queue can finally be removed from memory at the next garbage collection cycle. Unreachable normal objects are removed in one garbage collection cycle. However, unreachable finalizable objects require at least two garbage collection cycles. This is part of the expense of using finalizable objects. Finalizable objects should not have a deep object graph. The finalizable object is kept not only in memory but also in any object it references.

Figure 7-2 shows the garbage collection cycle for two groups of objects. F is a finalizable object that references objects G and H. I is a non-finalizable object that references J and K. G, H, J, and K are non-finalizable objects.

The *IDisposable.Dispose* method is an alternative to the *Finalize* method in non-deterministic garbage collection. Contrary to the *Finalize* method, *IDisposable.Dispose* is deterministic, called on demand, and has no latency.



F is the only finalizable object and references objects G and H.

Finalization Thread calls destruction on object F.

Garbage collection is performed.
I, J, and K removed from memory.
F reference moved to FReachable Queue.

F and I are no longer needed.

Garbage collection is performed.
K, G, and H removed from memory.

**FIGURE 7-2** Garbage collection cycle for finalizable object.

## Disposable Objects

Disposable objects implement the *IDisposable* interface, which has a single method—*Dispose*. The *Dispose* method is called deterministic or on demand. In the *Dispose* method, you can clean up for resources used by the object. Unlike the *Finalize* method, both the managed and unmanaged resources can be referenced in the *Dispose* method. Because the *Dispose* method is called on demand, you know the sequence of cleanup. Therefore, you know which objects have been previously cleaned up or not.

You can implement the *Dispose* method without inheriting the *IDisposable* interface. That is not the same as implementing the *IDisposable* interface and the resulting object is not a disposable object. The *IDisposable* interface is a marker indicating that the object is disposable. The Framework Class Library (FCL) relies on this marker to automatically call the *Dispose* method. For example, some .NET collections detect disposable objects to perform proper cleanup when the collection is disposed.

The method name *Dispose* is not the most transparent in all circumstances. Long-standing terminology or domain-specific phraseology may dictate using a different term. Most frequently, the alternate method name is *Close*. Whatever name is chosen, the method should delegate to the *Dispose* method. The user had the option to use the alternate name or the standard name, which is *Dispose*. The *File.Close* method is an example of using a different method name for deterministic cleanup. Avoid using alternate names for disposal unless there is a close affinity of the term with that type of object.

You can implement both the *Finalize* method for non-deterministic garbage collection and the *Dispose* method. Because you can clean up both managed and unmanaged resources there, the implementation of the *Dispose* method is usually a superset of the *Finalize* method. The *Finalize* method is limited to cleaning up unmanaged resources. In the *Dispose* method, call the *GC.SuppressFinalize* method. This method will remove the reference to the current object from the Finalization queue to avoid further overhead related to finalization. When both are implemented, the *Finalize* method is essentially a safety net if the *Dispose* method is not called.

To avoid inadvertently not calling the *Dispose* method on a disposable object, employ the *using* statement. Disposable objects defined in the using statement are automatically disposed of at the end of the *using* block. The *Dispose* method is called on those objects as the *using* block is exited. See the following code. In this code, *obj1* is a disposable object. The *Dispose* method is called after the *using* block is exited.

```
using( XClass obj1) {
}
// obj1 disposed.
```

Next is a more complex *using* statement. You can list more than one disposable object within a comma-delimited list in the *using* statement. Within a single *using* statement, you can define multiple instances of the same type. For disposing different types, precede the *using* block with more than one *using* statement—one for each type. In the following code, there are two *using* statements. There is one *using* statement for the *XClass* type, while the other is for the *YClass*. In total, three instances are defined. The *Dispose* method of the three objects is automatically called at the end of the *using* block.

```
using( XClass obj1=new XClass(),
              obj2=new XClass())
using (YClass obj3 = new YClass())
{

}
// Dispose method called on obj1, obj2, and obj3
```

## Dispose Pattern

Implementing proper disposal in a managed class can be non-trivial. When there is a base and derived types that are both disposable, the implementation can be even more complex. The dispose pattern is more than a pattern for implementing the *Dispose* method. It is the best practice for implementing deterministic and non-deterministic behavior and cleanup for a base and derived class. This relationship must be considered to implement the proper cleanup behavior. For easier understanding, the base and derive class implementation of the dispose pattern are presented separately in this chapter.

The dispose pattern has four primary goals: correctness, efficiency, robustness, and code reuse. Correctness is the goal for every pattern. The dispose pattern is the perspective from Microsoft on the correct implementation of the *Dispose* and *Finalize* methods. A disposed object is probably not immediately collected. For that reason, it remains available to the application. You should be able to call the *Dispose* method and other methods on a disposed object with predictable results. The dispose pattern provides robust behavior for disposed objects. The dispose pattern is refactored for code reuse to prevent redundant code. Redundant code is hard to maintain, and it is a place where problems can flourish.

