



Design Patterns in C#

A Hands-on Guide with
Real-World Examples

Vaskaran Sarcar

Foreword by Priya Shimanthoor



Design Patterns in C#

**A Hands-on Guide with
Real-World Examples**

Vaskaran Sarcar

Foreword by Priya Shimanthoor

Apress®

Design Patterns in C#

Vaskaran Sarcar
Whitefield, Bangalore, Karnataka, India

ISBN-13 (pbk): 978-1-4842-3639-0
<https://doi.org/10.1007/978-1-4842-3640-6>

ISBN-13 (electronic): 978-1-4842-3640-6

Library of Congress Control Number: 2018946636

Copyright © 2018 by Vaskaran Sarcar

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the author nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Smriti Srivastava
Development Editor: Matthew Moodie
Coordinating Editor: Divya Modi

Cover designed by eStudioCalamar

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/978-1-4842-3639-0. For more detailed information, please visit www.apress.com/source-code.

Printed on acid-free paper

This book is dedicated to our second child who came into our lives for only two short months. He did not see the light of this world because he stopped growing with no significant heartbeat. We pray to Almighty God: "Wherever you are now, let His blessings be with you and make you happy. Sweetheart, though we could not give you life and reveal the beauty of this wonderful world to you, please remember that your parents always love you."

Table of Contents

About the Author	xix
About the Technical Reviewers	xxi
Acknowledgments	xxiii
Foreword	xxv
Preface	xxvii
Guidelines for Using This Book.....	xxix
Part I: Gang of Four Design Patterns	1
I.A: Creational Patterns.....	3
Chapter 1: Singleton Pattern	5
GoF Definition.....	5
Concept.....	5
Real-Life Example	5
Computer World Example.....	5
Illustration	6
Class Diagram	6
Solution Explorer View.....	7
Discussion	7
Implementation	8
Output.....	9
Challenges	10
Q&A Session.....	11

TABLE OF CONTENTS

Chapter 2: Prototype Pattern	17
GoF Definition.....	17
Concept.....	17
Real-Life Example.....	17
Computer World Example.....	17
Illustration.....	18
Class Diagram	18
Directed Graph Document	20
Solution Explorer View.....	20
Implementation	22
Output.....	24
Q&A Session.....	25
Demonstration.....	27
Output.....	29
Chapter 3: Builder Pattern	31
GoF Definition.....	31
Concept.....	31
Real-Life Example.....	32
Computer World Example.....	32
Illustration.....	32
Class Diagram	33
Solution Explorer View.....	33
Implementation	34
Output.....	39
Q&A Session.....	39
Chapter 4: Factory Method Pattern	43
GoF Definition.....	43
Concept.....	43
Real-Life Example.....	43

TABLE OF CONTENTS

Computer World Example.....	44
Illustration	44
Class Diagram	44
Directed Graph Document	45
Solution Explorer View.....	45
Implementation	46
Output.....	49
Modified Implementation.....	49
Modified Output.....	51
Analysis	51
Q&A Session.....	51
Chapter 5: Abstract Factory Pattern.....	55
GoF Definition.....	55
Concept.....	55
Real-Life Example	56
Computer World Example.....	56
Illustration	56
Class Diagram	58
Solution Explorer View.....	58
Implementation	60
Output.....	63
Q&A Session.....	64
Conclusion	67
I.B: Structural Patterns.....	69
Chapter 6: Proxy Pattern	71
GoF Definition.....	71
Concept.....	71
Real-Life Example	71
Computer World Example.....	71

TABLE OF CONTENTS

Illustration	72
Class Diagram	72
Directed Graph Document	73
Solution Explorer View.....	73
Implementation	74
Output.....	76
Q&A Session.....	76
Modified Implementation.....	79
Modified Output.....	82
Chapter 7: Decorator Pattern.....	83
GoF Definition.....	83
Concept.....	83
Real-Life Example.....	83
Computer World Example.....	85
Illustration	85
Class Diagram	86
Solution Explorer View.....	87
Implementation	88
Output.....	90
Q&A Session.....	91
Chapter 8: Adapter Pattern.....	97
GoF Definition.....	97
Concept.....	97
Real-Life Example.....	97
Computer World Example.....	98
Illustration	99
Class Diagram	99
Directed Graph Document	100
Solution Explorer View.....	100
Implementation	102

TABLE OF CONTENTS

Output.....	103
Modified Illustration.....	103
Key Characteristics of the Modified Implementation.....	104
Modified Solution Explorer View	105
Modified Implementation.....	106
Modified Output.....	109
Types of Adapters	109
Q&A Session.....	110
Chapter 9: Facade Pattern	113
GoF Definition.....	113
Concept.....	113
Real-Life Example	113
Computer World Example.....	113
Illustration	114
Class Diagram	114
Directed Graph Document	115
Solution Explorer View	115
Implementation	116
Output.....	120
Q&A Session.....	121
Chapter 10: Flyweight Pattern.....	123
GoF Definition.....	123
Concept.....	123
Real-Life Example	124
Computer World Example.....	124
Illustration	124
Class Diagram.....	125
Directed Graph Document.....	126
Solution Explorer View	127

TABLE OF CONTENTS

Implementation.....	127
Output	130
Improvements to the Program	131
Modified Class Diagram	132
Modified Solution Explorer View	133
Modified Implementation	133
Modified Output	137
Q&A Session.....	138
Chapter 11: Composite Pattern.....	143
GoF Definition.....	143
Concept.....	143
Real-Life Example.....	144
Computer World Example.....	144
Illustration.....	144
Class Diagram	146
Solution Explorer View.....	147
Implementation	148
Output.....	151
Q&A Session.....	152
Chapter 12: Bridge Pattern.....	155
GoF Definition.....	155
Concept.....	155
Real-Life Example.....	155
Computer World Example.....	156
Illustration.....	156
Class Diagram	158
Solution Explorer View.....	159
Implementation	159
Output.....	163
Q&A Session.....	163

TABLE OF CONTENTS

I.C: Behavioral Patterns	165
Chapter 13: Visitor Pattern	167
GoF Definition.....	167
Concept.....	167
Real-Life Example	167
Computer World Example.....	168
Illustration	168
Class Diagram	169
Solution Explorer View.....	170
Implementation	170
Output.....	172
Modified Illustration.....	172
Modified Solution Explorer View.....	176
Modified Implementation.....	177
Modified Output.....	183
Q&A Session.....	184
Chapter 14: Observer Pattern	185
GoF Definition.....	185
Concept.....	185
Real-Life Example	188
Computer World Example.....	188
Illustration	189
Class Diagram	189
Solution Explorer View.....	190
Implementation	191
Output.....	194
Analysis of the Output	194
Q&A Session.....	195

TABLE OF CONTENTS

Chapter 15: Strategy (Policy) Pattern	199
GoF Definition.....	199
Concept.....	199
Real-Life Example.....	199
Computer World Example.....	199
Illustration.....	200
Class Diagram	200
Directed Graph Document.....	201
Solution Explorer View	202
Implementation.....	203
Output	205
Q&A Session.....	206
Chapter 16: Template Method Pattern	211
GoF Definition.....	211
Concept.....	211
Real-Life Example.....	211
Computer World Example.....	212
Illustration.....	212
Class Diagram	213
Solution Explorer View.....	214
Implementation	214
Output.....	217
Q&A Session.....	217
Modified Implementation.....	218
Modified Output.....	221
Chapter 17: Command Pattern	223
GoF Definition.....	223
Concept.....	223
Real-Life Example.....	223
Computer World Example.....	224

TABLE OF CONTENTS

Illustration	224
Class Diagram	225
Directed Graph Document	226
Solution Explorer View.....	227
Implementation	228
Output.....	231
Q&A Session.....	231
Modified Class Diagram	232
Modified Solution Explorer View.....	233
Modified Implementation.....	234
Modified Output.....	240
Chapter 18: Iterator Pattern	243
GoF Definition.....	243
Concept.....	243
Real-Life Example	244
Computer World Example.....	244
Illustration	244
Class Diagram.....	245
Directed Graph Document.....	246
Solution Explorer View	247
Implementation	248
Output	253
Q&A Session.....	253
Chapter 19: Memento Pattern.....	257
GoF Definition.....	257
Concept.....	257
Real-Life Example	257
Computer World Example.....	257

TABLE OF CONTENTS

Illustration	258
Class Diagram	258
Directed Graph Document	259
Solution Explorer View.....	260
Implementation	260
Output.....	264
Q&A Session.....	264
Modified Implementation.....	264
Modified Output.....	266
Chapter 20: State Pattern	269
GoF Definition.....	269
Concept.....	269
Real-Life Example.....	269
Computer World Example.....	270
Illustration	270
Class Diagram	271
Directed Graph Document	272
Solution Explorer View.....	272
Implementation	274
Output.....	279
Q&A Session.....	279
Chapter 21: Mediator Pattern	283
GoF Definition.....	283
Concept.....	283
Real-Life Example.....	283
Computer World Example.....	284
Illustration	284
Class Diagram	286
Solution Explorer View.....	286
Implementation	288

TABLE OF CONTENTS

Output.....	292
Analysis	292
Q&A Session.....	293
Modified Illustration.....	294
Modified Implementation.....	294
Modified Output.....	299
Chapter 22: Chain of Responsibility Pattern.....	303
GoF Definition.....	303
Concept.....	303
Real-Life Example	304
Computer World Example.....	304
Illustration	305
Class Diagram	306
Directed Graph Document	307
Solution Explorer View.....	307
Implementation	308
Output.....	312
Q&A Session.....	312
Chapter 23: Interpreter Pattern	315
GoF Definition.....	315
Concept.....	315
Real-Life Example	315
Computer World Example.....	315
Illustration	316
Class Diagram	316
Solution Explorer View.....	318
Implementation	318
Output.....	325
Q&A Session.....	326

TABLE OF CONTENTS

Part II: Additional Design Patterns	327
Chapter 24: Simple Factory Pattern	329
Definition.....	329
Concept.....	329
Real-Life Example.....	329
Computer World Example.....	330
Illustration.....	330
Class Diagram	331
Directed Graph Document	332
Solution Explorer View.....	332
Implementation	333
Output.....	336
Q&A Session.....	337
Chapter 25: Null Object Pattern	341
Definition.....	341
Concept.....	341
A Faulty Program.....	342
Output with Valid Inputs	344
Analysis with Unwanted Input.....	344
Encountered Exception.....	345
Immediate Remedy.....	345
Analysis	345
Real-Life Example.....	346
Computer World Example.....	346
Illustration.....	346
Class Diagram	347
Solution Explorer View.....	348
Implementation	349
Output.....	352
Q&A Session.....	353

TABLE OF CONTENTS

Chapter 26: MVC Pattern	355
Definition.....	355
Concept.....	355
Key Points to Remember	356
Variation 1	357
Variation 2	358
Variation 3	358
Real-Life Example	359
Computer World Example.....	359
Illustration	360
Class Diagram	362
Solution Explorer View.....	362
Implementation	363
Output.....	373
Q&A Session.....	377
Part III: Final Thoughts on Design Patterns	383
Chapter 27: Criticisms of Design Patterns.....	385
Q&A Session.....	387
Chapter 28: Anti-patterns.....	391
What Is an Anti-pattern?.....	391
Q&A Session.....	392
Chapter 29: Sealing the Leaks in Your Applications.....	397
How Garbage Collection Works	398
Demonstration 1.....	400
Output.....	403
Analysis	404
Q&A Session 1.....	405
Quiz.....	413
Quiz.....	415

TABLE OF CONTENTS

Quiz.....	417
Quiz.....	418
Understanding Memory Leaks	419
Demonstration 2.....	420
Snapshots.....	422
Analysis	423
Modified Code.....	424
Solutions.....	427
Analysis	428
Q&A Session 2.....	430
Chapter 30: FAQ	433
Appendix A: Brief Overview of GoF Design Patterns.....	439
Key Points	440
Q&A Session.....	443
Appendix B: Some Useful Resources	445
Appendix C: The Road Ahead.....	447
Index.....	449

About the Author



Vaskaran Sarcar obtained his Master of Engineering degree from Jadavpur University, Kolkata in Software Engineering. Currently he is a Senior Software Engineer and Team Lead in the R&D Hub at HP Inc. India. He was a national Gate Scholar and has more than 12 years of experience in education and the IT industry. He is an alumnus of prestigious institutions in India such as Jadavpur University, Vidyasagar University, and Presidency University (formerly Presidency College). Reading and learning new things are his passions. You can connect with him at vaskaran@rediffmail.com or find him on LinkedIn at

<https://www.linkedin.com/in/vaskaransarcar>.

Other books by Vaskaran include the following:

Interactive C# (Apress, 2017)

Interactive Object-Oriented Programming in Java (Apress, 2016)

Java Design Patterns (Apress, 2016)

C# Basics: Test Your Skill (CreateSpace, 2015)

Operating System: Computer Science Interview Series (CreateSpace, 2014)

About the Technical Reviewers



Shekhar Kumar Maravi is a system software engineer whose main interests are programming languages, algorithms, and data structures. He obtained his master's degree in Computer Science and Engineering from the Indian Institute of Technology Bombay. After graduation, he joined HP's R&D Hub in India to work on printer firmware. Currently he is a technical lead for automated lab diagnostic device firmware and software at Siemens Healthcare India. He can be reached by e-mail at shekhar.maravi@gmail.com or via LinkedIn at <https://www.linkedin.com/in/shekharmaravi/>.



Ankit Khare is a senior software engineer with expertise in software architecture and designing, programming languages, algorithms, and data structure. After obtaining a bachelor's degree in Computer Science, he joined HP's R&D Hub in India, where he worked with various laser-jet firmware teams. He is currently involved in future machine vision development for print image diagnostic tools involving ink-jet, large-format, and laser-jet printers. He can be reached by e-mail at akikhare@gmail.com or at <https://www.linkedin.com/in/khareankit/>.

Acknowledgments

First, I thank the Almighty. I sincerely believe that only with His blessings I was able to complete this book.

I extend my deepest gratitude and thanks to the following people:

Ratanlal Sarkar and Manikuntala Sarkar, my parents. Without your blessings, I could not have completed this work.

Indrani, my wife, and Ambika, my daughter. Sweethearts, once again, without your love, I could not proceed at all. I know that we needed to limit many social gatherings and invitations to complete this work on time.

Sambaran, my brother. Thank you for your constant encouragement toward me.

Shekhar and Ankit, my friends and technical advisers. Whenever I was in need, your support was there. Thank you one more time.

Priya Shimanthoor, my colleague and mentor. A special thanks to you for investing your time to write a foreword for my book. From the moment that an expert like you agreed to write about me, I was motivated to enhance the quality of my work.

Lastly, I extend my deepest gratitude to my publisher, the editorial board members, and everyone who directly or indirectly supported this book.

Foreword



Written programs need to be flexible, easily maintainable, and reusable, which means they must be elegant. How do we know that a program is as elegant as it can be? The answer is that a successful programmer should use two primary tools: a good programming language (here C#) and design patterns.

When working on a particular problem, it is unusual to tackle it by inventing a new solution that is completely dissimilar from existing ones. Instead, one often recalls a similar problem and reuses the essence of its solution to solve the new problem. This kind of problem-solving is common to many different domains but especially software engineering.

Design patterns are important building blocks for designing and modeling applications on all platforms. Design patterns help us understand, discuss, and reuse applications on a specific platform. The most commonly stated reasons for studying patterns are to reuse solutions and to establish common terminology. By reusing already established designs, the developer gets a head start on the problem and avoids common mistakes. The benefit of learning from the experience of others results in not having to reinvent solutions for commonly recurring problems. The other reason for using patterns is that common terminology brings a shared base of vocabulary and viewpoint of the problem to the developers. It provides a point of reference during the analysis and design phases of a project.

Vaskaran Sarcar, who has worked with me for several years now, has been our C# team's most valuable professional over the years. Vaskaran is that kind of software developer—enthusiastic, knowledgeable, talented, curious, analytical, and a teacher of others. He gets into the root of any particular problem he is trying to solve in a well-defined and organized way. He is committed and works hard until he gets to the solution.

FOREWORD

That is why I am excited about this book. *Design Patterns in C#* brings the frequently abstruse world of design patterns into sharp focus with the approach used: a definition, a core concept, a real-life example, a computer world example, and a sample program with output. I look forward to see where developers can go with this easy approach and language and what useful patterns they can build into the infrastructure of other languages.

Priya Shimanthoor

Test Architect, Firmware Team

HP India PPS R&D Hub

Bangalore, India

April 8, 2018

About Priya Shimanthoor

Priya Shimanthoor completed her graduation with Computer Science and Engineering in 2000. Currently she is a Test Architect. In last 17 years, she tested different Desktop/web applications, software solutions, firmware and mobile applications using various tools like WinRunner, Quick Test Professional, Rational Robot, Rational Test Manager, LoadRunner, Test Director, Quality Center, Application lifecycle management, Python Everest Framework and JUnit. She also worked on various virtualization solutions and she is well-versed with different process models like CMMi, Agile, Lean Sigma etc.

Preface

Welcome to your journey of *Design Patterns in C#*.

You probably know that the concept of design patterns became extremely popular with the Gang of Four's famous book *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley, 1994). Most important, these concepts still apply in today's programming world. The book came out at the end of 1994, and it primarily focused on C++. In 2005, C# had its first major release (C# 2.0). Since then, C# has become rich with features and is now a popular programming language.

In 2015, I wrote the book *Design Patterns in C#: Computer Science Interview Series*, which was basically a companion to this book.

In that tiny book, my core intention was to implement each of the 23 Gang of Four (GoF) design patterns with C# implementations. I wanted to present each pattern with simple examples. One thing was always in my mind when writing: I wanted to use the most basic constructs of C# so that the code would be compatible with both the upcoming version and the legacy version of C#. I have found this method helpful in the world of programming.

In the last two years, I have added enhancements to the book based on readers feedback. In 2017, Visual Studio launched with the upgraded version of C#, and it gave me pleasure when I saw the code being executed as expected with the updated version of C#.

This time, I wanted to focus on another important area; I call it the “doubt-clearing sessions.” I knew that if I could add some more information such as alternative ways to write these implementations, the pros and cons of these patterns, when to choose one approach over another, and so on, readers would find this book even more helpful. So, in this enhanced version of the original, I have added a “Q&A Session” section to each chapter that can help you to learn about each pattern in more depth.

In the world of programming, there is no shortage of patterns, and each has its own significance. So, in addition to the 23 GoF design patterns covered in Part I, I discuss three design patterns in Part II of the book that are equally important in today's world of programming. Finally, in Part III of the book, I give you an overview of anti-patterns and discuss the criticism of design patterns with regard to one important concept called *memory leak*. I have included this topic because in software development we cannot stamp the word *good* on the implementation of a design if it suffers from memory leaks.

Before jumping into the design patterns, I want to highlight a few more points.

- You are an intelligent person. You have chosen a subject that can assist you throughout your career. If you are a developer/programmer, you need these concepts. If you are an architect of a software organization, you need these concepts. If you are a college student, you need these concepts, not only to score high on exams but to enter the corporate world. Even if you are a tester who needs to take care of white-box testing or simply needs to know about the code paths of a product, these concepts will help you a lot.
- I already mentioned that this book was written using the most basic features of C# so that you do not need to be familiar with advanced topics in C#. These examples are simple and straightforward. I believe that these examples are written in such a way that even if you are familiar with another popular language such as Java, C++, and so on, you can still easily grasp the concepts in this book.
- There are many books about design patterns and related topics. You may be wondering why I would want to write a new one about the same topics. The simple answer to this question is that I have found other reference material to be scattered. Second, in most cases, many of those examples are unnecessarily large and complex. I like simple examples. I believe that anyone can grasp a new idea with simple examples, and if the core concept is clear, you can easily move into more advanced areas. I believe that this book scores high in this context. The illustrated examples are simple. I wanted to keep this book concise so that it motivates you to continue the journey of learning in your field.
- Each chapter is divided into six parts: a definition (which is basically termed as intent in GoF book), a core concept, a real-life example, a computer/coding world example, a sample program with various outputs, and the “Q&A Session” section. These “Q&A Session” sections can help you to learn about each pattern in more depth.
- Please remember that you have just started on this journey. As you learn about these concepts, try to write your own code; only then will you master this area.