The base class (*XParent*) implementation for the dispose pattern is as follows:

- In the dispose pattern, the base class implements two *Dispose* methods. The protected *Dispose* method performs the actual resource cleanup. At the start of the method, a flag (disposed) is set to indicate that the object is disposed. The only parameter (disposing) indicates whether the cleanup is deterministic or non-deterministic. If disposing is true, it is deterministic and the *Dispose* has been called programmatically. You can clean up both managed and unmanaged resources. If false, you are restricted to the cleanup of unmanaged resources.
- The second *Dispose* method, which is part of the public interface for the class, is called to initiate deterministic cleanup. It delegates to the one-parameter *Dispose* method to

perform the actual cleanup. The parameter is set to *true* to indicate deterministic garbage collection. Because the cleanup has been performed, the *Dispose* method invokes *GC.SuppressFinalize* and removes a reference to a disposed object from the Finalization queue. This prevents further costs from finalization.

- *BaseFunction* represents any method of the class. Methods of a disposable object should be callable even after the object is disposed. In the method, check if the object is disposed first. If so, throw the object-disposed exception. This is demonstrated in *BaseFunction*.
- The base class destructor (*~XParent*) delegates to the one-parameter *Dispose* method also. However, the parameter is false to indicate non-deterministic garbage collection.

```
// Base class

public class XParent: IDisposable {

    // Deterministic garbage collection

    public void Dispose() {

        // if object disposed, throw exception.

        if (disposed) {
            throw new ObjectDisposedException("XParent");
        }

        // Call the general Dispose routine

        Dispose(true);

        // Collection already performed. Suppress further finalization.

        GC.SuppressFinalize(this);
    }

    // dispose property true if object has been disposed.

    protected bool disposed = false;

    // Deterministic and non-determenistic garbage collection
    // disposing parameter = true ( determinstic )
    //                       false (non-deterministic)

    protected virtual void Dispose(bool disposing) {

        disposed = true;

        if (disposing) {

            // if deterministic garbage collection, cleanup
            // managed resources.
        }
```

```
            // cleanup unmanaged resources.
        }

        // Representative of any base class method

        public void BaseFunction() {

            // if object disposed, throw exception.

            if (disposed) {
                throw new ObjectDisposedException("XParent");
            }

            // implement method behavior
        }

        // Non-deterministic garbage collection

        ~XParent() {

            // Call the general Dispose routine

            Dispose (false);
        }
    }
```

The child class (*XChild*) implementation is as follows.

- The child class inherits the public *Dispose* method (parameterless) and the disposed property.
- The one-parameter *Dispose* method is overriden in the child class to clean up for child resources. The overriden function is almost identical to the version in the parent. The only other difference is that this version calls the base class *Dispose* method. This affords the base class an opportunity to clean up for its resources.
- *DerivedFunction* represents any method of the child class. In the method, you must check whether the object is disposed. If so, throw the *object-disposed* exception.
- The child class destructor (*~XChild*) delegates to the one-parameter *Dispose* method for proper cleanup.

```
    // Derived class

    public class XDerived: XParent {

        // Deterministic and non-determenistic garbage collection
        // disposing parameter = true ( determinstic )
        //                       false (non-deterministic)

        protected override void Dispose(bool disposing) {
            disposed = true;
            if (disposing)
            {
                // if deterministic garbage collection, cleanup
                // managed resources.
            }

            // Call base class Dispose method for base class cleanup.

            base.Dispose(disposing);

            // cleanup unmanaged resources of derived class.

        }

        // Representative of any derived class method

        public void DerivedFunction() {

            // if object disposed, throw exception.

            if (disposed) {
                throw new ObjectDisposedException("XChild");
            }
            // implement method behavior
        }

        // Non-deterministic garbage collection

        ~XDerived(){

            // Call the general Dispose routine

            Dispose(false);
        }

    }
```

## Weak References

There are strong and weak references. Until now, this chapter has focused on strong references. Both weak and strong references are created with the new operator. The difference is how a weak reference is collected unlike a strong reference. A strong reference cannot be collected unless unreachable. This is within the control of the application and not the Garbage Collector. A weak reference, unlike a strong reference, can be collected at the discretion of the Garbage Collector.

In managed code, strong references are the default reference. There is a strong commitment from the Garbage Collector to keep the associated object in memory—no exceptions or flexibility. Conversely, a weak reference has a weak commitment from the Garbage Collector. The Garbage Collector has the flexibility to remove the weakly referenced object from the managed heap when memory stress is applied to the application and more memory is needed.

Weak references represent the best of both worlds. Both the application and the Garbage Collector can access the weakly referenced object. If not collected, the application can continue to use the object referenced by the weak reference. In addition, the Garbage Collector can collect the weak reference whenever needed.