Guidelines for Using This Book

Here are some suggestions so you can use the book more effectively:

- I assume that you have some idea about the GoF design patterns. If you are absolutely new to design patterns, I suggest you quickly go through Appendix A. This appendix will help you to become familiar with the basic concepts of design patterns.
- If you are confident with the coverage of Appendix A, you can start with any part of the book. But I suggest you go through the chapters sequentially. The reason is that some fundamental design techniques may be discussed in the “Q&A Session” section of a previous chapter, and I have not repeated those techniques in the later chapters.
- There is only one exception to the previous suggestion. There are three factory patterns: Simple Factory, Factory Method, and Abstract Factory. These three patterns are related, but the Simple Factory pattern does not directly fall into the GoF design catalog, so it appears in Part II of the book. So, it is suggested that when you start learning about these 3 factory patterns, you start with the Simple Factory Pattern.
- These programs are tested with Visual Studio Community edition (2017) in a windows 10 environment. This Community edition is free of cost. If you do not use windows operating system, you can use Visual Studio Code Development environment IDE that can support Linux or Mac operating system. This multi-platform IDE is also free. Though I believe that the results should not vary in other environments, but you know the nature of softwares-they are naughty. So, I recommend that if you want to see the exact same outputs, it will be better if you can mimic the same environment.
- You can download and install the Visual Studio IDE from <https://www.visualstudio.com/downloads/> (Figure 1). You can use the free Visual Studio Community edition.

GUIDELINES FOR USING THIS BOOK

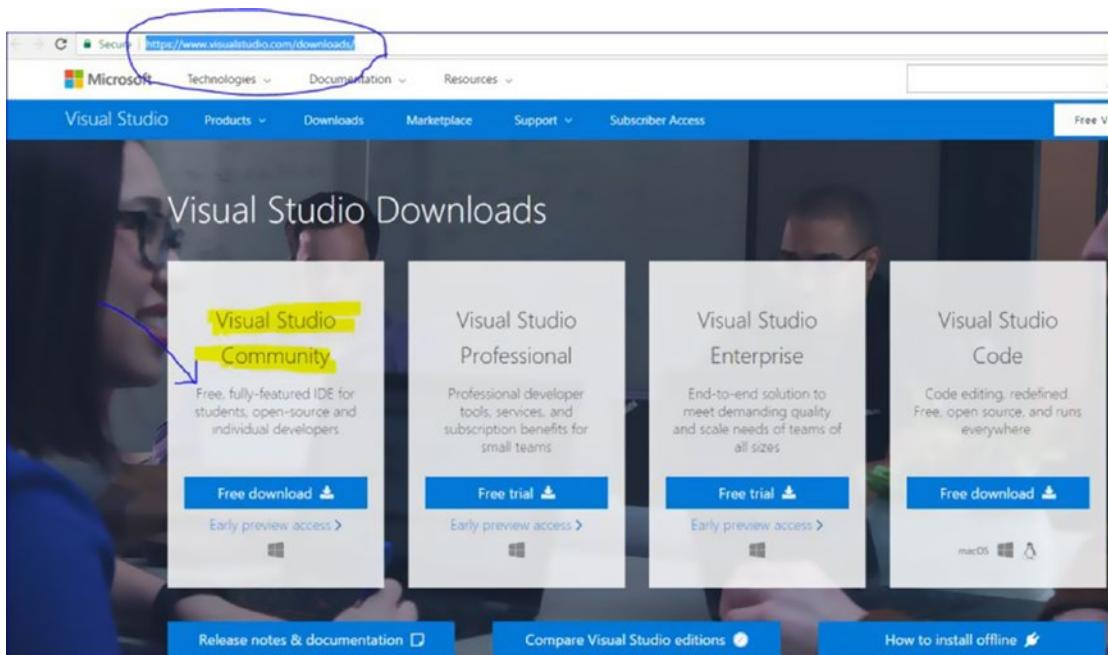


Figure 1. Download link for Visual Studio

Note At the time of this writing, this link works fine, and the information is correct. But the link and policies may change in the future.

- UML designers are removed in Visual Studio 2017. To know the reason, you can refer the following link:

<https://blogs.msdn.microsoft.com/devops/2016/10/14/uml-designers-have-been-removed-layer-designer-now-supports-live-architectural-analysis/>.

In this book, I have drawn few Directed Graph Document (DAG) to explain the things better. To serve those needs, as per the recommendation, I have used an older version of Visual Studio (Ultimate 2013) to draw those. But to understand the concepts, these are not mandatory for you. So, you can move smoothly with Visual Studio Community 2017.

PART I

Gang of Four Design Patterns

I.A: Creational Patterns

CHAPTER 1

Singleton Pattern

This chapter covers the Singleton pattern.

GoF Definition

Ensure a class has only one instance, and provide a global point of access to it.

Concept

A particular class should have only one instance. You can use this instance whenever you need it and therefore avoid creating unnecessary objects.

Real-Life Example

Suppose you are a member of a sports team and your team is participating in a tournament. When your team plays against another team, as per the rules of the game, the captains of the two sides must have a coin toss. If your team does not have a captain, you need to elect someone to be the captain first. Your team must have one and only one captain.

Computer World Example

In some software systems, you may decide to maintain only one file system so that you can use it for the centralized management of resources.

Illustration

These are the key characteristics in the following implementation:

- The constructor is private in this example. So, you cannot instantiate in a normal fashion (using new).
- Before you attempt to create an instance of a class, you check whether you already have an available copy. If you do not have any such copy, you create it; otherwise, you simply reuse the existing copy.

Class Diagram

Figure 1-1 shows the class diagram for the illustration of the Singleton pattern.

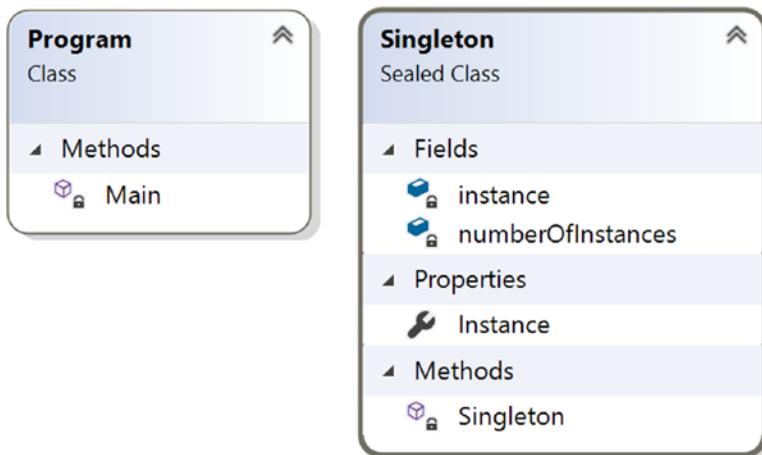


Figure 1-1. Class diagram

Solution Explorer View

Figure 1-2 shows the high-level structure of the parts of the program.

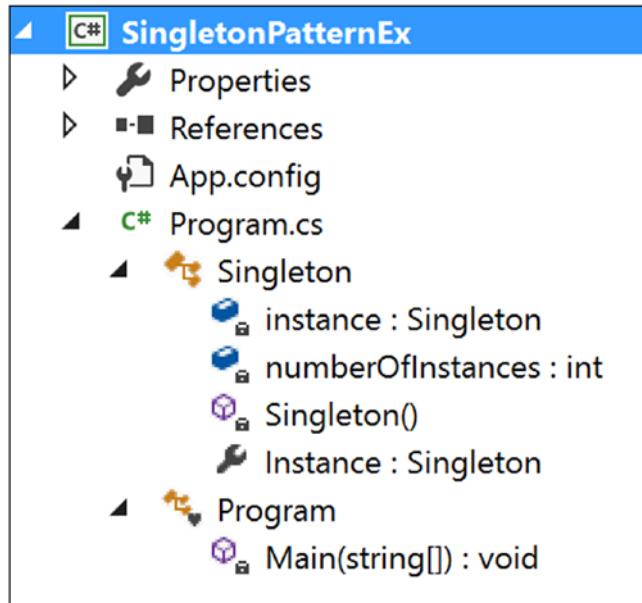


Figure 1-2. Solution Explorer View

Discussion

This simple example illustrates the concept of the Singleton pattern. This approach is known as *static initialization*.

Initially the C++ specification had some ambiguity about the initialization order of static variables (remember that the origin of C# is closely tied with C and C++), but the .NET Framework resolved these issues.

The following are the notable characteristics of this approach:

- The Common Language Runtime (CLR) takes care of the variable initialization process.
- You create an instance when any member of the class is referenced.

- The `public static` member ensures a global point of access. It confirms that the instantiation process will not start until you invoke the `Instance` property of the class (in other words, it supports lazy instantiation). The `sealed` keyword prevents the further derivation of the class (so that its subclass cannot misuse it), and `readonly` ensures that the assignment process takes place during the static initialization.
- The constructor is private. So, you cannot instantiate the `Singleton` class inside `Main()`. This will help you refer to the one instance that can exist in the system.

Implementation

Here is the implementation of the example:

```
using System;

namespace SingletonPatternEx
{
    public sealed class Singleton
    {
        private static readonly Singleton instance=new Singleton();
        private int numberofInstances = 0;
        //Private constructor is used to prevent
        //creation of instances with 'new' keyword outside this class
        private Singleton()
        {
            Console.WriteLine("Instantiating inside the private constructor.");
            numberofInstances++;
            Console.WriteLine("Number of instances ={0}", numberofInstances);
        }
        public static Singleton Instance
        {
            get
            {
                Console.WriteLine("We already have an instance now.Use it.");
            }
        }
    }
}
```

```

        return instance;
    }
}
}

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***Singleton Pattern Demo***\n");
        //Console.WriteLine(Singleton.MyInt);
        // Private Constructor.So,we cannot use 'new' keyword.
        Console.WriteLine("Trying to create instance s1.");
        Singleton s1 = Singleton.Instance;
        Console.WriteLine("Trying to create instance s2.");
        Singleton s2 = Singleton.Instance;
        if (s1 == s2)
        {
            Console.WriteLine("Only one instance exists.");
        }
        else
        {
            Console.WriteLine("Different instances exist.");
        }
        Console.Read();
    }
}
}

```

Output

Here is the output of the example:

```
***Singleton Pattern Demo***
```

```
Trying to create instance s1.
```

```
Instantiating inside the private constructor.
```

Number of instances =1
We already have an instance now.Use it.
Trying to create instance s2.
We already have an instance now.Use it.
Only one instance exists.

Challenges

Consider the following code. Suppose you have added one more line of code (shown in bold) in the Singleton class.

```
public sealed class Singleton
{
    private static readonly Singleton instance = new Singleton();
    private int numberofInstances = 0;
    //Private constructor is used to prevent
    //creation of instances with 'new' keyword outside this class
    private Singleton()
    {
        Console.WriteLine("Instantiating inside the private constructor.");
        numberofInstances++;
        Console.WriteLine("Number of instances ={0}", numberofInstances);
    }
    public static Singleton Instance
    {
        get
        {
            Console.WriteLine("We already have an instance now.Use it.");
            return instance;
        }
    }
    public static int MyInt = 25;
}
```

And suppose, your `Main()` method looks like this:

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***Singleton Pattern Demo***\n");
        Console.WriteLine(Singleton.MyInt);
        Console.Read();
    }
}
```

Now, if you execute the program, you will see following output:

Number of instances =1

Singleton Pattern Demo

25

This illustrates the downside of this approach. Specifically, inside the `Main()` method, you tried to use the static variable `MyInt`, but your application still created an instance of the `Singleton` class. In other words, with this approach, you have less control over the instantiation process, which starts whenever you refer to any static member of the class.

However, in most cases, you do not care about this drawback. You can tolerate it because you know that it is a one-time activity and the process will not be repeated, so this approach is widely used in .NET applications.

Q&A Session

1. Why are you complicating stuff? You can simply write your Singleton class as follows:

```
public class Singleton
{
    private static Singleton instance;

    private Singleton() { }
```

```

public static Singleton Instance
{
    get
    {
        if (instance == null)
        {
            instance = new Singleton();
        }
        return instance;
    }
}

```

Answer:

This approach can work in a single-threaded environment. But consider a multithreaded environment. In a multithreaded environment, suppose two (or more) threads try to evaluate this:

```
if (instance == null)
```

If they see that the instance has not been created yet, each of them will try to create a new instance. As a result, you may end up with multiple instances of the class.

2. Are there any alternative approaches for modeling Singleton design patterns?

Answer:

There are many approaches. Each of them has pros and cons. I'll discuss one approach called *double checked locking*. MSDN outlines the approach as shown here:

```

//Double checked locking
using System;

public sealed class Singleton

```

```

{
    //We are using volatile to ensure that
    //assignment to the instance variable finishes before it's
    //access.
    private static volatile Singleton instance;
    private static object lockObject = new Object();

    private Singleton() { }

    public static Singleton Instance
    {
        get
        {
            if (instance == null)
            {
                lock (lockObject)
                {
                    if (instance == null)
                        instance = new Singleton();
                }
            }
            return instance;
        }
    }
}

```

This approach can help you create the instances when they are really needed. But you must remember that, in general, the locking mechanism is expensive.

If you are further interested in Singleton patterns, you can refer to <http://csharpindepth.com/Articles/General/Singleton.aspx>, which discusses various alternatives (with their pros and cons) to model a Singleton pattern.

3. Why are you marking the instance as volatile in double checked locking example?

Answer:

Let's see what C# specification tells you:

The volatile keyword indicates that a field might be modified by multiple threads that are executing at the same time. Fields that are declared volatile are not subject to compiler optimizations that assume access by a single thread. This ensures that the most up-to-date value is present in the field at all times.

In simple terms, the volatile keyword can help you to provide a serialize access mechanism. In other words, all threads will observe the changes by any other thread as per their execution order. You will also remember that the volatile keyword is applicable for class (or struct) fields; you cannot apply it to local variables.

4. Why are multiple object creations a big concern?

Answer:

- Object creations in the real world are treated as costly operations.
- Sometimes you may need to implement a centralized system for easy maintenance. This also helps you to provide a global access mechanism.

5. Why are you using the keyword “sealed”? The singleton class has a private constructor that can stop the derivation process. Is the understanding correct?

Answer:

Good catch. It was not mandatory but it is always better to show your intention clearly. I have used it to guard one special case-if you are tempted to use a derived nested class as below:

```
//public sealed class Singleton  
//Not using "sealed" keyword now
```

```

public class Singleton
{
    private static readonly Singleton instance = new Singleton();
    private static int numberofInstances = 0;
    //Private constructor is used to prevent
    //creation of instances with 'new' keyword outside this class
    //protected Singleton()
    private Singleton()
    {
        Console.WriteLine("Instantiating inside the private
constructor.");
        numberofInstances++;
        Console.WriteLine("Number of instances ={0}",
        numberofInstances);
    }
    public static Singleton Instance
    {
        get
        {
            Console.WriteLine("We already have an instance
now.Use it.");
            return instance;
        }
    }
    //The keyword "sealed" can guard this scenario.
    public class NestedDerived : Singleton { }
}

```

Now inside Main() method, you can create multiple objects with statements like these:

```

Singleton.NestedDerived nestedClassObject1 =
new Singleton.NestedDerived(); //1
Singleton.NestedDerived nestedClassObject2 =
new Singleton.NestedDerived(); //2

```

So, I always prefer to use “sealed” keyword in a similar context.

CHAPTER 2

Prototype Pattern

This chapter covers the Prototype pattern.

GoF Definition

Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

Concept

This pattern provides an alternative method for instantiating new objects by copying or cloning an instance of an existing object. You can avoid the expense of creating a new instance using this concept.

Real-Life Example

Suppose you have a master copy of a valuable document. You need to incorporate some change into it to analyze the effect of the change. In this case, you can make a photocopy of the original document and edit the changes in the photocopied document.

Computer World Example

Let's assume that you already have an application that is stable. In the future, you may want to modify the application with some small changes. You must start with a copy of your original application, make the changes, and then analyze further. Surely you do not want to start from scratch to merely make a change; this would cost you time and money.

Illustration

In this example, I will follow the structure shown in Figure 2-1.

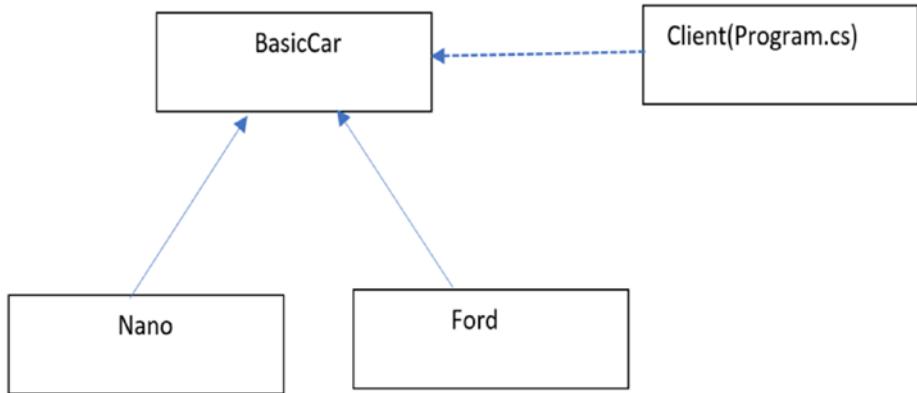


Figure 2-1. Prototype example

Here `BasicCar` is the prototype. `Nano` and `Ford` are the concrete prototypes, and they have implemented the `Clone()` method defined in `BasicCar`. Notice that in this example I have created a `BasicCar` object with some default price. Later I have modified that price as per the model. `Program.cs` is the client in the implementation.

Class Diagram

Figure 2-2 shows the class diagram.

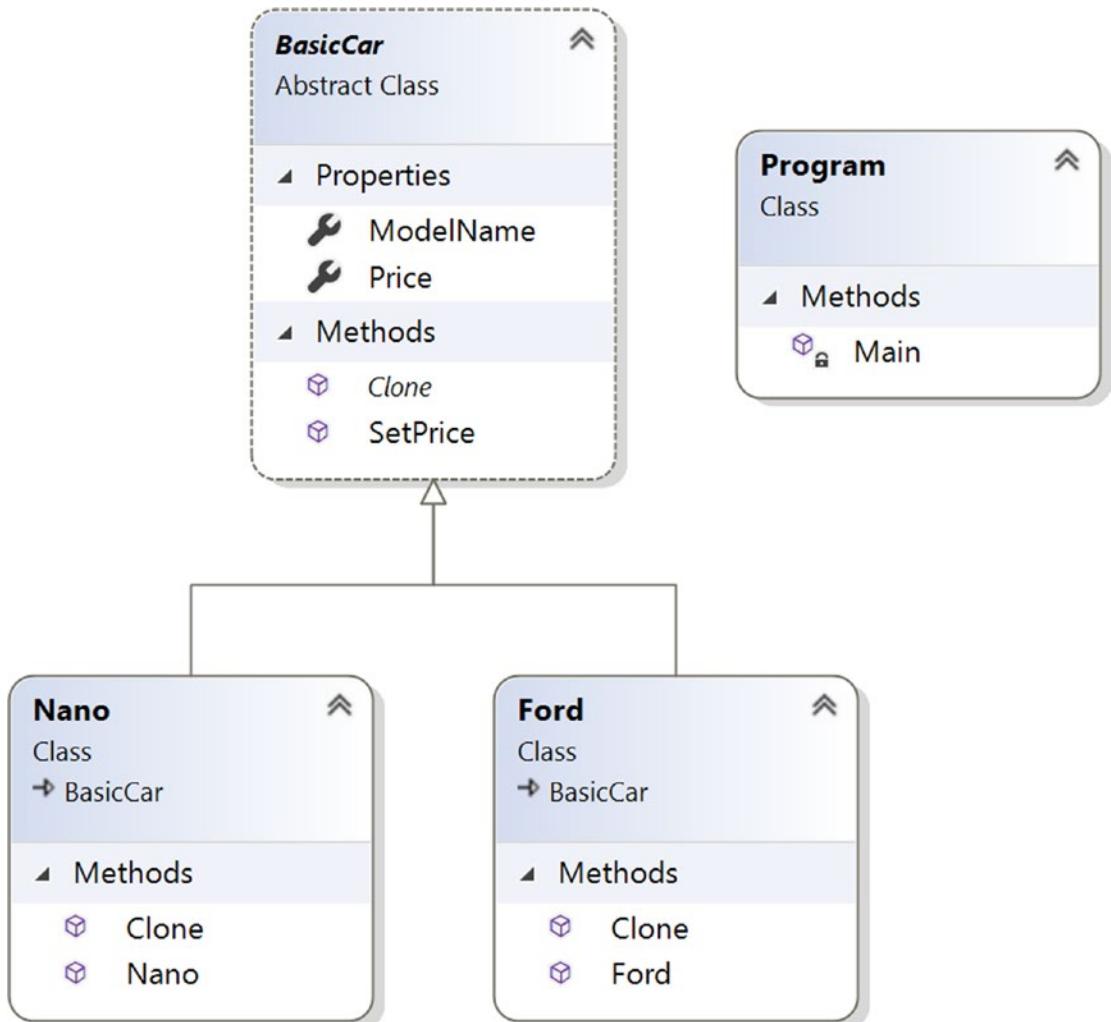


Figure 2-2. Class diagram

Directed Graph Document

Figure 2-3 shows the directed graph document.

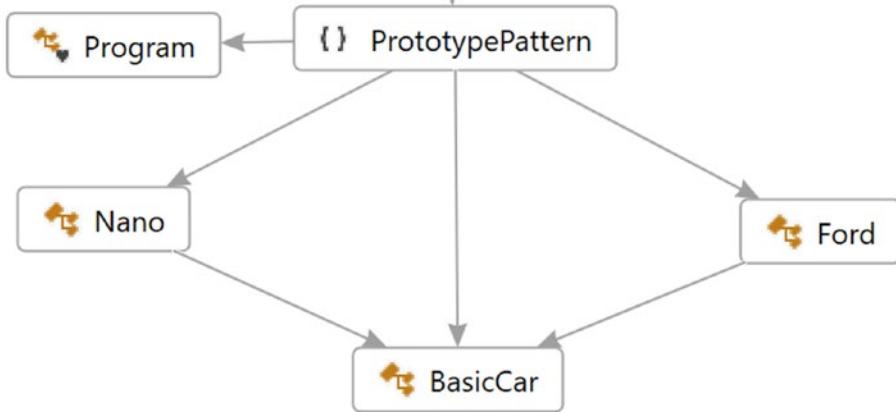


Figure 2-3. Directed Graph Document

Solution Explorer View

Figure 2-4 shows the high-level structure of the parts of the program.

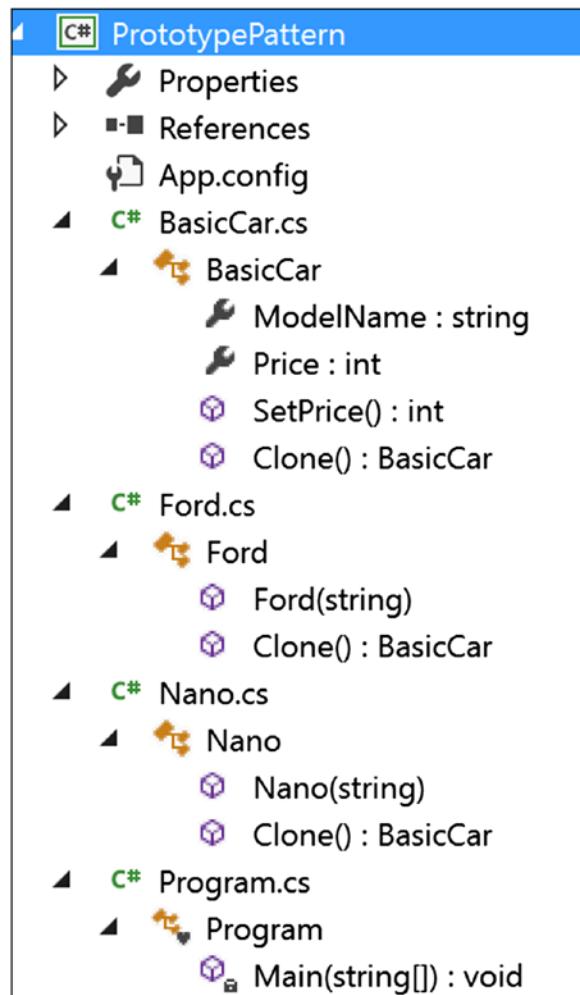


Figure 2-4. Solution Explorer View

Implementation

Here's the implementation:

```
//BasicCar.cs
using System;

namespace PrototypePattern
{
    public abstract class BasicCar
    {
        public string ModelName{get;set;}
        public int Price {get; set;}
        public static int SetPrice()
        {
            int price = 0;
            Random r = new Random();
            int p = r.Next(200000, 500000);
            price = p;
            return price;
        }
        public abstract BasicCar Clone();
    }
}

//Nano.cs
using System;

namespace PrototypePattern
{
    public class Nano:BasicCar
    {
        public Nano(string m)
        {
            ModelName = m;
        }

        public override BasicCar Clone()
        {

```

```
        return (Nano) this.MemberwiseClone(); //shallow Clone
    }
}
}
//Ford.cs

using System;

namespace PrototypePattern
{
    public class Ford:BasicCar
    {
        public Ford(string m)
        {
            ModelName = m;
        }

        public override BasicCar Clone()
        {
            return (Ford)this.MemberwiseClone();
        }
    }
}

//Client

using System;

namespace PrototypePattern
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("***Prototype Pattern Demo***\n");
            //Base or Original Copy
            BasicCar nano_base = new Nano("Green Nano") {Price = 100000};
```

```
BasicCar ford_base = new Ford("Ford Yellow") {Price = 500000};  
BasicCar bc1;  
//Nano  
bc1 = nano_base.Clone();  
bc1.Price = nano_base.Price+BasicCar.SetPrice();  
Console.WriteLine("Car is: {0}, and it's price is Rs. {1}  
",bc1.ModelName,bc1.Price);  
  
//Ford  
bc1 = ford_base.Clone();  
bc1.Price = ford_base.Price+BasicCar.SetPrice();  
Console.WriteLine("Car is: {0}, and it's price is Rs. {1}",  
bc1.ModelName, bc1.Price);  
  
Console.ReadLine();  
}  
}  
}
```

Output

Here is some example output:

Prototype Pattern Demo

Car is: Green Nano, and it's price is Rs. 486026
Car is: Ford Yellow, and it's price is Rs. 886026

Note You may see a different price in your system because here I am generating a random price in the SetPrice() method inside the BasicCar class. But I have ensured that the price of Ford will be greater than Nano.

Q&A Session

1. What are the advantages of using the Prototype design pattern?

Answer:

- You can include or discard products at runtime.
- In some contexts, you can create new instances with a cheaper cost.
- You can focus on the key activities rather than focusing on complicated instance creation processes.
- Clients can ignore the complex creation process for objects and instead clone or copy objects.

2. What are the challenges associated with using the Prototype design pattern?

Answer:

- Each subclass must implement the cloning or copying mechanism.
- Implementing the cloning mechanism can be challenging if the objects under consideration do not support copying or if there are circular references.
- In this example, I have used the `MemberwiseClone()` member that performs a shallow copy in C#. Actually, it creates an object and then copies the nonstatic fields of the current object into the newly created object. MSDN further says that for a value type field, it performs a bit-by-bit copy, but for a reference type field, the references are copied but referred objects are not copied. So, the original object and the cloned object both refer to the same object. If you need a deep copy in your application, that can be expensive.

3. Can you elaborate on the difference between a shallow copy and a deep copy in C#?

Answer:

A shallow copy creates a new object and then copies the nonstatic fields from the original object to the new object. If there exists a value type field in the original object, a bit-by-bit copy is performed. But if the field is a reference type, this method will copy the reference, not the actual object. Let's try to understand the mechanism with a simple diagram; see Figure 2-5. Suppose you have an object X1 and it has a reference to another object, Y1. Further, assume that object Y1 has a reference to object Z1.

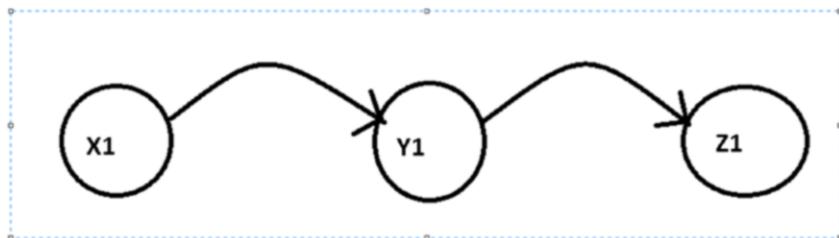


Figure 2-5. Before the shallow copy of the references

Now, with a shallow copy of X1, a new object (say, X2) will be created that will also have a reference to Y1 (see Figure 2-6).

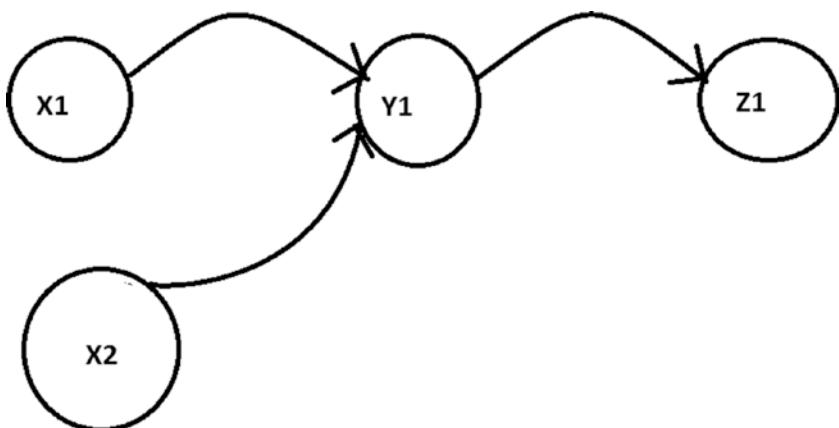


Figure 2-6. After a shallow copy of the reference

I have already used `MemberwiseClone()` in the implementation. It performs a shallow copy.

But for a deep copy of `X1`, a new object (say, `X3`) will be created, and `X3` will have a reference to the new object `Y3` that is actually a copy of `Y1`. Also, `Y3`, in turn, will have a reference to another new object, `Z3`, that is a copy of `Z1` (see Figure 2-7).

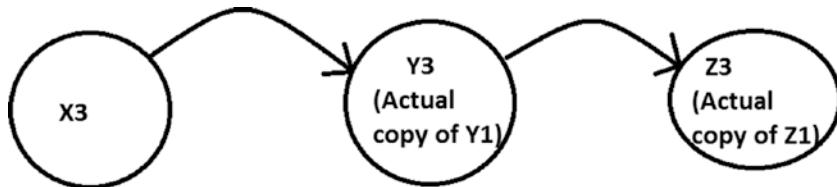


Figure 2-7. After a deep copy of the reference

To get an idea of a different type of copying technique, you can refer to this wiki page:

https://en.wikipedia.org/wiki/Object_copying#Shallow_copy

4. **Can you show a simple example that demonstrates a user-defined copy constructor?**

Answer:

Consider the following program with output for your immediate reference.

Demonstration

Here is the demonstration code:

```

using System;

namespace PrototypePatternQAs
{
    class Student
    {
        int rollNo;
        string name;
    }
}
  
```

CHAPTER 2 PROTOTYPE PATTERN

```
//Instance Constructor
public Student(int rollNo, string name)
{
    this.rollNo = rollNo;
    this.name = name;
}
//Copy Constructor
public Student(Student student)
{
    this.name = student.name;
    this.rollNo = student.rollNo;
}
public void DisplayDetails()
{
    Console.WriteLine("Student name :{0}, Roll no: {1}",
        name,rollNo);
}
}
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***A simple copy constructor demo***\n");
        Student student1 = new Student(1, "John");
        Console.WriteLine("The details of student1 is as follows:");
        student1.DisplayDetails();
        Console.WriteLine("\n Copying student1 to student2 now");
        Student student2 = new Student (student1);
        Console.WriteLine("The details of student2 is as follows:");
        student2.DisplayDetails();
        Console.ReadKey();
    }
}
```

Output

Here is some sample output:

A simple copy constructor demo

The details of student1 is as follows:

Student name :John, Roll no: 1

Copying student1 to student2 now

The details of student2 is as follows:

Student name :John, Roll no: 1

CHAPTER 3

Builder Pattern

This chapter covers the Builder pattern.

GoF Definition

Separate the construction of a complex object from its representation so that the same construction processes can create different representations.

Concept

The Builder pattern is useful for creating complex objects that have multiple parts.

The creation process of an object should be independent of these parts; in other words, the construction process does not care how these parts are assembled. In addition, you should be able to use the same construction process to create different representations of the objects.

The following structure of the pattern is adopted from the book of GoF; see Figure 3-1.

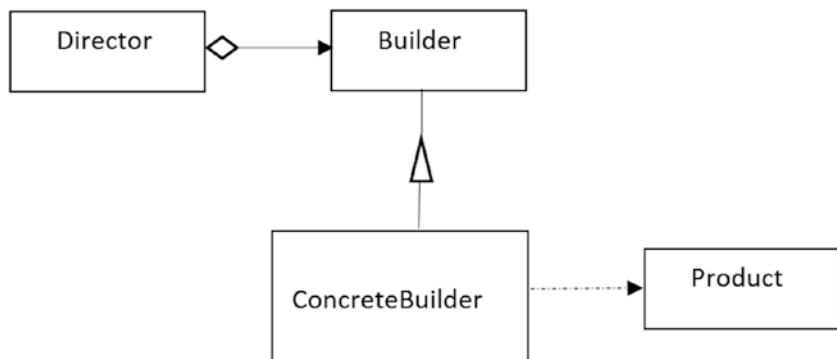


Figure 3-1. A sample of Builder pattern

Here, Product is the complex object under consideration. `ConcreteBuilder` constructs and assembles the parts of a `Product` object by implementing an abstract interface called `Builder`. The `ConcreteBuilder` objects builds the internal representations of the products and defines the creation process and assembly mechanisms. `Director` is responsible for creating the final object using the `Builder` interface.

Real-Life Example

To complete an order for a computer, different hardware parts are assembled based on customer preferences. For example, a customer can opt for a 500GB hard disk with an Intel processor, and another customer can choose a 250GB hard disk with an AMD processor.

Computer World Example

You can use this pattern when you want to convert one text format to another text format, such as converting from RTF to ASCII.

Illustration

This example has the following participants: `IBuilder`, `Car`, `MotorCycle`, `Product`, and `Director`. `IBuilder` is used to create parts of the `Product` object, where `Product` represents the complex object under construction. `Car` and `MotorCycle` are the concrete implementations of the `IBuilder` interface. They implement the `IBuilder` interface. That's why they need to supply the body for these methods: `StartUpOperations()`, `BuildBody()`, `InsertWheels()`, `AddHeadlights()`, `EndOperations()`, and `GetVehicle()`. The first five methods are straightforward; they are used to perform some operation at the beginning, build the body of the vehicle, insert a number of wheels into it, add headlights to the vehicle, and perform an operation at the end, respectively. `GetVehicle()` returns the ultimate product. Finally, `Director` is responsible for constructing the final representation of these products using the `IBuilder` interface. (See the structure defined by GoF in Figure 3-1.) Notice that `Director` is calling the same `Construct()` method to create different types of vehicles.

Now let's go through the code to see how different parts are assembled for this pattern.

Class Diagram

Figure 3-2 shows the class diagram.

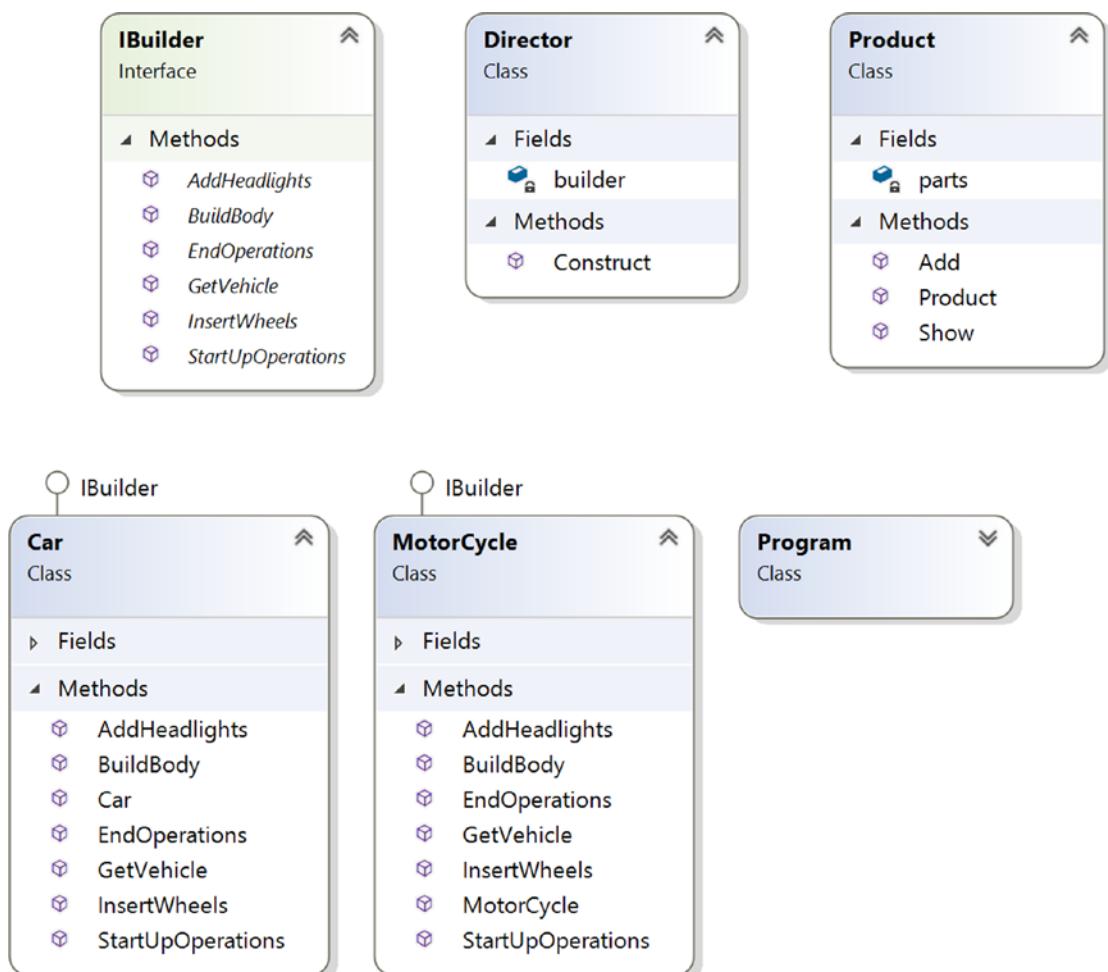


Figure 3-2. Class diagram

Solution Explorer View

Figure 3-3 shows the high-level structure of the parts of the program.

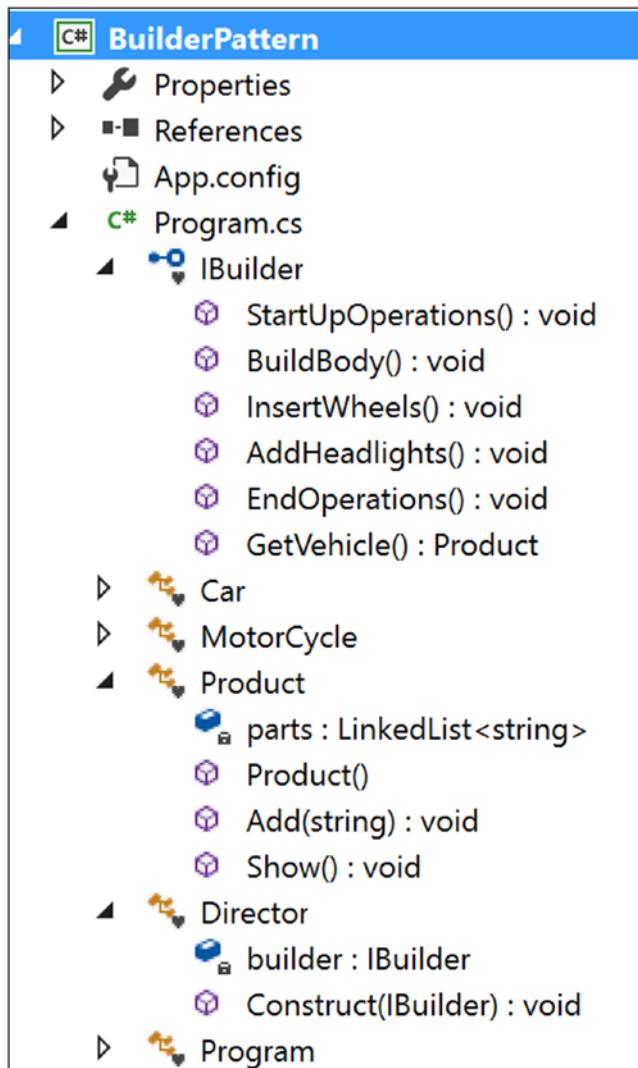


Figure 3-3. Solution Explorer View

Implementation

Here's the implementation:

```
using System;
using System.Collections.Generic;//For LinkedList

namespace BuilderPattern
{
```

```
// Builders common interface
interface IBuilder
{
    void StartUpOperations();
    void BuildBody();
    void InsertWheels();
    void AddHeadlights();
    void EndOperations();
    Product GetVehicle();
}

// ConcreteBuilder: Car
class Car : IBuilder
{
    private string brandName;
    private Product;
    public Car(string brand)
    {
        product = new Product();
        this.brandName = brand;
    }
    public void StartUpOperations()
    {   //Starting with brandname
        product.Add(string.Format("Car Model name :{0}",this.
        brandName));
    }
    public void BuildBody()
    {
        product.Add("This is a body of a Car");
    }
    public void InsertWheels()
    {
        product.Add("4 wheels are added");
    }
    public void AddHeadlights()
    {
```

```
        product.Add("2 Headlights are added");
    }
    public void EndOperations()
    {   //Nothing in this case
    }

    public Product GetVehicle()
    {
        return product;
    }
}

// ConcreteBuilder:Motorcycle
class MotorCycle : IBuilder
{
    private string brandName;
    private Product product;
    public MotorCycle(string brand)
    {
        product = new Product();
        this.brandName = brand;
    }
    public void StartUpOperations()
    {   //Nothing in this case
    }

    public void BuildBody()
    {
        product.Add("This is a body of a Motorcycle");
    }

    public void InsertWheels()
    {
        product.Add("2 wheels are added");
    }

    public void AddHeadlights()
    {
```

```
        product.Add("1 Headlights are added");
    }
    public void EndOperations()
    {
        //Finishing up with brandname
        product.Add(string.Format("Motorcycle Model name :{0}",
        this.brandName));
    }
    public Product GetVehicle()
    {
        return product;
    }
}

// "Product"
class Product
{
    // We can use any data structure that you prefer e.g.List<string> etc.
    private LinkedList<string> parts;
    public Product()
    {
        parts = new LinkedList<string>();
    }

    public void Add(string part)
    {
        //Adding parts
        parts.AddLast(part);
    }

    public void Show()
    {
        Console.WriteLine("\nProduct completed as below :");
        foreach (string part in parts)
            Console.WriteLine(part);
    }
}
```

CHAPTER 3 BUILDER PATTERN

```
// "Director"
class Director
{
    IBuilder builder;
    // A series of steps-in real life, steps are complex.
    public void Construct(IBuilder builder)
    {
        this.builder = builder;
        builder.StartUpOperations();
        builder.BuildBody();
        builder.InsertWheels();
        builder.AddHeadlights();
        builder.EndOperations();
    }
}
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine(" ***Builder Pattern Demo***");
        Director director = new Director();

        IBuilder b1 = new Car("Ford");
        IBuilder b2 = new MotorCycle("Honda");

        // Making Car
        director.Construct(b1);
        Product p1 = b1.GetVehicle();
        p1.Show();

        //Making MotorCycle
        director.Construct(b2);
        Product p2 = b2.GetVehicle();
        p2.Show();

        Console.ReadLine();
    }
}
```

Output

Here's the output:

Builder Pattern Demo

Product completed as below :

Car Model name :Ford

This is a body of a Car

4 wheels are added

2 Headlights are added

Product completed as below :

This is a body of a Motorcycle

2 wheels are added

1 Headlights are added

Motorcycle Model name :Honda

Q&A Session

1. What is the advantage of using a Builder pattern?

Answer:

- You direct the builder to build the objects step-by-step, and you promote encapsulation by hiding the details of the complex construction process. The director can retrieve the final product from the builder when the whole construction is over. In general, at a high level, you seem to have only one method that makes the complete product, but other internal methods are involved in the creation process. So, you have finer control over the construction process.
- Using this pattern, the same construction process can produce different products.
- You can also vary the internal representation of products.

2. What are the drawbacks associated with a Builder pattern?

Answer:

- It is not suitable if you want to deal with mutable objects (which can be modified later).
- You may need to duplicate some portion of the code. These duplications may have significant impact in some contexts.
- To create more products, you need to create more concrete builders.

3. Could you use an abstract class instead of the interface in the illustration of this pattern?

Answer:

Yes. You could use an abstract class instead of an interface in this example.

4. How can you decide whether to use an abstract class or an interface in an application?

Answer:

If you want to have some centralized or default behaviors, an abstract class is a better choice. In those cases, you can provide some default implementation. On the other hand, the interface implementation starts from scratch and indicates some kind of rules/contracts such as what is to be done, but it does not enforce the “how” part upon you. Also, interfaces are preferred when you are trying to implement the concept of multiple inheritance.

Remember that if you need to add a new method in an interface, then you need to track down all the implementations of that interface, and you need to put the concrete implementation for that method in all those places. In such a case, an abstract class is a better choice because you can add a new method in an abstract class with a default implementation, and the existing code can run smoothly.

MSDN provides following recommendations:

- If you anticipate creating multiple versions of your component, create an abstract class. Abstract classes provide a simple and easy way to version your components. By updating the base class, all inheriting classes are automatically updated with the change. Interfaces, on the other hand, cannot be changed once created. If a new version of an interface is required, you must create a whole new interface.
 - If the functionality you are creating will be useful across a wide range of disparate objects, use an interface. Abstract classes should be used primarily for objects that are closely related, whereas interfaces are best suited for providing common functionality to unrelated classes.
 - If you are designing small, concise bits of functionality, use interfaces. If you are designing large functional units, use an abstract class.
 - If you want to provide common, implemented functionality among all implementations of your component, use an abstract class. Abstract classes allow you to partially implement your class, whereas interfaces contain no implementation for any members.
- 5. In the example, for cars, the model names are added in the beginning, but for motorcycles, the model names are added at the end. Is it intentional?**

Answer:

Yes. I did this to demonstrate the fact that each of the concrete builders can decide how it wants to produce the final products. They have this freedom.

CHAPTER 4

Factory Method Pattern

This chapter covers the Factory Method pattern.

GoF Definition

Define an interface for creating an object, but let subclasses decide which class to instantiate. The Factory Method pattern lets a class defer instantiation to subclasses.

Note To understand this pattern, I suggest you refer to Chapter 24, which covers the Simple Factory pattern. The Simple Factory pattern does not fall directly into the Gang of Four design patterns, so the discussion of that pattern appears in Part II of the book. The Factory Method pattern will make more sense to you if you can understand the pros and cons of the Simple Factory pattern first.

Concept

The concept can be best described with the following examples.

Real-Life Example

In a restaurant, based on customer inputs, a chef varies the taste of dishes to make the final products.

Computer World Example

In an application, you may have different database users. For example, one user may use Oracle, and the other may use SQL Server. Whenever you need to insert data into your database, you need to create either a `SqlConnection` or an `OracleConnection` and only then can you proceed. If you put the code into `if-else` (or `switch`) statements, you need to repeat a lot of code, which isn't easily maintainable. This is because whenever you need to support a new type of connection, you need to reopen your code and make those modifications. This type of problem can be resolved using the Factory Method pattern. Here I'll provide an abstract creator class (`IAnimalFactory`) to define the basic structure. As per the definition, the instantiation process will be carried out through the subclasses that derive from this abstract class.

Illustration

I have created all the classes in a single file. There is no need to create separate folders.

Class Diagram

Figure 4-1 shows the class diagram.

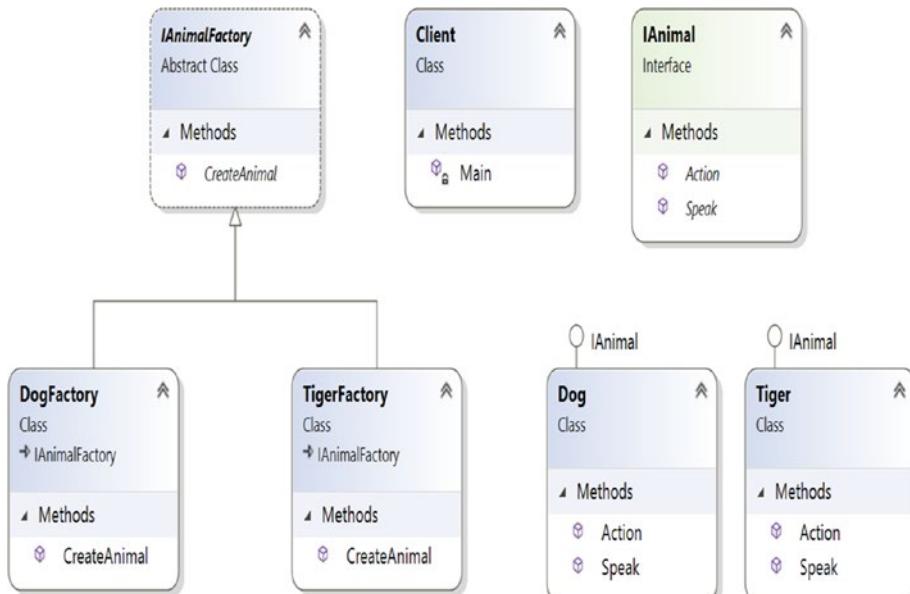


Figure 4-1. Class diagram

Directed Graph Document

Figure 4-2 shows the directed graph document.

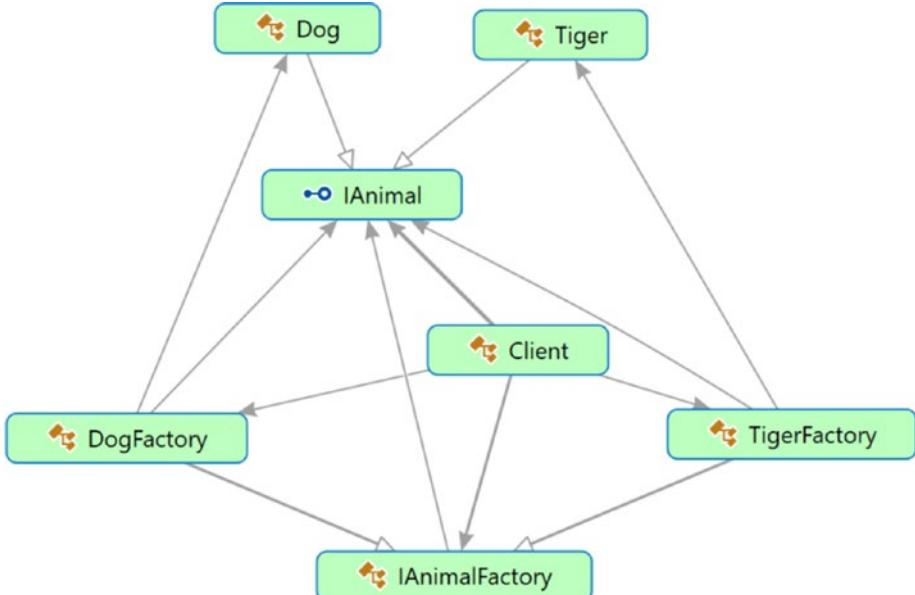


Figure 4-2. Directed Graph Document

Solution Explorer View

Figure 4-3 shows the high-level structure of the parts of the program.

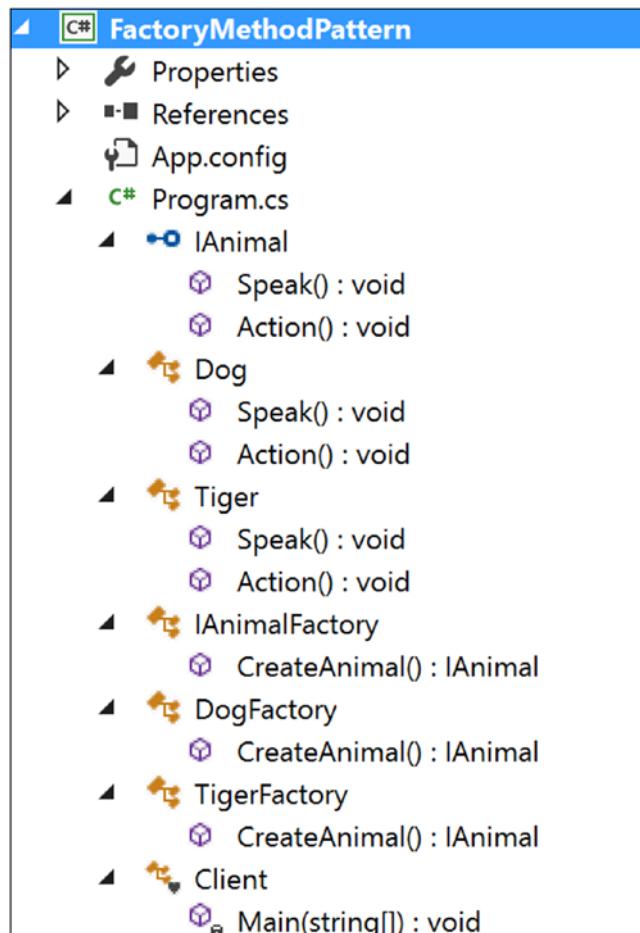


Figure 4-3. Solution Explorer View

Implementation

Here is the implementation:

```
using System;

namespace FactoryMethodPattern
{
    public interface IAnimal
    {
        void Speak();
        void Action();
    }
}
```

```

public class Dog : IAnimal
{
    public void Speak()
    {
        Console.WriteLine("Dog says: Bow-Wow.");
    }
    public void Action()
    {
        Console.WriteLine("Dogs prefer barking...\n");
    }
}
public class Tiger : IAnimal
{
    public void Speak()
    {
        Console.WriteLine("Tiger says: Halum.");
    }
    public void Action()
    {
        Console.WriteLine("Tigers prefer hunting...\n");
    }
}
public abstract class IAnimalFactory
{
//Remember the GoF definition which says "....Factory method lets a class
//defer instantiation to subclasses." Following method will create a Tiger
//or Dog But at this point it does not know whether it will get a Dog or a
//Tiger. It will be decided by the subclasses i.e.DogFactory or TigerFactory.
//So, the following method is acting like a factory (of creation).
    public abstract IAnimal CreateAnimal();
}
public class DogFactory : IAnimalFactory
{
    public override IAnimal CreateAnimal()
    {

```

CHAPTER 4 FACTORY METHOD PATTERN

```
//Creating a Dog
    return new Dog();
}
}

public class TigerFactory : IAnimalFactory
{
    public override IAnimal CreateAnimal()
    {
        //Creating a Tiger
        return new Tiger();
    }
}

class Client
{
    static void Main(string[] args)
    {
        Console.WriteLine("***Factory Pattern Demo***\n");
        // Creating a Tiger Factory
        IAnimalFactory tigerFactory =new TigerFactory();
        // Creating a tiger using the Factory Method
        IAnimal aTiger = tigerFactory.CreateAnimal();
        aTiger.Speak();
        aTiger.Action();

        // Creating a DogFactory
        IAnimalFactory dogFactory = new DogFactory();
        // Creating a dog using the Factory Method
        IAnimal aDog = dogFactory.CreateAnimal();
        aDog.Speak();
        aDog.Action();

        Console.ReadKey();
    }
}
}
```

Output

Here is some output:

```
***Factory Pattern Demo***
```

```
Tiger says: Halum.  
Tigers prefer hunting...  
  
Dog says: Bow-Wow.  
Dogs prefer barking...
```

Modified Implementation

In this modified implementation, more flexibilities are added. Notice that, the `IAnimalFactory` class is an abstract class. So, you can take the advantage of using an abstract class. Suppose you want a subclass to follow a rule that can be imposed from its parent (or base) class. I have tested such a scenario in the following design.

Here are the key characteristics of the design:

- Only `IAnimalFactory` is modified, as shown here. In other words, I am introducing a new method called `MakeAnimal()`.