Weak references are ideal for objects that require a lot of memory and are persistent in some manner. For example, you could have an application that maintains large spreadsheets that is cached to a permanent or temporary file. Large spreadsheets that consist of hundreds of rows and columns are memory intensive. Naturally, the application performance improves when the spreadsheet is memory resident. However, that applies considerable memory stress. The Garbage Collector should have the option to remove the spreadsheet object if necessary. The spreadsheet object is the perfect candidate for a weak reference. This would keep the spreadsheet object in memory, and accessible by the application, but also collectible by the Garbage Collector, if needed. If collected, the application could easily rehydrate the spreadsheet from the backing file.

Weak references are also ideal for maintaining caches. Cache can be memory intensive. The weak reference can be used to vary the lifetime of the cache based on a time-out, variables, or other criteria. For example, a cache may have a time-out. Before the time-out, the cache could be maintained as a strong reference. When the cache expires, it would be converted to a weak reference and be available for collection, if needed. If the cache is backed by a persistent source, such as a Microsoft SQL database, associating the cache with a weak reference is done to conserve memory resources as required.

There are two types of weak references: a short and long weak reference. A short weak reference is the default. With a short weak reference, the strong reference is released before finalization. For long weak references, the reference is tracked through finalization. More than extending the lifetime of the object reference, it allows the object to be resurrected.

Following are the steps for using a weak reference:

1. Create a strong reference.
2. Create a weak reference that is initialized with the strong reference. The default constructor creates a short weak reference.
3. Set the strong reference to null.
4. The weak reference is accessible from the *WeakReference.Target* property.
5. If the *WeakReference.Target* property is *null* and the *WeakReference.IsLive* property is *false*, the weak reference has been collected and is no longer available.
6. If the weak reference is available, assign the *WeakReference.Target* property to a strong reference, and then use the object.

7. If the weak reference is no longer available, rehydrate the data from a persistent source. When you are finishing using the updated strong reference, reinitialize a weak reference with the new strong reference.

The following code is a partial listing from an application that uses a weak reference. The program displays an array of names that is read from a persistent file. The array is assigned to a weak reference. The *hScrollBar1_Scroll* function scrolls through the names. First the function creates a strong reference. This is the *WeakReference.Target* assignment. If null, the *names* array has been collected, and the weak reference is no longer available. If that occurs, the array is rehydrated with the *GetNames* function. At the end of the function, the names reference is assigned null. This negates the strong reference, which leaves the weak reference to control the lifetime of the array.

```
 Name[] names = null;
 WeakReference wk;
 List<byte[]> data = new List<byte[]>();

 private void hScrollBar1_Scroll(object sender, ScrollEventArgs e) {
     names= (Name[]) wk.Target;
     if (null == names) {
         MessageBox.Show("Rehydrate");
         names=GetNames();
         wk.Target = names;
     }
     if (e.NewValue > names.Length) {
         return;
     }
     txtItem.Text = names[e.NewValue].first+" "+
         names[e.NewValue].last;
     names = null;
}
```

## Pinning

Unmanaged code expects normal pointers, which are assigned a fixed address. For example, a pointer parameter in a native function call is a fixed pointer. A reference in managed code is an abstraction of a moveable pointer. When calling a native function via interoperability, you must be careful about passing references as parameters where pointers are expected. Because the reference is movable, the native call may behave incorrectly or even crash the application. A reference can be fixed in memory, which is called pinning. The referenced object is then considered a pinned object.

Pinned pointers can interfere with normal garbage collection. The Garbage Collector cannot move the memory associated with the pinned objects on the managed heap. Therefore, the generation with the pinned object cannot be fully compacted into contiguous memory. For this reason, pinning is the exception where a generation can possibly become fragmented. Objects that would otherwise fit comfortably in the combined free space do not because of fragmentation. This translates into potentially more garbage collection, which is expensive and harms the performance of the application. Keep pinning to a minimum to avoid this behavior. If possible, pin objects for a short duration—ideally within a garbage collection cycle. This avoids most of the problems in garbage collection related to pinning.

If possible, pin older objects and not younger objects. Older objects are objects that have been promoted to Generation 2. Generation 2 is collected less frequently. Therefore, the Garbage Collector is less likely to have to work around pinned objects. Objects on Generation 0 and 1 are more volatile and move frequently. Pinning objects in these generations creates considerable more work for the Garbage Collector. If an application pins objects regularly, particularly small or young objects, create a pool of pinned objects. Fragmentation is limited because the pinned objects are in contiguous memory and not scattered about the managed heap. This will allow the Garbage Collector to compact storage into contiguous free space more effectively. Performance of the Garbage Collector and application will improve.