```
//Modifying the IAnimalFactory class.  
public abstract class IAnimalFactory  
{  
    public IAnimal MakeAnimal()  
    {  
        Console.WriteLine("\n IAnimalFactory.MakeAnimal()-You  
cannot ignore parent rules.");  
        /*  
         At this point, it doesn't know whether it will get a  
         Dog or a Tiger. It will be decided by the subclasses  
         i.e.DogFactory or TigerFactory. But it knows that it  
         will Speak and it will have a preferred way of Action.  
        */  
    }  
}
```

```

        IAnimal animal = CreateAnimal();
        animal.Speak();
        animal.Action();
        return animal;
    }
    //So, the following method is acting like a factory
    //(of creation).
    public abstract IAnimal CreateAnimal();
}

```

- The client code has these changes:

```

class Client
{
    static void Main(string[] args)
    {
        Console.WriteLine("***Beautification to Factory
Pattern Demo***\n");
        // Creating a tiger using the Factory Method
        IAnimalFactory tigerFactory = new TigerFactory();
IAnimal aTiger = tigerFactory.MakeAnimal();
        //IAnimal aTiger = tigerFactory.CreateAnimal();
        //aTiger.Speak();
        //aTiger.Action();

        // Creating a dog using the Factory Method
        IAnimalFactory dogFactory = new DogFactory();
IAnimal aDog = dogFactory.MakeAnimal();
        //IAnimal aDog = dogFactory.CreateAnimal();
        //aDog.Speak();
        //aDog.Action();

        Console.ReadKey();
    }
}

```

Modified Output

Here is the modified output:

Beautification to Factory Pattern Demo

AnimalFactory.MakeAnimal()-You cannot ignore parent rules.

Tiger says: Halum.

Tigers prefer hunting...

AnimalFactory.MakeAnimal()-You cannot ignore parent rules.

Dog says: Bow-Wow.

Dogs prefer barking...

Analysis

Notice that in each case you see the following warning: "...You cannot ignore parent rules."

Q&A Session

1. Why have you separated the **CreateAnimal()** method from client code?

Answer:

This is on purpose. I want the subclasses to create specialized objects. If you look carefully, you will also find that only this "creational part" varies across the products. I discussed this in detail in the "Q&A Session" section of Chapter [24](#).

2. What are the advantages of using a factory like this?

Answer:

- You are separating the code that varies from the code that does not vary (in other words, the advantages of using the Simple Factory pattern are still present). This helps you to maintain the code easily.

- The code is not tightly coupled, so you can add new classes such as Lion, Bear, and so on, at any time in the system without modifying the existing architecture. In other words, I have followed the “closed for modification but open for extension” principle.

3. What are the challenges of using a factory like this?

Answer:

If you need to deal with many different types of objects, then the overall performance of the system can be affected.

4. I am seeing that the Factory Method pattern is supporting two parallel hierarchies. Is this understanding correct?

Answer:

Good catch. Yes, from the class diagram, it is evident that this pattern supports parallel class hierarchies; see Figure 4-4.

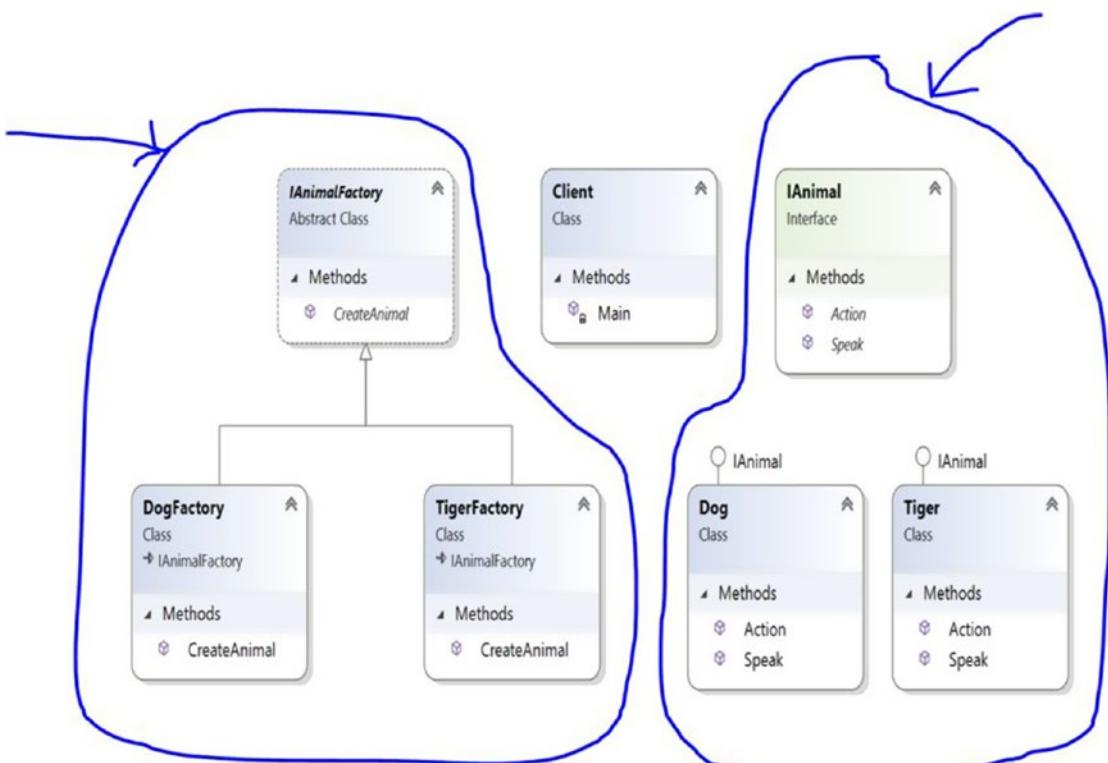


Figure 4-4. The two class hierarchies in this example

In this example, `IAnimalFactory`, `DogFactory`, and `TigerFactory` are placed in one hierarchy, and `IAnimal`, `Dog`, and `Tiger` are placed in another hierarchy. So, you can see that creators and their creations/products are the two hierarchies that are running in parallel.

5. **You should always mark the factory method with an `abstract` keyword so that subclasses can complete them. Is that understanding correct?**

Answer:

No. Sometimes you may be interested in a default factory method if the creator has no subclasses. In that case, you cannot mark the factory method with an `abstract` keyword.

However, to see the real power of the Factory Method pattern, you may need to follow the design that is implemented here.

6. **It still appears to me that the Factory Method pattern is not that much different from the Simple Factory pattern. Is that understanding correct?**

Answer:

If you look at the subclasses in the examples in both chapters, you may find some similarities. But you should not forget the key aim of the Factory Method pattern; it is supplying you with the framework through which different subclasses can make different products. In the case of the Simple Factory pattern, you cannot vary the products in a similar manner. You can think of the Simple Factory pattern as a one-time deal, but most important, your creational part will not be closed for modification. Whenever you want to add something new, you need to add an `if-else` block or a `switch` statement in the factory class of your Simple Factory pattern.

In this context, remember the GoF definition (“The Factory Method pattern lets a class defer instantiation to subclasses.”). So, in the Simple Factory pattern demonstration, you could omit the abstract class `IAnimalFactory` and its abstract method `CreateAnimal()` and instead use only one `SimpleFactory` class. In that case, you would not need to override the `CreateAnimal()` method; in addition, it’s considered a good practice to code to an interface/abstract class (as in this case). Also, this mechanism provides you with the flexibility to put some common behavior in the abstract class.

CHAPTER 5

Abstract Factory Pattern

This chapter covers the Abstract Factory pattern.

GoF Definition

Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Note To understand this pattern, I suggest you refer to Chapter 24 and learn about the Simple Factory pattern first. Then you can turn to Chapter 4 to learn about the Factory Method pattern. The Simple Factory pattern does not fall directly into the Gang of Four design patterns, so the discussion of that pattern appears in Part II of the book. The Abstract Factory pattern will make more sense to you if you have an understanding of both the Simple Factory pattern and the Factory Method pattern.

Concept

An abstract factory is called a *factory of factories*. In this pattern, you provide a way to encapsulate a group of individual factories that have a common theme. In this process, you do not mention or specify their concrete classes.

This pattern helps you to interchange specific implementations without changing the code that uses them, even at runtime. However, it may result in unnecessary complexity and extra work. Even debugging becomes tough in some cases.

Real-Life Example

Suppose you are decorating your room with two different types of tables; one is made of wood and one of steel. For the wooden type, you need to visit to a carpenter, and for the other type, you may need to go to a metal shop. All of these are table factories. So, based on demand, you decide what kind of factory you need.

Computer World Example

ADO.NET has already implemented similar concepts to establish a connection to a database.

To understand this pattern, you may want to extend your understanding of the Factory Method pattern. In the Factory Method pattern, you have two factories; one is for creating dogs and another is for creating tigers. But now suppose, you want to categorize dogs and tigers further; you choose to get a pet animal (dog or tiger) or a wild animal (dog or tiger) through the factories. To fulfill that demand, you introduce two concrete factories: `WildAnimalFactory` and `PetAnimalFactory`. `WildAnimalFactory` is responsible for creating wild animals, and `PetAnimalFactory` is responsible for creating pet animals.

Illustration

Wikipedia describes a typical structure of this pattern, which is similar to Figure 5-1 (https://en.wikipedia.org/wiki/Abstract_factory_pattern).

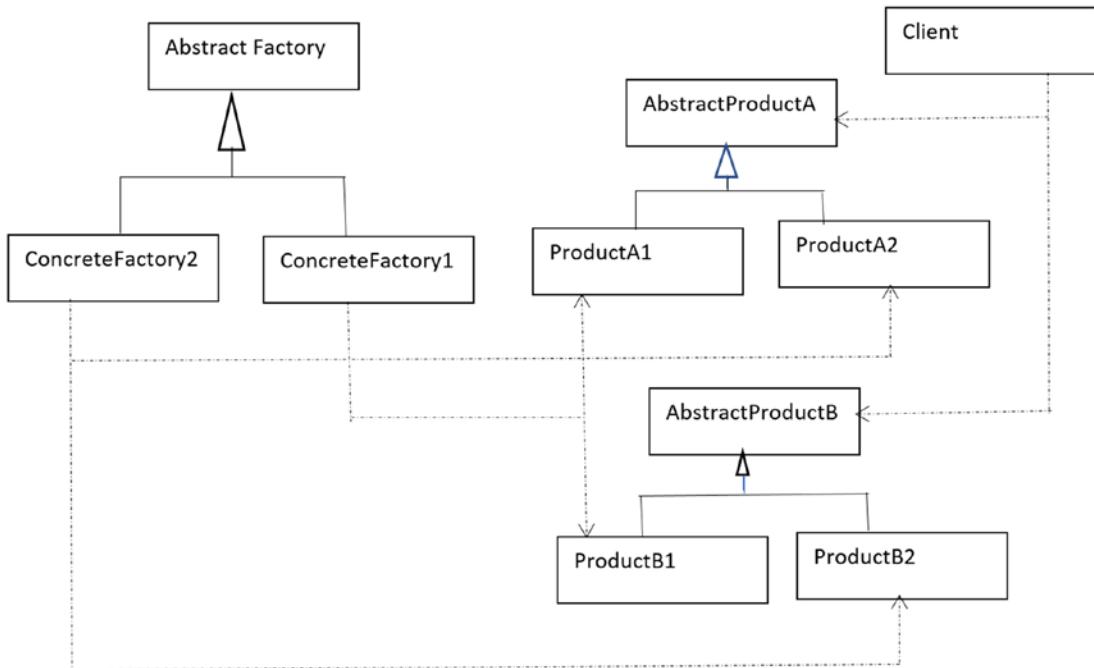


Figure 5-1. Abstract Factory pattern

I will follow a similar structure in the implementation in this chapter. `Program.cs` is the client looking for some animals (which are dogs and tigers in this case). You are exploring the construction process of both pets and wild animals in this implementation.

In this example, you have two concrete factories: `WildAnimalFactory` and `PetAnimalFactory`. You can guess that they are responsible for creating the concrete products of dogs and tigers. `WildAnimalFactory` will create wild animals (wild dogs and wild tigers), and `PetAnimalFactory` will create pet animals (pet dogs and pet tigers). For your reference, the participants and their roles are summarized here:

- `IAnimalFactory`: Abstract factory.
- `WildAnimalFactory`: Concrete factory. This will create wild dogs and wild tigers.
- `PetAnimalFactory`: Another concrete factory. This will create pet dogs and pet tigers.
- `ITiger` and `IDog`: Abstract products in this case.
- `PetTiger`, `PetDog`, `WildTiger`, and `WildDog`: These are the concrete products in this example.

Class Diagram

Figure 5-2 shows the class diagram.

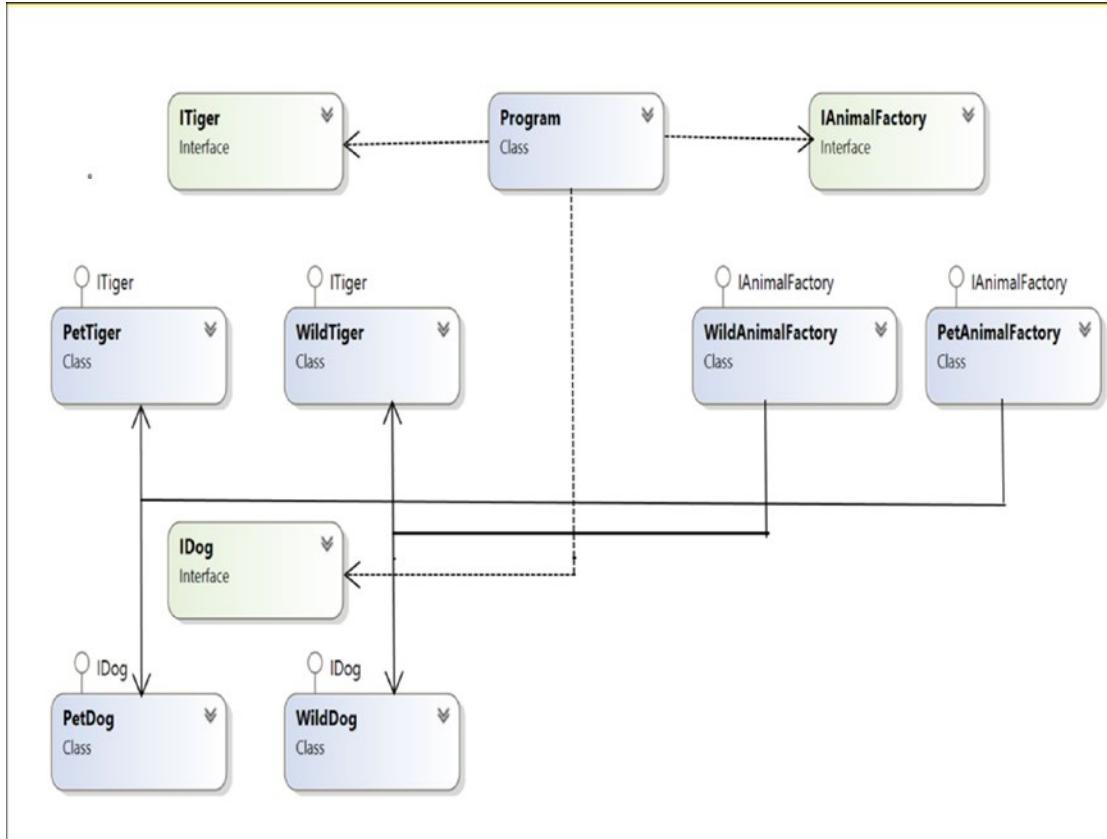


Figure 5-2. Class diagram

Solution Explorer View

Figure 5-3 shows the high-level structure of the parts of the program.

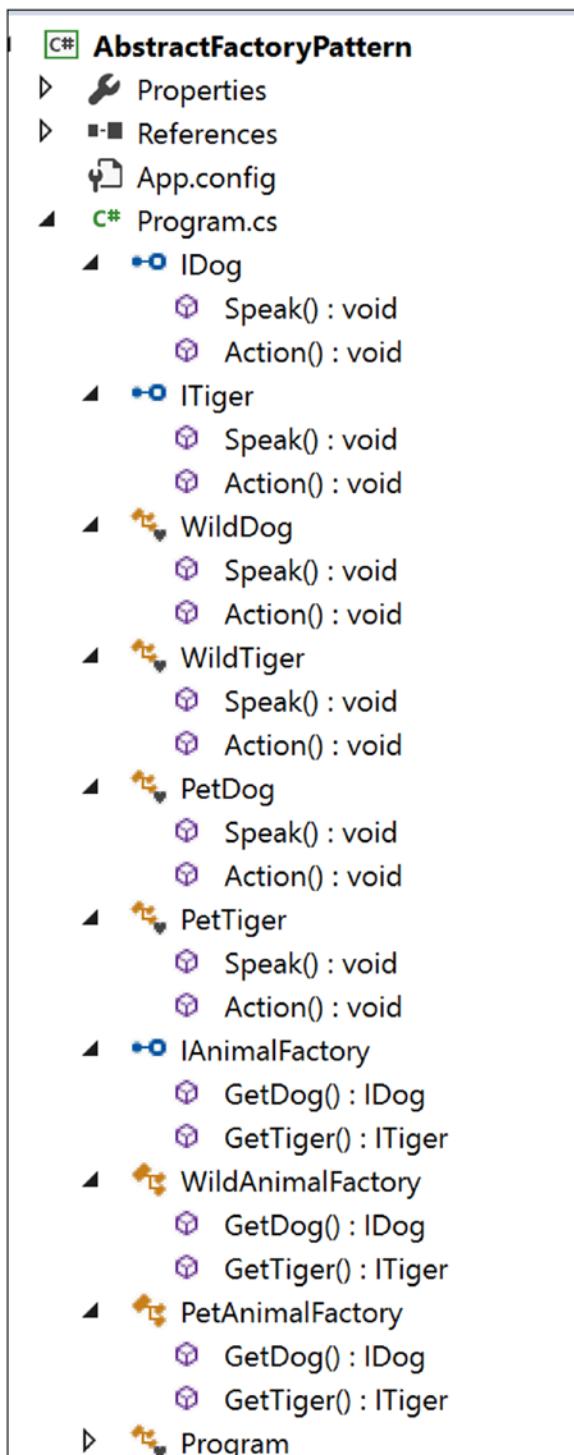


Figure 5-3. Solution Explorer View

Implementation

Here's the implementation:

```
using System;

namespace AbstractFactoryPattern
{
    public interface IDog
    {
        void Speak();
        void Action();
    }

    public interface ITiger
    {
        void Speak();
        void Action();
    }

    #region Wild Animal collections

    class WildDog : IDog
    {
        public void Speak()
        {
            Console.WriteLine("Wild Dog says: Bow-Wow.");
        }

        public void Action()
        {
            Console.WriteLine("Wild Dogs prefer to roam freely in
jungles.\n");
        }
    }

    class WildTiger : ITiger
    {
        public void Speak()
        {
            Console.WriteLine("Wild Tiger says: Halum.");
        }
    }
}
```

```
}

public void Action()
{
    Console.WriteLine("Wild Tigers prefer hunting in jungles.\n");
}

}

#endregion

#region Pet Animal collections

class PetDog : IDog
{
    public void Speak()
    {
        Console.WriteLine("Pet Dog says: Bow-Wow.");
    }

    public void Action()
    {
        Console.WriteLine("Pet Dogs prefer to stay at home.\n");
    }
}

class PetTiger : ITiger
{
    public void Speak()
    {
        Console.WriteLine("Pet Tiger says: Halum.");
    }

    public void Action()
    {
        Console.WriteLine("Pet Tigers play in an animal circus.\n");
    }
}

#endregion

//Abstract Factory
public interface IAnimalFactory
{
    IDog GetDog();
```

CHAPTER 5 ABSTRACT FACTORY PATTERN

```
    ITiger GetTiger();
}
//Concrete Factory-Wild Animal Factory
public class WildAnimalFactory : IAnimalFactory
{
    public IDog GetDog()
    {
        return new WildDog();
    }

    public ITiger GetTiger()
    {
        return new WildTiger();
    }
}

//Concrete Factory-Pet Animal Factory
public class PetAnimalFactory : IAnimalFactory
{
    public IDog GetDog()
    {
        return new PetDog();
    }

    public ITiger GetTiger()
    {
        return new PetTiger();
    }
}

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***Abstract Factory Pattern Demo***\n");

        //Making a wild dog through WildAnimalFactory
        IAnimalFactory wildAnimalFactory = new WildAnimalFactory();
        IDog wildDog = wildAnimalFactory.GetDog();
```

```

        wildDog.Speak();
        wildDog.Action();
        //Making a wild tiger through WildAnimalFactory
        ITiger wildTiger = wildAnimalFactory.GetTiger();
        wildTiger.Speak();
        wildTiger.Action();

        Console.WriteLine("*****");
        //Making a pet dog through PetAnimalFactory
        IAnimalFactory petAnimalFactory = new PetAnimalFactory();
        IDog petDog = petAnimalFactory.GetDog();
        petDog.Speak();
        petDog.Action();
        //Making a pet tiger through PetAnimalFactory
        ITiger petTiger = petAnimalFactory.GetTiger();
        petTiger.Speak();
        petTiger.Action();

        Console.ReadLine();
    }
}
}

```

Output

Here's the output:

```

***Abstract Factory Pattern Demo***

Wild Dog says: Bow-Wow.
Wild Dogs prefer to roam freely in jungles.