There are three ways to pin an object:

- During interoperability, pinning sometimes occurs automatically. For example, passing strings from managed code into a native API as a method parameter. The managed reference for the string is automatically pinned.

```
[DllImport("user32.dll", CharSet = CharSet.Auto)]
public static extern int MessageBox(IntPtr hWnd,
    [MarshalAs(UnmanagedType.LPTStr)] string text,
    [MarshalAs(UnmanagedType.LPTStr)] string caption, int options);

static void Main(string[] args) {
    string message = "Hello, world!";
    string caption = "Solid Code";

    // pinned
    MessageBox(IntPtr.Zero, message, caption, 0);
}
```

- The *fixed* statement is used to obtain a native pointer to a reference. In the *fixed* block, the reference is not moveable and the related pointer can be used.

```
public class TwoIntegers {
    public int first = 10;
    public int second = 15;
}

unsafe static void Main(string[] args) {
    TwoIntegers obj = new TwoIntegers();

    // pinned
    fixed(int *pointer=&obj.first) {
        Console.WriteLine("First ={0}", *pointer);
        Console.WriteLine("Second={0}", *(pointer+1));
    }
}
```

- You can also pin objects using the *GCHandle* type, which is part of the *System.Runtime.InteropServices* namespace. *GCHandle* holds a reference to a managed type that can be used in unmanaged code or an unsafe block. As the method name implies, *GCHandle.AddrOfPinnedObject* returns the address of the pinned object.

```
static int[] integers = new int[] { 10, 15 };
unsafe static void Main(string[] args)
{
    GCHandle handle = GCHandle.Alloc(integers, GCHandleType.Pinned);
    IntPtr ptrRef= handle.AddrOfPinnedObject();
    int *pointer=(int*)ptrRef.ToPointer();
    Console.WriteLine("First  = {0}", *pointer);
    Console.WriteLine("Second = {0}", *(pointer+1));
}
```

## Tips for the Managed Heap

These are tips for interacting with the managed heap. Some of these tips, such as avoiding the *GC.Collect* method, have been articulated previously in this chapter. However, they are included here for completeness.

- Do not program contrary to the garbage collection paradigm in the managed environment. Small objects should be short lived, while larger objects should be long lived. Objects are expected to communicate with like-sized objects.
- Avoid boxing. Frequent boxing, as occurs when using non-generic collections with value types, flood Generation 0 with small objects. This will trigger extra garbage collections.
- Because of the cost of finalization, use a *Finalize* method only when imperative. Furthermore, empty destructors are not innocuous as in C++. You still incur the full cost of finalization.
- Classes that have a *Finalize* method should not have deep object graphs. Finalizable objects are kept in memory longer than normal objects. Objects referenced by the finalizable objects are also kept in memory longer.
- If possible, do not refer to other managed objects in the *Finalize* method. First, those objects may no longer exist. Second, you may inadvertently create a back reference to yourself and resurrect the current object. Resurrected objects can be problematic.
- Define disposable objects in the using statement, which will automatically call the *Dispose* method and guarantee cleanup.
- Do not call *GC.Collect*. This is especially true for a complete garbage collection, which is expensive. Allow garbage collection to occur naturally.
- Keep short-lived objects short lived. Do not reference short-lived objects from long-lived objects. That links the lifetime of the two objects, and both are then essentially long-lived objects.

- Set objects as class members and local objects to null as early as possible. This allows them to be collected as soon as possible.
- Do not allocate objects in either hashing or comparison methods. When sorting or comparing, these methods can be called repeatedly in a short period of time. If the methods contain allocations, this could result in considerable memory pressure on the managed heap and additional garbage collection activity.
- Avoid near-large objects. These are objects that are close to 85 KB in size. As near-large objects, expect those objects to migrate to Generation 2. Add a buffer to the type and increase the near-large object to a large object. This will place the object immediately on the Large Object Heap and avoid the overhead of promoting the object through to Generation 2.
- Keep code in a *Finalize* method short. All *Finalize* methods are serviced by a separate thread—the finalizable thread. An extended *Finalize* method prevents a thread from servicing other *Finalize* methods and releasing the reference to the related object.

Even after adhering to every tip, don't be surprised to have the occasional memory problem. The CLR Profiler from Microsoft is helpful in those occasions. This tool allows developers to diagnose issues with the managed heap.

## CLR Profiler

*Look up in the sky! It's a bird! It's a plane! It's the CLR Profiler!*

*—Donis Marshall*

The CLR Profiler is an excellent diagnostic tool that monitors an executing managed application and collects data points on object allocation, the managed heap, and garbage collection. The tool is available from Microsoft. If you suspect problems related to the managed heap, the CLR Profiler is an effective tool for diagnosing and pinpointing particular issues. The results of the CLR Profiler are available in a variety of text reports and graphs (mostly histograms). In addition to specific data on the managed heap, the CLR Profiler can provide information on methods in detailed call graphs. Information can be reported during program execution and post mortem. For example, you can obtain a memory summary of the managed heap while the application is executing. Conversely, you can also get a list of objects allocated while the application was running at program completion.

I have great reverence for the CLR Profile as the previous quote would indicate. CLR Profiler is one of the best written .NET applications. The breadth of information and level of detail pertaining to the managed heap and garbage collection is invaluable:

- An easy-to-understand summary of the managed heap.
- A comprehensive overview of object allocations.

- A list of methods that allocate memory on the managed heap, which includes the percentage of allocation attributed to each method.
- A variety of call graphs.
- Ability to track the lifetime of the Garbage Collector: when garbage collection occurs, the duration between garbage collections, which objects were affected by a particular garbage collection, and more.
- A list of finalized objects.
- A wide variety of graphs that paint an accurate description of managed memory for non-developers, which is helpful for meeting with managers.

Download the current version of the CLR Profiler from the Microsoft downloads Web site: *www.microsoft.com/downloads*. You can download both the 32- and 64-bit versions of the application. Once installed, the target application can be launched from within the CLR Profiler. The CLR Profiler is intrusive and will adversely affect the performance of the application. For this reason, do not use the product in a production environment.

The CLR Profiler is a complex tool. The following walkthrough provides an introduction to the product. This is not a comprehensive review of the CLR Profiler. Refer to the reference material on the CLR Profiler from Microsoft for additional details.

## CLR Profiler Walkthrough

This walkthrough demonstrates the fundamentals of the CLR Profiler. The NoBigPool and BigPool applications are used during the walkthrough. BigPool was described earlier in this chapter. The application maintains a pool of 10 large objects, which are reusable. A large object is defined as an object that resides on the Large Object Heap. The NoBigPool is identical to the BigPool application except it does not maintain a pool of large objects. We assert that BigPool is more efficient because of the pool. In the walkthrough, CLR Profiler will confirm this assertion or force me to rewrite this chapter. We will create 20 big objects. Depending on the application, this will require either releasing or reusing 10 of the big objects. CLR Profiler will allow us to compare the result of the managed heap for both applications.

Each application randomly places secondary large objects, which are increasingly larger, on the Large Object Heap. These secondary objects are occasionally freed. As mentioned previously, during a full garbage collection, the Large Object Heap is swept but not compacted. For this reason, the disparate-sized objects have the potential to slowly fragment the Large Object Heap of both the NoBigPool and BigPool applications. This is being done to simulate a normal pattern of allocation.

Start the CLR Profiler to begin the walkthrough. See Figure 7-3. The Allocations and Calls check boxes should be selected by default. If not, select them to profile the managed heap and function calls, respectively.

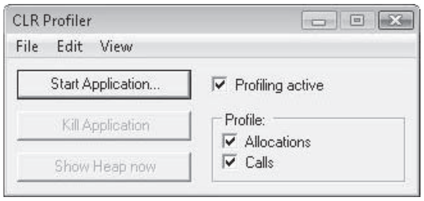


**FIGURE 7-3** CLR Profiler window.

1. We start the walkthrough by profiling the NoBigPool application. Press the Start Application button. Browse to the folder containing bigpool.exe, and select the assembly. The CLR Profiler will start the application, and the NoBigPool user interface will appear momentarily.

   The NoBigPool application is shown in Figure 7-4. The Get Large button creates a large object on the Large Object Heap. The Clear Object button sets the reference to a large object to null, which makes the object unreachable and a candidate for future garbage collection. The spin control specifies the big object to clear. Adjust the spin control before pressing the Clear Object button.

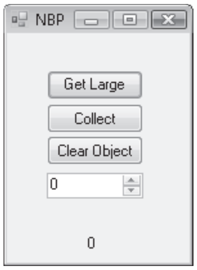
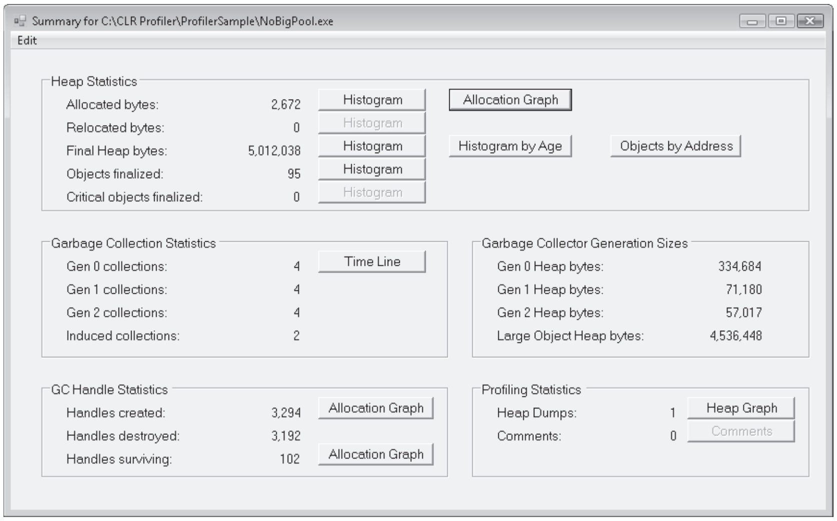
   

   **FIGURE 7-4** The user interface for the NoBigPool application.

2. For the walkthrough, create 10 large objects. Press the Get Large button 10 times. Using the spin control and the Clear Object button, clear the 10 objects. Create another 10 objects. You have now touched 20 big objects in some manner.
3. Using the CLR Profiler, we can now examine the details of the managed heap for the NoBigPool application. Click the Show Heap Now button in the CLR Profiler to collect current heap information pertaining to the application. The Heap Graph window is displayed. Close the window.
4. We are more interested in displaying a text summary of the managed heap. From the View menu, select Summary. In the Summary window, find the Garbage Collector Generation Sizes group. This is where the size of the Large Object Heap is displayed. For our example, the size of the Large Object Heap is 4.5 megabytes (MB). See Figure 7-5. This number may vary based on several factors, such as the version of the .NET Framework.