Wild Tiger says: Halum.
Wild Tigers prefer hunting in jungles.

*****
Pet Dog says: Bow-Wow.
Pet Dogs prefer to stay at home.

Pet Tiger says: Halum.
Pet Tigers play in an animal circus.

```

Q&A Session

1. I am seeing that both the **IDog** and **ITiger** interfaces are containing methods that have the same names. For example, both interfaces contain the methods **Speak()** and **Action()**. Is that mandatory?

Answer:

No. You can use different names for your methods. Also, the number of methods can be different in these interfaces. However, in Chapter 24, I covered the Simple Factory pattern, and in Chapter 4, I covered the Factory Method pattern. In this chapter, I am continuing the examples from those chapters, which is why I have kept the methods **Speak()** and **Action()** in this example.

2. What are the challenges of using an abstract factory like this?

Answer:

Any change in the abstract factory will force you to propagate the modification to the concrete factories. If you follow the design philosophy that says “Program to an interface, not to an implementation,” you need to prepare for this. This is one of the key principles that developers should always keep in mind. In most scenarios, developers do not want to change their abstract factories.

In addition, the overall architecture may look complex. Also, debugging becomes tricky in some scenarios.

3. How can you distinguish a Simple Factory pattern from Factory Method pattern or an Abstract Factory pattern?

Answer:

I discussed the differences of a Simple Factory pattern and a Factory Method pattern in the “Q&A Session” section of Chapter 4.

Let's revise all three factories as shown in the following diagrams.

Here's the Simple Factory pattern:

```
IAnimal preferredType=null;
ISimpleFactory simpleFactory = new SimpleFactory();
#region The code region that will vary based on users preference
preferredType = simpleFactory.CreateAnimal();
```

Figure 5-4 shows the Simple Factory pattern.

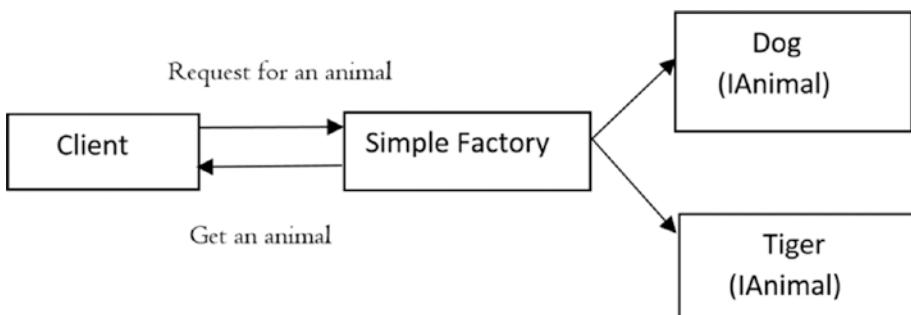


Figure 5-4. Simple Factory pattern

Here's the Factory Method pattern:

```
// Creating a Tiger Factory
IAnimalFactory tigerFactory =new TigerFactory();
// Creating a tiger using the Factory Method
IAnimal aTiger = tigerFactory.CreateAnimal();

.....
// Creating a DogFactory
IAnimalFactory dogFactory = new DogFactory();
// Creating a dog using the Factory Method
IAnimal aDog = dogFactory.CreateAnimal();
```

Figure 5-5 shows the Factory Method pattern.

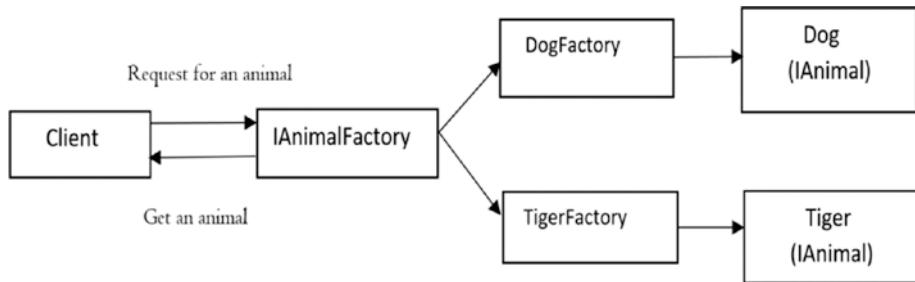


Figure 5-5. Factory Method pattern

Here's the Abstract Factory pattern:

```

//Making a wild dog through WildAnimalFactory
IAnimalFactory wildAnimalFactory = new WildAnimalFactory();
IDog wildDog = wildAnimalFactory.GetDog();
.....
//Making a wild tiger through WildAnimalFactory
ITiger wildTiger = wildAnimalFactory.GetTiger();
.....
//Making a pet dog through PetAnimalFactory
IAnimalFactory petAnimalFactory = new PetAnimalFactory();
IDog petDog = petAnimalFactory.GetDog();
.....
//Making a pet tiger through PetAnimalFactory
ITiger petTiger = petAnimalFactory.GetTiger();
  
```

Figure 5-6 shows the Abstract Factory pattern.

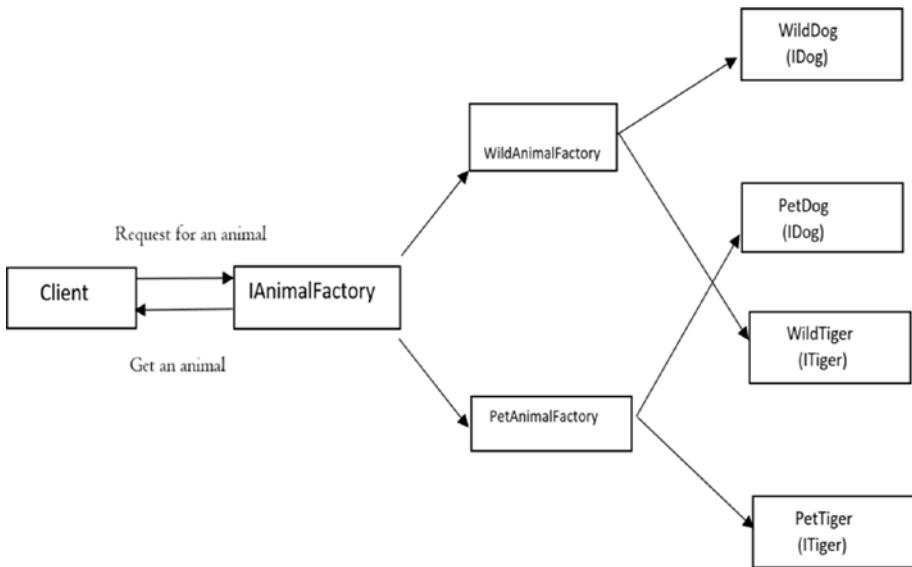


Figure 5-6. Abstract Factory pattern

Conclusion

In short, with the Simple Factory pattern, you can separate the code that will vary from the rest of the code (basically, you decouple the client code). This approach helps you to manage the code more easily. Another key advantage of this approach is that the client is unaware of how the objects are created. So, it promotes both security and abstraction.

However, this approach can violate the open-close principle. You can overcome this drawback using the Factory Method pattern, which allows subclasses to decide how the instantiation process will be completed. Put simply, you delegate the object creation to the subclasses that implement the factory method to create objects.

The abstract factory is basically factory of factories. It creates the family of related objects, but it does not depend on the concrete classes.

Lastly, I tried to keep the examples simple. A factory method promotes inheritance, and its subclasses need to implement the factory method to create objects. But in some implementations, you may clearly notice that the Abstract Factory pattern can promote object composition by creating the related objects using the methods that are exposed in a factory interface.

These factories promote loose coupling by reducing the dependencies on concrete classes.

I.B: Structural Patterns

CHAPTER 6

Proxy Pattern

This chapter covers the Proxy pattern.

GoF Definition

Provide a surrogate or placeholder for another object to control access to it.

Concept

A proxy is basically a substitute for an intended object. When a client deals with a proxy object, it thinks that it is dealing with the actual object. You need to support this kind of design because dealing with an original object is not always possible. This is because of many factors such as security issues, for example. So, in this pattern, you may want to use a class that can perform as an interface to something else.

Real-Life Example

In a classroom, when one student is absent, his best friend may try to mimic his voice during roll call to try to get the teacher to think his friend is there.

Computer World Example

An ATM implementation will hold proxy objects for bank information that exists on a remote server. In the real programming world, creating multiple instances of a complex object (a heavy object) is costly in general. So, whenever you can, you should create multiple proxy objects that can point to the original object. This mechanism can also help you to save the computer/system memory.

Illustration

In this program, you are calling the `DoSomeWork()` method of the proxy object that, in turn, calls the `DoSomeWork()` method of an object of `ConcreteSubject`. When customers see the output, they think they have invoked the method from their intended object directly.

Class Diagram

Figure 6-1 shows the class diagram.

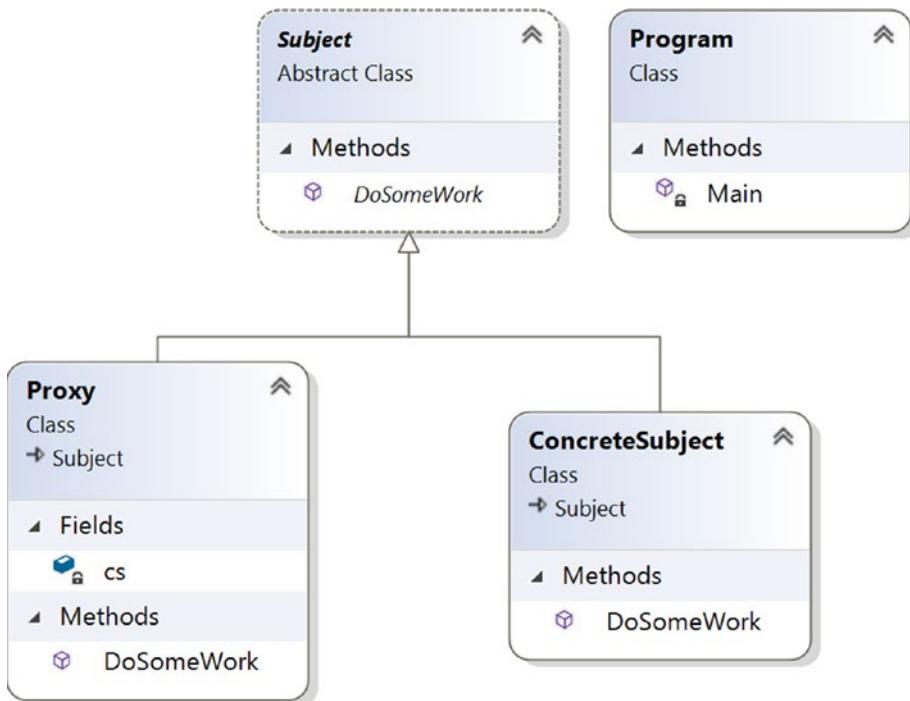


Figure 6-1. Class diagram

Directed Graph Document

Figure 6-2 shows the directed graph document.

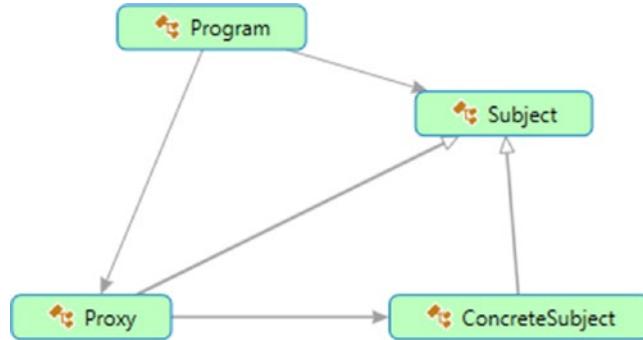


Figure 6-2. Directed Graph Document

Solution Explorer View

Figure 6-3 shows the high-level structure of the parts of the program. (Note that you could separate the proxy class into a different file. But since the parts are small in this example, I have put everything in a single file.)

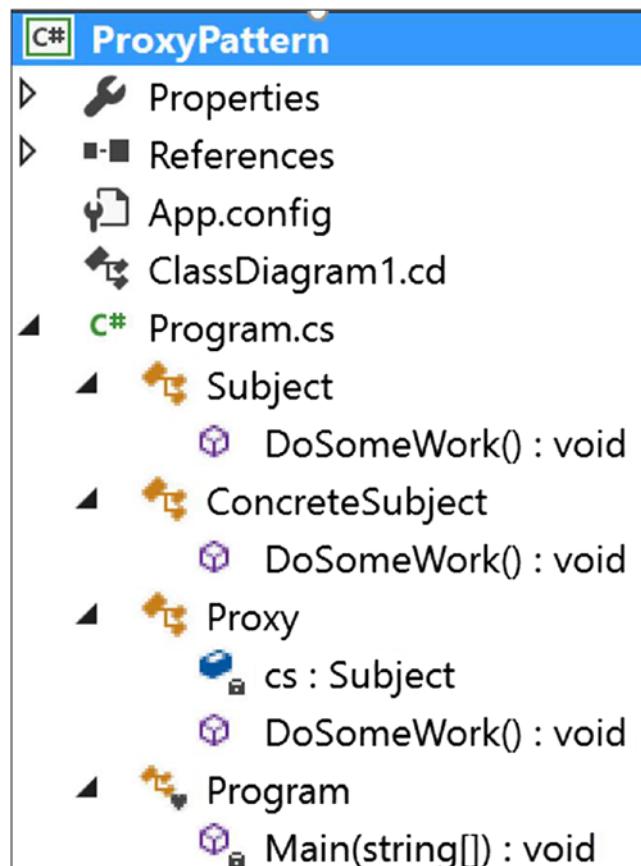


Figure 6-3. Solution Explorer View

Implementation

Here's the implementation:

```
using System;

namespace ProxyPattern
{
    /// <summary>
    /// Abstract class Subject
    /// </summary>
    public abstract class Subject
    {
```

```
    public abstract void DoSomeWork();
}

/// <summary>
/// ConcreteSubject class
/// </summary>
public class ConcreteSubject : Subject
{
    public override void DoSomeWork()
    {
        Console.WriteLine("ConcreteSubject.DoSomeWork()");
    }
}

/// <summary>
/// Proxy class
/// </summary>
public class Proxy : Subject
{
    Subject cs;

    public override void DoSomeWork()
    {
        Console.WriteLine("Proxy call happening now...");
        //Lazy initialization:We'll not instantiate until the method is
        //called
        if (cs == null)
        {
            cs = new ConcreteSubject();
        }
        cs.DoSomeWork();
    }
}

class Program
{
    static void Main(string[] args)
    {
```

```
Console.WriteLine("***Proxy Pattern Demo***\n");
Proxy px = new Proxy();
px.DoSomeWork();
Console.ReadKey();
}
}
}
```

Output

Here's the output:

```
***Proxy Pattern Demo***

Proxy call happening now...
ConcreteSubject.DoSomeWork()
```

Q&A Session

1. What are the different types of proxies?

Answer:

These are the common types of proxies:

- *Remote proxies*: These proxies can hide an object that stays in a different address space.
- *Virtual proxies*: These proxies are used to perform optimization techniques such as creating a heavy object on an on-demand basis.
- *Protection proxies*: These proxies generally deal with different access rights.
- *Smart reference*: This can perform some additional housekeeping when an object is accessed by a client. A typical operation may include counting the number of references to an object at a particular moment.

2. You could create the ConcreteSubject instance in the proxy class constructor as shown here:

```

/// <summary>
/// Proxy class
/// </summary>
public class Proxy : Subject
{
    Subject cs;
    public Proxy()
    {
        //Instantiating inside the constructor
        cs = new ConcreteSubject();
    }
    public override void DoSomeWork()
    {
        Console.WriteLine("Proxy call happening now..");
        cs.DoSomeWork();
    }
}

```

Is this understanding correct?

Answer:

Yes, you could do that. But if you follow this design, whenever you instantiate a proxy object, you would need to instantiate an object of the ConcreteSubject class also. So, this process may end up creating unnecessary objects.

3. But with this lazy instantiation process, you may create unnecessary objects in a multithreaded application. Is this understanding correct?

Answer:

Yes. In this book, I am presenting simple illustrations only, so I have ignored that part. In the discussions of the Singleton pattern in Chapter 1, you analyzed some alternative approaches for

dealing with multithreaded environments. You can always refer to those discussions in situations like this. (For example, in this particular scenario, you could implement a smart proxy to ensure that a particular object is locked before you grant access to the object.)

4. Can you give an example of a remote proxy?

Answer:

Suppose you want to call a method of an object but the object is running in a different address space (for example, in a different location or on a different computer). How can you proceed? With the help of remote proxies, you can call the method on the proxy object, which in turn will forward the call to the actual object that is running on the remote machine. This type of need can be realized through different well-known mechanisms such as ASP.NET, CORBA, or Java's RMI. In C# applications, you can exercise a similar mechanism with WCF (.NET Framework version 3.0 onward) or .NET web services/remoting (mainly used in earlier versions).

Figure 6-4 shows a simple remote proxy structure.

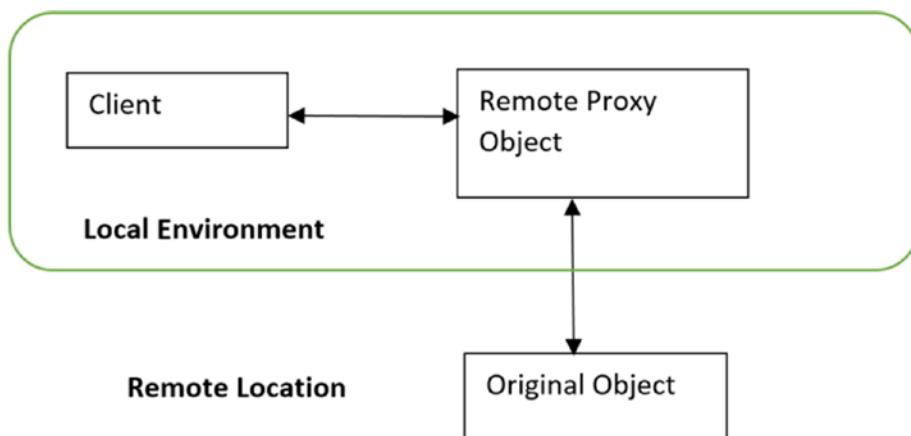


Figure 6-4. A simple remote proxy diagram

5. When can you use a virtual proxy?

Answer:

You can use virtual proxies to avoid loading an extremely large image multiple times.

6. When can you use a protection proxy?

Answer:

In an organization, the security team can implement a protection proxy to block Internet access to specific web sites.

Consider the following example, which is basically a modified version of the Proxy pattern implementation described earlier. For simplicity, let's assume you have only three registered users who can exercise the proxy method `DoSomeWork()`. If any other user (say Robin) tries to invoke the method, the system will reject those attempts. When the system rejects this kind of unwanted access, there is no point in making a proxy object. So, if you avoid instantiating an object of `ConcreteSubject` in the proxy class constructor, you can easily avoid creating objects unnecessarily.

Now let's go through the modified implementation.

Modified Implementation

Here's the modified implementation:

```
using System;
using System.Linq;//For Contains() method below

namespace ProxyPatternQAs
{
    /// <summary>
    /// Abstract class Subject
    /// </summary>
    public abstract class Subject
    {
```