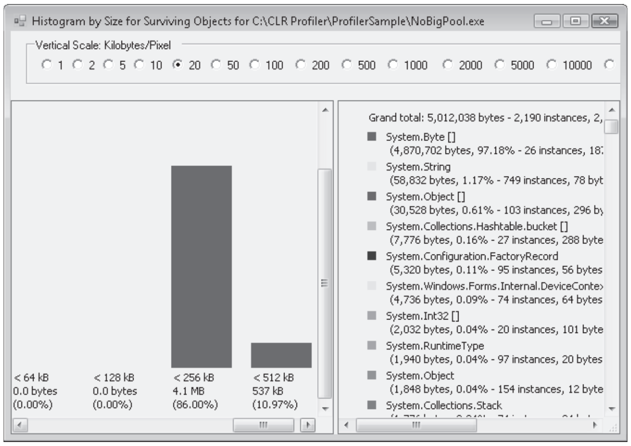


```
Summary for C:\CLR Profiler\ProfilerSample\NoBigPool.exe
Edit

Heap Statistics
  Allocated bytes:                 2,672   [Histogram]  [Allocation Graph]
  Relocated bytes:                     0   [Histogram]
  Final Heap bytes:            5,012,038   [Histogram]  [Histogram by Age]  [Objects by Address]
  Objects finalized:                  95   [Histogram]
  Critical objects finalized:          0   [Histogram]

Garbage Collection Statistics              Garbage Collector Generation Sizes
  Gen 0 collections:        4  [Time Line]   Gen 0 Heap bytes:            334,684
  Gen 1 collections:        4                Gen 1 Heap bytes:             71,180
  Gen 2 collections:        4                Gen 2 Heap bytes:             57,017
  Induced collections:      2                Large Object Heap bytes:   4,536,448

GC Handle Statistics                                 Profiling Statistics
  Handles created:     3,294  [Allocation Graph]       Heap Dumps:   1  [Heap Graph]
  Handles destroyed:   3,192                           Comments:     0  [Comments]
  Handles surviving:     102  [Allocation Graph]
```

**FIGURE 7-5** The Summary window of the managed heap for the NoBigPool application.

When the managed heap is larger than expected, the CLR Profiler offers a variety of helpful reports to diagnose the problem. For example, you can request list objects and their sizes that have been allocated. You can also view a report that lists the methods where significant allocations are occurring. The list can be sorted by total allocation per method, which is particularly helpful.

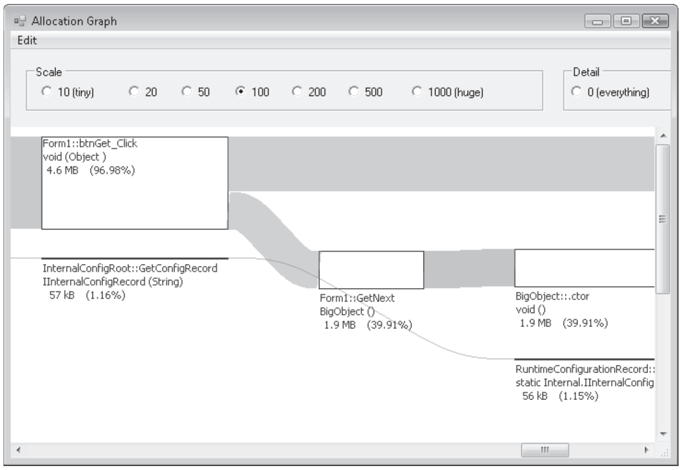
5. From the Summary window, you can display the allocated objects that are currently on the managed heap. In the Heap Statistics group of the Summary window, press the Histogram button next to the Final Heap Bytes value. The Histogram By Size For Surviving Objects window will be displayed. See Figure 7-6.



```
Histogram by Size for Surviving Objects for C:\CLR Profiler\ProfilerSample\NoBigPool.exe

Vertical Scale: Kilobytes/Pixel
( ) 1  ( ) 2  ( ) 5  ( ) 10  (•) 20  ( ) 50  ( ) 100  ( ) 200  ( ) 500  ( ) 1000  ( ) 2000  ( ) 5000  ( ) 10000

< 64 kB      < 128 kB     < 256 kB     < 512 kB
0.0 bytes    0.0 bytes    4.1 MB       537 kB
(0.00%)      (0.00%)      (86.00%)     (10.97%)

Grand total: 5,012,038 bytes - 2,190 instances, 2,
  System.Byte []
   (4,870,702 bytes, 97.18% - 26 instances, 18
  System.String
   (58,832 bytes, 1.17% - 749 instances, 78 byt
  System.Object []
   (30,528 bytes, 0.61% - 103 instances, 296 by
  System.Collections.Hashtable.bucket []
   (7,776 bytes, 0.16% - 27 instances, 288 byte
  System.Configuration.FactoryRecord
   (5,320 bytes, 0.11% - 95 instances, 56 bytes
  System.Windows.Forms.Internal.DeviceConte
   (4,736 bytes, 0.09% - 74 instances, 64 bytes
  System.Int32 []
   (2,032 bytes, 0.04% - 20 instances, 101 byte
  System.RuntimeType
   (1,940 bytes, 0.04% - 97 instances, 20 bytes
  System.Object
   (1,848 bytes, 0.04% - 154 instances, 12 byte
  System.Collections.Stack
```

**FIGURE 7-6** The Histogram By Size For Surviving Objects window.

6. The Histogram By Size For Surviving Objects window is separated into two panes. The left pane displays a graph of allocated objects—grouped by size. Scroll the pane right to displayed larger objects, such as objects that are on the Large Object Heap. The right pane is both a legend for the left pane and a sequential listing (descending order) of types that are on the managed heap. In our example, *System.Byte* arrays account for almost 97 percent of the allocated memory, which is worth further investigating. It would be helpful to know where *System.Byte* arrays are being allocated. That would be an important first step in diagnosing a potential problem. In the right pane, open a context menu (right-click) for the *System.Byte* array item in the legend. Select Show Who Allocated from the menu. An Allocation Graph is displayed, as shown in Figure 7-7.



Scale: ( ) 10 (tiny) ( ) 20 ( ) 50 [✓ 100] ( ) 200 ( ) 500 ( ) 1000 (huge)

Detail: ( ) 0 (everything)

```
Form1::btnGet_Click
void (Object )
 4.6 MB   (96.98%)

InternalConfigRoot::GetConfigRecord
IInternalConfigRecord (String)
 57 kB   (1.16%)

Form1::GetNext
BigObject ()
 1.9 MB   (39.91%)

BigObject::.ctor
void ()
 1.9 MB   (39.91%)

RuntimeConfigurationRecord::
static Internal.IInternalConfig
 56 kB   (1.15%)
```

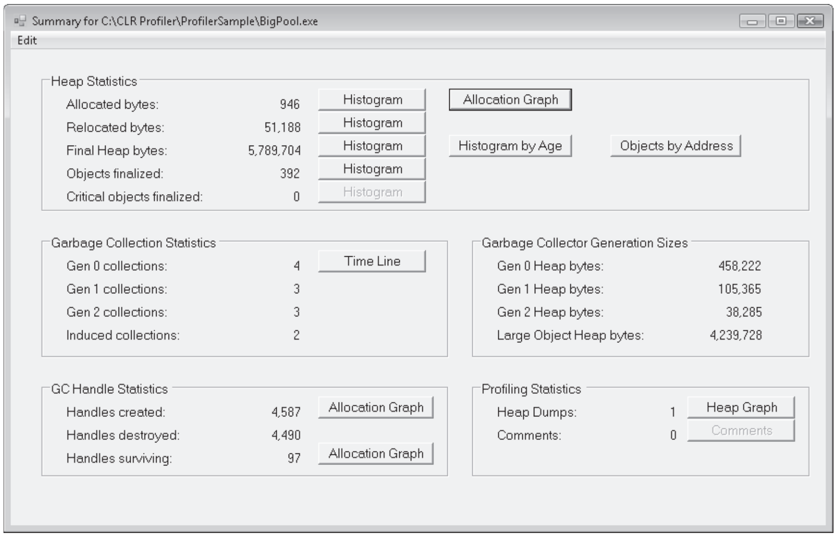
**FIGURE 7-7** The Allocation Graph for the CLR Profiler.

Scroll to the right of the Allocation Graph window to view the actual method, or nearest, of the allocation. The graph shows *Form1.GetNext* as the method where the large object byte array is being allocated. This pinpoints the location of the potential problem, which is helpful. You now know what source code to investigate first.

7. Let us create 20 objects using the BigPool application and then compare the results with the NoBigPool application. First, close the CLR Profiler and the NoBigPool application. Restart the CLR Profiler. Use the Start Application button to launch the BigPool application from within the CLR Profiler. In the BigPool user interface, press the Get Large button repeatedly to use the 10 objects in the pool. This exhausts the object pool. Clear 10 objects using the spin control and the Clear Object button. Finally, get another 10 objects using the Get Large button. You have now touched 20 big objects.

8. Let us view the impact of this activity on the Large Object Heap. Press the Show Heap Now button to collect current heap information about the application. Close the Heap Graph window when displayed. Choose View from the menu, and select Summary. The

size of the Large Object Heap is reported as 4.2 MB, which is about 8 percent less than the NoBigPool example. See Figure 7-8. This is a significant difference considering the minimum number of objects that were allocated. If that was hundreds of objects, the difference would be substantial.