```
public abstract void DoSomeWork();  
}  
/// <summary>  
/// ConcreteSubject class  
/// </summary>  
public class ConcreteSubject : Subject  
{  
    public override void DoSomeWork()  
    {  
        Console.WriteLine("ConcreteSubject.DoSomeWork()");  
    }  
}  
/// <summary>  
/// Proxy class  
/// </summary>  
public class Proxy : Subject  
{  
    Subject cs;  
    string[] registeredUsers;  
    string currentUser;  
    public Proxy(string currentUser)  
    {  
        //Avoiding to instantiate inside the constructor  
        //cs = new ConcreteSubject();  
        //Registered users  
        registeredUsers =new string[]{"Admin","Rohit","Sam"};  
        this.currentUser = currentUser;  
    }  
    public override void DoSomeWork()  
    {  
        Console.WriteLine("\nProxy call happening now...");  
        Console.WriteLine("{0} wants to invoke a proxy  
method.",currentUser);  
        if (registeredUsers.Contains(currentUser))  
        {
```

```
//Lazy initialization: We'll not instantiate until the
method is called
if (cs == null)
{
    cs = new ConcreteSubject();
}
cs.DoSomeWork();

}

else
{
    Console.WriteLine("Sorry {0}, you do not have access.", currentUser);
}

}

}

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***Proxy Pattern Demo***\n");
        //Authorized user-Admin
        Proxy px1 = new Proxy("Admin");
        px1.DoSomeWork();
        //Unwanted User-Robin
        Proxy px2 = new Proxy("Robin");
        px2.DoSomeWork();
        Console.ReadKey();
    }
}
}
```

Modified Output

Here's the modified output:

```
***Proxy Pattern Demo***
```

```
Proxy call happening now...
Admin wants to invoke a proxy method.
ConcreteSubject.DoSomeWork()
```

```
Proxy call happening now...
Robin wants to invoke a proxy method.
Sorry Robin, you do not have access.
```

7. It looks like proxies act like decorators. Is this understanding correct?

Answer:

A protection proxy might be implemented like a decorator, but you should not forget the intent of a proxy. Decorators focus on adding responsibilities, but proxies focus on controlling the access to an object. Proxies differ from each other through their types and implementations. So, if you can remember their purposes, in most cases you will be able to clearly distinguish proxies from decorators.

CHAPTER 7

Decorator Pattern

This chapter covers the Decorator pattern.

GoF Definition

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Concept

This pattern promotes the concept that your class should be closed for modification but open for extension. In other words, you can add a functionality without disturbing the existing functionalities. The concept is useful when you want to add some special functionality to a specific object instead of the whole class. This pattern prefers object composition over inheritance. Once you master this technique, you can add new responsibilities to an object without affecting the underlying classes.

Real-Life Example

Suppose you own a single-story house and you decide to build a second floor on top of it. Obviously, you may not want to change the architecture of the ground floor. But you may want to change the design of the architecture for the newly added floor without affecting the existing architecture.

Figure 7-1, Figure 7-2, and Figure 7-3 illustrate this concept.



Figure 7-1. Original house

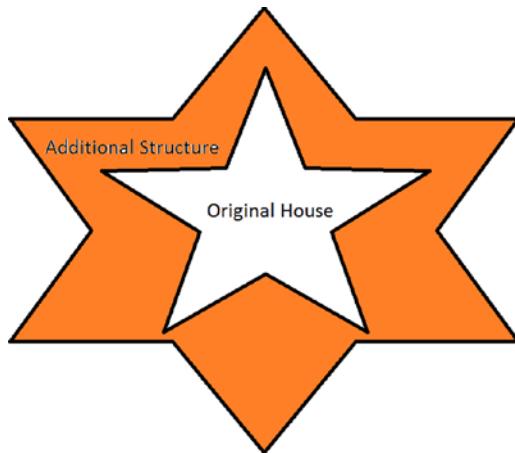


Figure 7-2. Original house with a decorator (the additional structure is built on top of original structure)

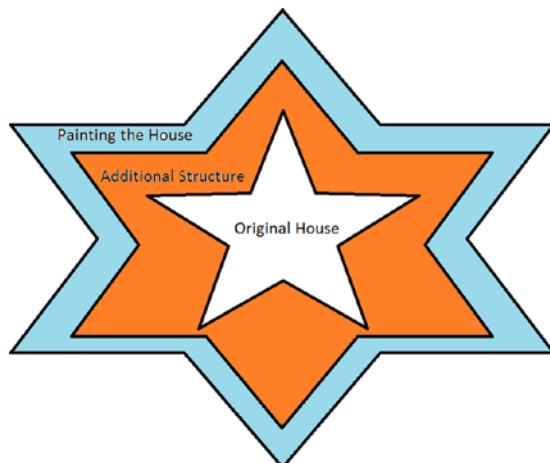


Figure 7-3. Creating an additional decorator from an existing decorator and modifying the house (now painting the house)

Note The case shown in Figure 7-3 is optional. You can use an existing decorator object to enhance the behavior in this way, or you can create a new decorator object and put all the new behavior in it.

Computer World Example

Suppose in a GUI-based toolkit you want to add some border properties. You could do this with inheritance, but that cannot be treated as an ultimate solution because you do not have absolute control over everything from the beginning. So, this technique is static in nature.

Decorators offer a flexible approach. They promote the concept of dynamic choices. For example, you can surround the component in another object. The enclosing object is termed a *decorator*, and it must conform to the interface of the component that it decorates. It will forward the requests to the component, and it can perform additional operations before or after those requests. In fact, you can add an unlimited number of responsibilities with this concept.

Note The I/O streams implementations in both .NET Framework and Java use the concept of Decorator pattern.

Illustration

In this example, I have not modified the core `MakeHouse()` method. I have created two additional decorators, `ConcreteDecoratorEx1` and `ConcreteDecoratorEx2`, to serve my needs, but I have kept the original structure intact.

Class Diagram

Figure 7-4 shows the class diagram.

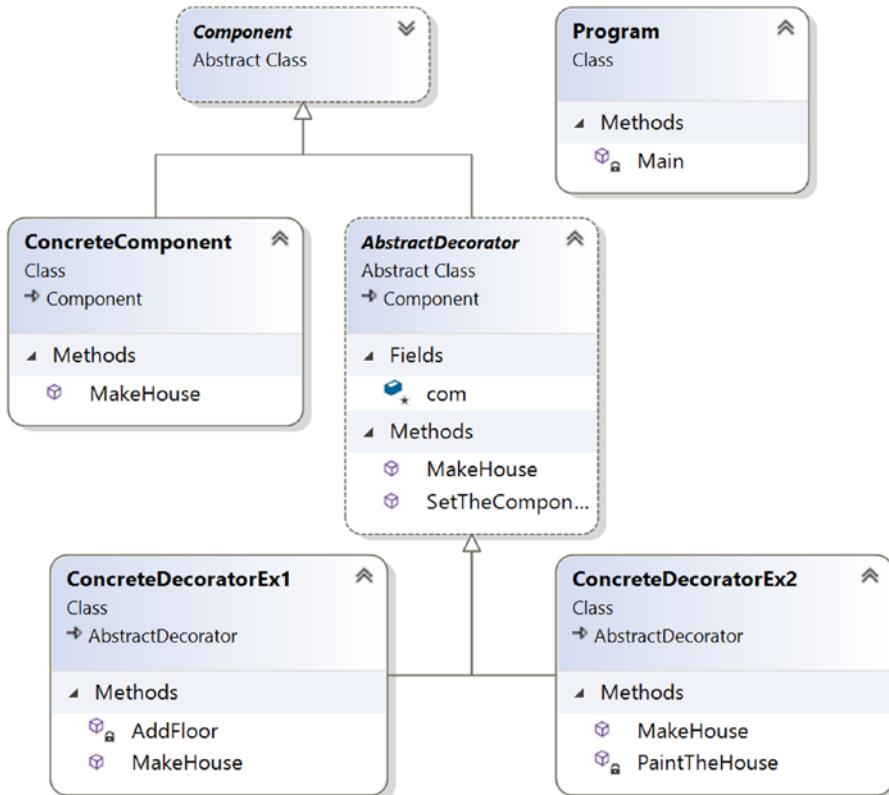


Figure 7-4. Class diagram

Solution Explorer View

Figure 7-5 shows the high-level structure of the parts of the program.

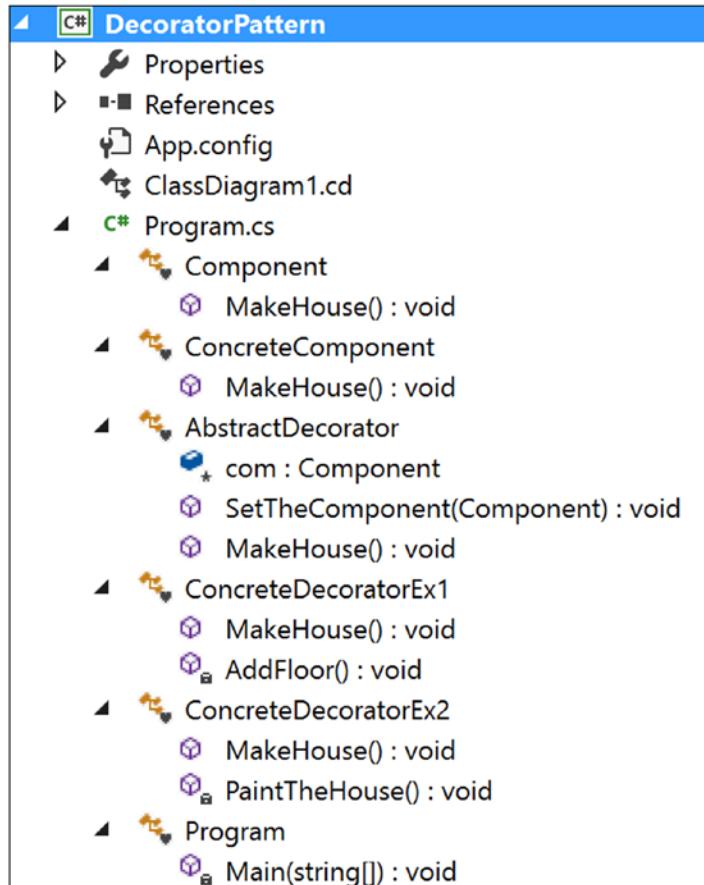


Figure 7-5. Solution Explorer View

Implementation

Here's the implementation:

```
using System;

namespace DecoratorPattern
{
    abstract class Component
    {
        public abstract void MakeHouse();
    }

    class ConcreteComponent : Component
    {
        public override void MakeHouse()
        {
            Console.WriteLine("Original House is complete. It is closed for
modification.");
        }
    }

    abstract class AbstractDecorator : Component
    {
        protected Component com ;
        public void SetTheComponent(Component c)
        {
            com = c;
        }
        public override void MakeHouse()
        {
            if (com != null)
            {
                com.MakeHouse();//Delegating the task
            }
        }
    }
}
```

```
class ConcreteDecoratorEx1 : AbstractDecorator
{
    public override void MakeHouse()
    {
        base.MakeHouse();
        Console.WriteLine("/**Using a decorator**");
        //Decorating now.
        AddFloor();
        //You can put additional stuff as per your needs.
    }
    private void AddFloor()
    {
        Console.WriteLine("I am making an additional floor on top of
it.");
    }
}
class ConcreteDecoratorEx2 : AbstractDecorator
{
    public override void MakeHouse()
    {
        Console.WriteLine("");
        base.MakeHouse();
        Console.WriteLine("/**Using another decorator**");
        //Decorating now.
        PaintTheHouse();
        //You can add additional stuffs as per your need
    }
    private void PaintTheHouse()
    {
        Console.WriteLine("Now I am painting the house.");
    }
}
```

```

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***Decorator pattern Demo***\n");
        ConcreteComponent cc = new ConcreteComponent();

        ConcreteDecoratorEx1 decorator1 = new ConcreteDecoratorEx1();
        decorator1.SetTheComponent(cc);
        decorator1.MakeHouse();

        ConcreteDecoratorEx2 decorator2 = new ConcreteDecoratorEx2();
        //Adding results from decorator1
        decorator2.SetTheComponent(decorator1);
        decorator2.MakeHouse();

        Console.ReadKey();
    }
}
}

```

Output

Here's the output. Notice the lines in bold.

*****Decorator pattern Demo*****

Original House is complete. It is closed for modification.

*****Using a decorator*****

I am making an additional floor on top of it.

Original House is complete. It is closed for modification.

*****Using a decorator*****

I am making an additional floor on top of it.

*****Using another decorator*****

Now I am painting the house.

Q&A Session

1. Can you explain how composition is promoting dynamic behavior that inheritance cannot?

Answer:

When a derived class inherits from a base class, it actually inherits the behavior of the base class at that time only. Though different subclasses can extend the base/parent class in different ways, this type of binding is known at compile time. So, the method is static. But by using the concept of composition, as in the previous example, you get dynamic behavior.

When you design a base/parent class, you may not have enough visibility about what kind of additional responsibilities your clients may want in some later phase. Since the constraint is that you cannot modify the existing code, in this case object composition not only outclasses inheritance, but it also ensures that you are not introducing bugs in the old architecture.

Lastly, in this context, you must try to remember one key design principle that says *classes should be open for extension but closed for modification*.

2. What are the key advantages of using a decorator?

Answer:

- The existing structure is untouched, so you cannot introduce bugs there.
- New functionalities can be added to an existing object easily.
- You do not need to predict/implement all the supported functionalities at once (in other words, in the initial design phase). You can develop incrementally. For example, you can add decorator objects one by one to support your needs. You must acknowledge that if you make a complex class first and then want to extend the functionalities, that will be a tedious process.

3. How is the overall design pattern different from inheritance?

Answer:

You can add or remove responsibilities by simply attaching or detaching decorators. But with simple inheritance techniques, you need to create new classes for new responsibilities. So, it is possible that you will end up with a complex system.

Consider the example again. Suppose you want to add a new floor, paint the house, and do some extra work. To fulfill this need, you can start with `decorator2` because it is already providing the support to add a floor and paint. Then you need to add a simple wrapper to complete those additional responsibilities.

But if you start with inheritance from the beginning, then you may have multiple subclasses, in other words, one for adding a floor and one for painting the house, as shown in Figure 7-6 (a hierarchical inheritance).

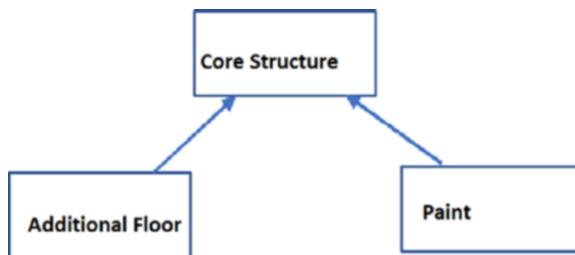


Figure 7-6. A hierarchical inheritance

So, if you need an additional painted floor with some extra features, you may need to end up with a design like in Figure 7-7.

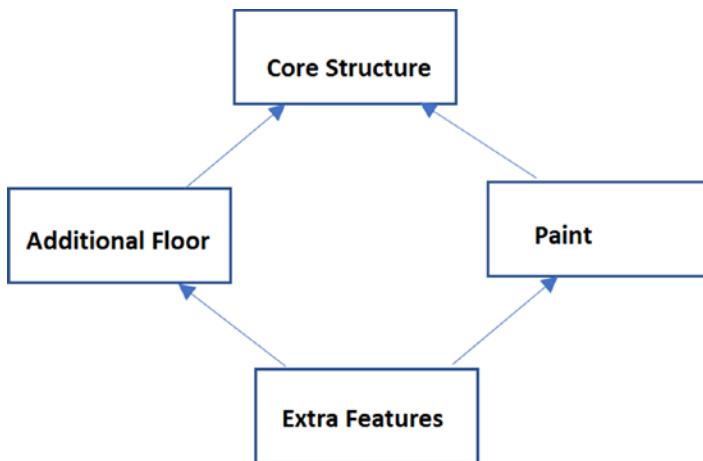


Figure 7-7. A class (*Extra Features*) needs to inherit from multiple base classes

Now you will feel the heat of the “diamond effect” because in many programming languages, including C#, multiple base classes are not allowed.

You will also discover that the inheritance mechanism is not only much more challenging and time-consuming compared to the Decorator pattern, but it may promote duplicate code in your application. Lastly, do not forget that inheritance promotes only compile-time binding (not dynamic binding).

4. **Why are you creating a class with a single responsibility? You could make a subclass that can simply add a floor and then paint. In that case, you may end up with fewer subclasses. Is this understanding correct?**

Answer:

If you are familiar with SOLID principles, you will know that there is a principle called *single responsibility*. The idea behind this principle is that each class should have a responsibility over a single part of the functionality provided in the software. The Decorator pattern is effective when you use the single responsibility principle because you can simply add or remove responsibilities dynamically.

5. What are the disadvantages associated with this pattern?

Answer:

I believe that if you are careful, there are no significant disadvantages. But if you create too many decorators in the system, it will be hard to maintain and debug. So, in that case, they can create unnecessary confusion.

6. In the example, the AbstractDecorator class is abstract, but there is no abstract method in it. How is this possible?

Answer:

In C#, a class can be abstract without containing any abstract method, but the reverse is not true. In other words, if a class contains at least one abstract method, it means that the class itself is incomplete and you are forced to mark it with the `abstract` keyword.

Also, if you read the comment in Figure 7-8, you'll understand that you are delegating the task to a concrete decorator in this case because you want to use and instantiate the concrete decorators only.

```
abstract class AbstractDecorator : Component
{
    protected Component com ;
    public void SetTheComponent(Component c)
    {
        com = c;
    }
    public override void MakeHouse()
    {
        if (com != null)
        {
            com.MakeHouse(); //Delegating the task
        }
    }
}
```



Figure 7-8. An abstract class-AbstractDecorator

So, in this example, you cannot simply instantiate an `AbstractDecorator` instance because it is marked with the `abstract` keyword.

The following line will create a compilation error (Figure 7-9):

```
AbstractDecorator abstractDecorator = new AbstractDecorator();
```



CS0144 Cannot create an instance of the abstract class or interface 'AbstractDecorator'

Figure 7-9. Error message

7. In this example, instead of using concrete decorators, you could use the concept of polymorphism to generate the same output.

```
AbstractDecorator decorator1 = new ConcreteDecoratorEx1();
decorator1.SetTheComponent(cc);
decorator1.MakeHouse();

//ConcreteDecoratorEx2 decorator2 = new ConcreteDecoratorEx2();
AbstractDecorator decorator2 = new ConcreteDecoratorEx2();
//Adding results from decorator1
decorator2.SetTheComponent(decorator1);
decorator2.MakeHouse();
```

Is this understanding correct?

Answer:

Yes.

8. Then using polymorphism, can you reduce the lines of code in this way?