Heap Statistics

| | |
|---|---|
| Allocated bytes: | 946 |
| Relocated bytes: | 51,188 |
| Final Heap bytes: | 5,789,704 |
| Objects finalized: | 392 |
| Critical objects finalized: | 0 |

Garbage Collection Statistics

| | |
|---|---|
| Gen 0 collections: | 4 |
| Gen 1 collections: | 3 |
| Gen 2 collections: | 3 |
| Induced collections: | 2 |

Garbage Collector Generation Sizes

| | |
|---|---|
| Gen 0 Heap bytes: | 458,222 |
| Gen 1 Heap bytes: | 105,365 |
| Gen 2 Heap bytes: | 38,285 |
| Large Object Heap bytes: | 4,239,728 |

GC Handle Statistics

| | |
|---|---|
| Handles created: | 4,587 |
| Handles destroyed: | 4,490 |
| Handles surviving: | 97 |

Profiling Statistics

| | |
|---|---|
| Heap Dumps: | 1 |
| Comments: | 0 |

**FIGURE 7-8** Summary of the managed heap for the BigPool application.

## Summary

The Common Language Runtime provides several services to managed applications, such as the Garbage Collector (GC). The developer is responsible for allocating memory for reference types on the managed heap using the new operator. However, the Garbage Collector is responsible for freeing managed objects on the managed heap.

The managed heap is organized in generations: Generation 0, 1, and 2. By partitioning the heap into generations, partial garbage collections can be performed to avoid the overhead of collecting the entire heap. There is also the Large Object Heap, which holds large objects. Because large objects typically live longer, this avoids the expense of promoting large objects between generations.

Garbage collection is initiated when a new allocation would cause the memory threshold for Generation 0 to be exceeded. A full garbage collection is a Generation 2 collection, which also collects Generations 0 and 1. Conversely, a garbage collection of Generation 1 also collects Generation 0. Finally, a collection of Generation 0, which is a minimum collection, only collects that generation. Garbage collection is performed on the Large Object Heap during a full garbage collection. Memory for objects on the Large Object Heap can be reclaimed. However, the Large Object Heap is not compacted.

There is sometimes a disparity between the size of a native resource and a managed wrapper class for that resource. Use the *GC.AddMemoryPressure* and *GC.RemoveMemoryPressure* methods to account for the differences.

Non-deterministic garbage collection occurs when additional memory is needed for Generation 0, which is somewhat unpredictable. This can delay the cleanup of resources associated with unreachable objects. For the deterministic cleanup of resources, implement the *IDisposable* interface. The *IDisposable* interface has a single method—the *Dispose* method. Call the *Dispose* method on a disposable object to immediately clean up related resources. If the base and derive classes are both disposable, implement the dispose pattern.

Use the CLR Profiler to diagnose memory issues in managed applications. The CLR Profiler offers a variety of graphs and reports that detail the current or historic state of the managed heap of a managed application. This information can be helpful in resolving difficult memory problems.

## Key Points

- The managed heap is segmented into Generations 0, 1, and 2 and the Large Object Heap.
- The assumption of the memory model for the managed environment is that small objects are short lived, while large objects live longer. In addition, objects of like size are likely to communicate with each other.
- Garbage collection in .NET is non-deterministic. You can force garbage collection with the *GC.Collect* method. However, this is not recommended.
- *GC.AddMemoryPressure* and *GC.RemoveMemoryPressure* apply artificial pressure to the managed heap. This is useful to account for the difference in size between a managed wrapper and the native resource.
- The Large Object Heap is collected, but not compacted, with a full garbage collection.
- Finalizable objects implement a *Finalize* method. Disposable objects implement the *IDisposable.Dispose* method. Implement the dispose pattern to properly define a base and derive a class as disposable.
- Weak references can be reclaimed at the discretion of the Garbage Collector.
- References abstract moveable pointers. Pin the reference to fix the pointer, which can then be safely passed to native code.

# Index

# About the Authors

## Donis Marshall

Donis Marshall is the chief executive officer at *Debuglive.com*. He manages a team of expert software developers creating the first entirely Web-based debugger for Windows applications. With 20 years of development experience and an in-depth background on Microsoft .NET technologies, he has authored several books, including *Programming Microsoft Visual C# 2008: The Language* and *.NET Security Programming*. As a trainer and consultant, Donis teaches classes and conducts seminars on .NET programming, debugging, security, and design and architecture.

## John Bruno

John Bruno is a senior program manager at Microsoft with over 10 years of application development experience. He specializes in designing and building scalable Web-based applications and services using Microsoft .NET technologies. Since joining Microsoft, John has played key roles in launching multiple versions of Windows Live and has been responsible for the service architecture and developer platform of Windows Live Spaces, which is delivered to over 100 million users worldwide. His current focus is on bringing the next generation of Web-based services for Windows Mobile to the world. You can contact John through his Web site at *http://johnbruno.net*.