```
AbstractDecorator decorator1 = new ConcreteDecoratorEx1();
decorator1.SetTheComponent(cc);
decorator1.MakeHouse();
//Adding results from decorator1
decorator1.SetTheComponent(decorator1);
decorator2.MakeHouse();
```

Answer:

No. First, is this easily readable? Second, you will receive a `StackOverflowException` at runtime.

9. Is it mandatory to use decorators for dynamic binding only?

Answer:

No. You can use the concept for both static and dynamic binding. But dynamic binding is its strength, so I have concentrated on that here. The GoF definition also focuses on dynamic binding only.

CHAPTER 8

Adapter Pattern

This chapter covers the Adapter pattern.

GoF Definition

Convert the interface of a class into another interface that clients expect. The Adapter pattern lets classes work together that could not otherwise because of incompatible interfaces.

Concept

The core concept is best described by the following examples.

Real-Life Example

A common use of this pattern is when you use an electrical outlet adapter/AC power adapter in international travels. These adapters can act as middlemen so that an electronic device, say a laptop that accepts a U.S. power supply, can be plugged into a European power outlet. Consider another example. Suppose you need to charge your mobile phone. But you see that the electrical outlet is not compatible with your charger. In this case, you may need to use an adapter. Even a translator who is converting one language to another can be considered to be following this pattern in real life.

Now imagine a situation where you need to plug an application into an adapter (which is X-shaped in this example) to use the intended interface. Without using this adapter, you cannot join the application and the interface.

Figure 8-1 shows the case before using an adapter.

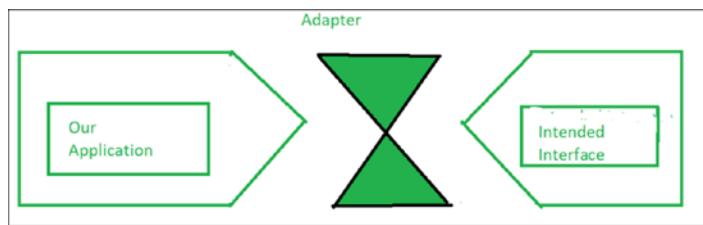


Figure 8-1. Before using an adapter

Figure 8-2 shows the case after using an adapter.



Figure 8-2. After using an adapter

Computer World Example

Suppose you have an application that can be broadly classified into two parts: the user interface (UI or front end) and the database (back end). Through the user interface, clients can pass some specific type of data or objects. Your database is compatible with those objects and can store them smoothly. Over a period of time, you may feel that you need to upgrade your software to make your clients happy. So, you may want to allow some other type of object also to pass through the UI. But in this case, the first resistance will come from your database because it cannot store these new types of objects. In such a situation, you can use an adapter that will take care of the conversion of these new objects to a compatible form that your old database can accept.

A simple use of this pattern is described in the following example.

Illustration

In this example, you can calculate the area of a rectangle easily. In the Calculator class, you need to supply a rectangle in the GetArea() method to get the area of the rectangle. Now suppose you want to calculate the area of a triangle. But your constraint is that you want to get the area of it through the GetArea() method of the Calculator class.

How can you achieve that?

To deal with this problem, you make a CalculatorAdapter for a triangle and pass a triangle in its GetArea() method. In turn, the method will treat these triangles like rectangles to get the areas of them from the GetArea() method of the Calculator class.

Class Diagram

Figure 8-3 shows the class diagram.

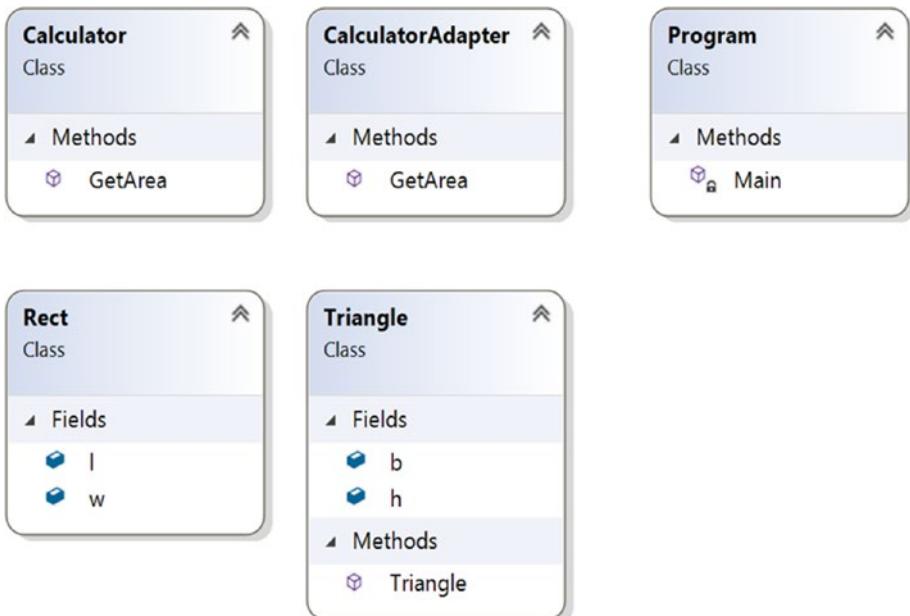


Figure 8-3. Class diagram

Directed Graph Document

Figure 8-4 shows the directed graph document.

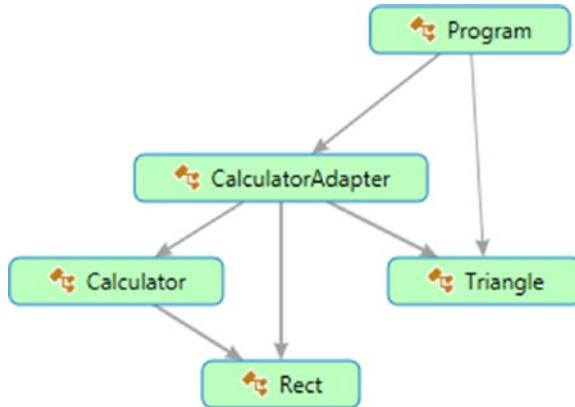


Figure 8-4. Directed Graph Document

Solution Explorer View

Figure 8-5 shows the high-level structure of the parts of the program.

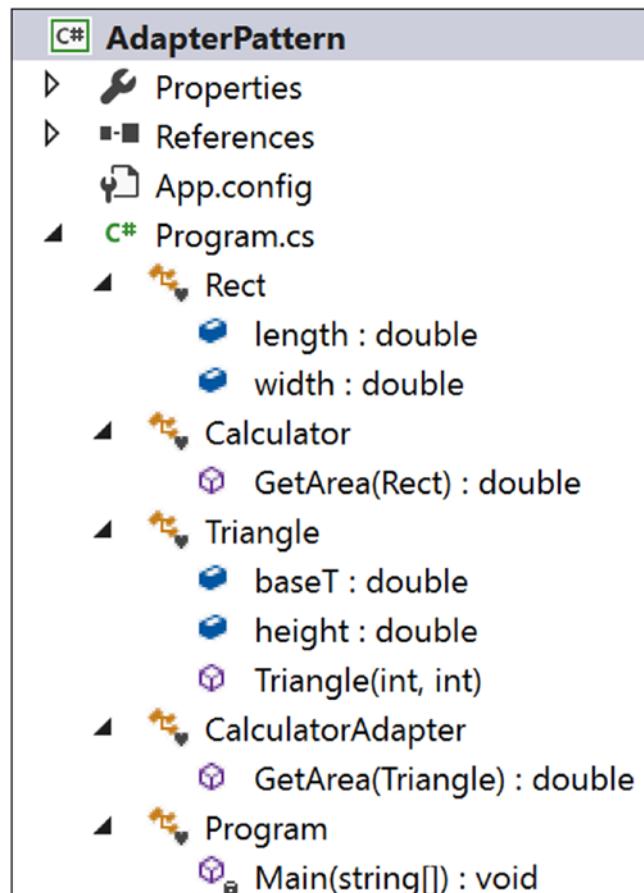


Figure 8-5. Solution Explorer View

Implementation

Here's the implementation:

```
using System;

namespace AdapterPattern
{
    class Rect
    {
        public double length;
        public double width;
    }

    class Calculator
    {
        public double GetArea(Rect rect)
        {
            return rect.length * rect.width;
        }
    }

    class Triangle
    {
        public double baseT;//base
        public double height;//height
        public Triangle(int b, int h)
        {
            this.baseT = b;
            this.height = h;
        }
    }

    class CalculatorAdapter
    {
        public double GetArea(Triangle triangle)
        {
            Calculator c = new Calculator();
            Rect rect = new Rect();
```

```

        //Area of Triangle=0.5*base*height
        rect.length = triangle.baseT;
        rect.width = 0.5 * triangle.height;
        return c.GetArea(rect);
    }
}

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***Adapter Pattern Demo***\n");
        CalculatorAdapter cal = new CalculatorAdapter();
        Triangle t = new Triangle(20, 10);
        Console.WriteLine("Area of Triangle is" + cal.GetArea(t) +
        "Square unit");
        Console.ReadKey();
    }
}
}

```

Output

Here's the output:

```

***Adapter Pattern Demo***

Area of Triangle is 100 Square unit

```

Modified Illustration

You have already seen a simple example of the Adapter design pattern. But if you want to follow object-oriented design principles, you may feel that you need to modify the implementation because you have learned that instead of using concrete classes, you should always prefer to use interfaces. So, keeping this aim in mind, let's modify the illustration.

Key Characteristics of the Modified Implementation

Here are some key characteristics of the modified example:

- Class Rect is implementing RectInterface and the CalculateAreaOfRectangle() method calculates the area of a rectangle object.
- The Triangle class implements TriInterface and the CalculateAreaOfTriangle() method calculates the area of a triangle object.
- The constraint is that you need to calculate the area of the triangle using RectInterface. To serve this purpose, you make an adapter that can interact with RectInterface.
- Notice the beauty of using this pattern. Neither the Rectangle code nor the Triangle code needs to change. You are using an adapter that helps you to interact with RectInterface, and at high level, it appears to you that by using a RectInterface method, you are computing the area of a triangle.
- Notice that the GetArea(RectInterface r) method also does not know that through TriangleAdapter it is getting a Triangle object instead of a Rectangle object.
- Suppose you have a higher demand for how many rectangle objects you have. With the Adapter pattern, you can use some of the triangle objects that may behave like rectangle objects. How? Well, when using the adapter, though you are calling CalculateAreaOfRectangle(), it is actually invoking CalculateAreaOfTriangle(). So, you can modify the method body as per your needs. For example, you could multiply the triangle area by 2.0 to get an area of 200 square units in this example (just like a rectangle object with a length 20 of units and a width of 10 units). So, this technique can help you when you need to deal with objects that are not exactly the same but are similar.

Note In the context of the previous point, you should not make an attempt to convert a circle to a rectangle (or do a similar type of conversion) to get an area because they are totally different shapes. But in this example, I am talking about triangles and rectangles because they have similarities, and areas can be computed easily with minor changes.

Modified Solution Explorer View

Figure 8-6 shows the structure of the modified program.

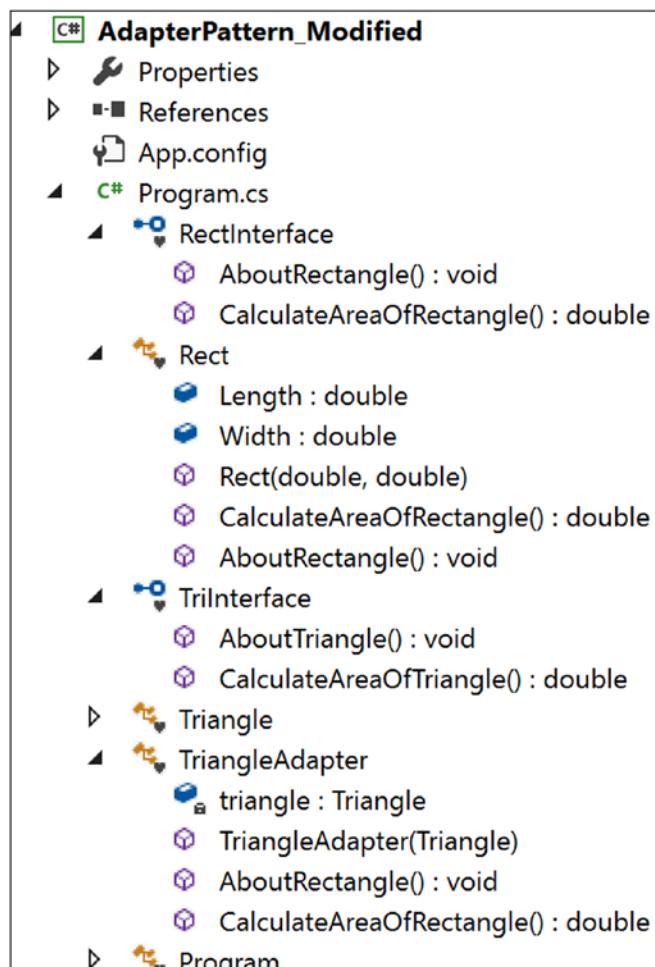


Figure 8-6. Solution Explorer View

Modified Implementation

Here's the modified implementation:

```
using System;
namespace AdapterPattern_Modified
{
    interface RectInterface
    {
        void AboutRectangle();
        double CalculateAreaOfRectangle();
    }
    class Rect : RectInterface
    {
        public double Length;
        public double Width;
        public Rect(double l, double w)
        {
            this.Length = l;
            this.Width = w;
        }

        public double CalculateAreaOfRectangle()
        {
            return Length * Width;
        }

        public void AboutRectangle()
        {
            Console.WriteLine("Actually, I am a Rectangle");
        }
    }

    interface TriInterface
    {
        void AboutTriangle();
        double CalculateAreaOfTriangle();
    }
}
```

```
class Triangle : TriInterface
{
    public double BaseLength;//base
    public double Height;//height
    public Triangle(double b, double h)
    {
        this.BaseLength = b;
        this.Height = h;
    }

    public double CalculateAreaOfTriangle()
    {
        return 0.5 * BaseLength * Height;
    }

    public void AboutTriangle()
    {
        Console.WriteLine("Actually, I am a Triangle");
    }
}

/*TriangleAdapter is implementing RectInterface.
So, it needs to implement all the methods defined
in the target interface.*/
class TriangleAdapter:RectInterface
{
    Triangle triangle;
    public TriangleAdapter(Triangle t)
    {
        this.triangle = t;
    }

    public void AboutRectangle()
    {
        triangle.AboutTriangle();
    }

    public double CalculateAreaOfRectangle()
    {
```

```
        return triangle.CalculateAreaOfTriangle();
    }
}

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***Adapter Pattern Modified Demo***\n");
        //CalculatorAdapter cal = new CalculatorAdapter();
        Rect r = new Rect(20, 10);
        Console.WriteLine("Area of Rectangle is :{0} Square unit",
            r.CalculateAreaOfRectangle());
        Triangle t = new Triangle(20, 10);
        Console.WriteLine("Area of Triangle is :{0} Square unit",
            t.CalculateAreaOfTriangle());
        RectInterface adapter = new TriangleAdapter(t);
        //Passing a Triangle instead of a Rectangle
        Console.WriteLine("Area of Triangle using the triangle adapter
            is :{0} Square unit", GetArea(adapter));
        Console.ReadKey();
    }

    /*GetArea(RectInterface r) method does not know that through
     *TriangleAdapter, it is getting a Triangle instead of a Rectangle*/
    static double GetArea(RectInterface r)
    {
        r.AboutRectangle();
        return r.CalculateAreaOfRectangle();
    }
}
```

Modified Output

Here's the output:

```
***Adapter Pattern Modified Demo***
```

Area of Rectangle is :200 Square unit

Area of Triangle is :100 Square unit

Actually, I am a Triangle

Area of Triangle using the triangle adapter is :100 Square unit

Types of Adapters

The GoF described two types of adapters: class adapters and object adapters.

Object Adapters

Object adapters adapt through object compositions, as shown in Figure 8-7. So, the adapter discussed so far is an example of an object adapter.

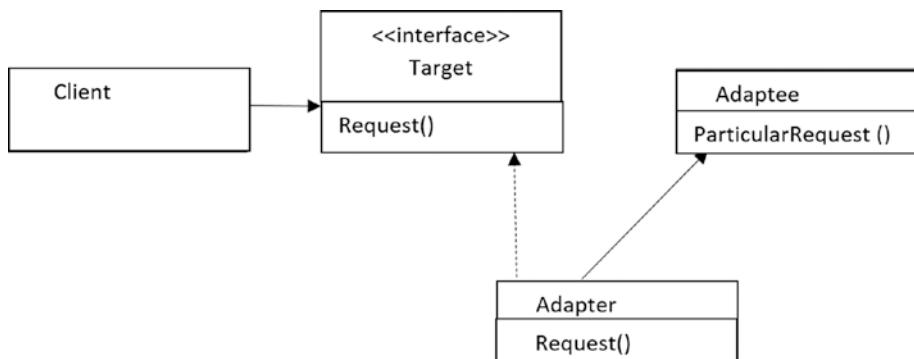


Figure 8-7. Object adapter

In this example, `TriangleAdapter` is the adapter that implements `RectInterface` (`Target interface`). `Triangle` is the `Adaptee` interface. The adapter holds the `Adaptee` instance.

Class Adapters

Class adapters adapt through subclassing and support multiple inheritance. But you know that in C# multiple inheritance through classes is not supported. (You need interfaces to implement the concept of multiple inheritance.)

Figure 8-8 shows the typical class diagram for class adapters, which support multiple inheritance.

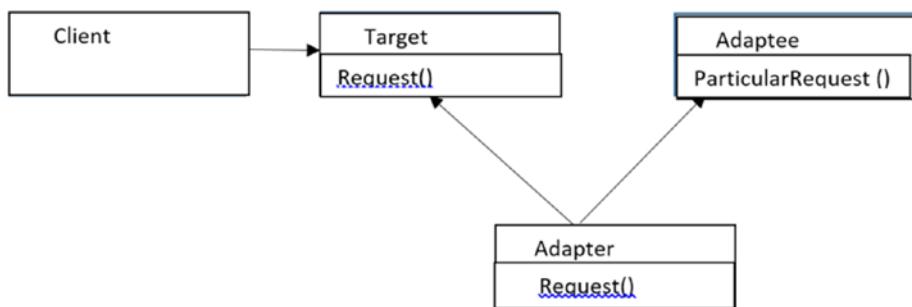


Figure 8-8. Class adapter

Q&A Session

1. How can you implement a class adapter design pattern in C#?

Answer:

You can subclass an existing class and implement the desired interface. Here's an example:

```

class ClassAdapter : Triangle, RectInterface
{
    public ClassAdapter(double b, double h) : base(b, h)
    {
    }

    public void AboutRectangle()
    {
        Console.WriteLine("Actually, I am an Adapter");
    }
}
    
```

```
public double CalculateAreaOfRectangle()
{
    return 2.0 * base.CalculateAreaOfTriangle();
}
```

Note that this approach may not be applicable in all scenarios. For example, you may need to adapt a method that is not specified in a C# interface. In those cases, object adapters are useful.

2. Which one do you prefer, class adapters or object adapters?

Answer:

In most cases, I prefer compositions over inheritance. Object adapters use compositions and are more flexible. Also, in many cases, you may not implement a true class adapter (refer to the answer of the previous question).

3. What are the drawbacks associated with this pattern?

Answer:

I do not see any big challenges. I believe that an adapter's job is simple and straightforward, but you may need to write some additional code. However, the payoff is great, particularly for those legacy systems that cannot be changed but you still want to use them for their stability.

CHAPTER 9

Facade Pattern

This chapter covers the Facade pattern.

GoF Definition

Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

Concept

This pattern supports loose coupling. With this pattern, you can emphasize the abstraction and hide the complex details by exposing a simple interface.

Real-Life Example

Suppose you are going to host a birthday party with 300 guests. Nowadays you can hire a party organizer and let them know the key information such as the party type, date and time of the party, number of attendees, and so on. The organizer will do the rest for you. You do not need to think about how they will decorate the party room, whether the food will be buffet style, and so on.

Computer World Example

Think about a case when you use a method from a library (in the context of a programming language). You don't care how the method is implemented in the library. You just call the method for its easy usage.

Illustration

In this example, your clients can construct or destroy a particular kind of robot by invoking simple methods such as `ConstructMilanoRobot` and `DestroyMilanoRobot`. From the client's point of view, the client needs to interact only with the facade (see the `Program.cs` file). `RobotFacade` takes full responsibility for creating or destroying a particular kind of robot. This facade talks to each of these subsystems (`RobotHands`, `RobotBody`, `RobotColor`) to fulfil the client's request. Clients do not have to worry about the creation or destruction process or the calling sequence of these methods.

Class Diagram

Figure 9-1 shows the class diagram.

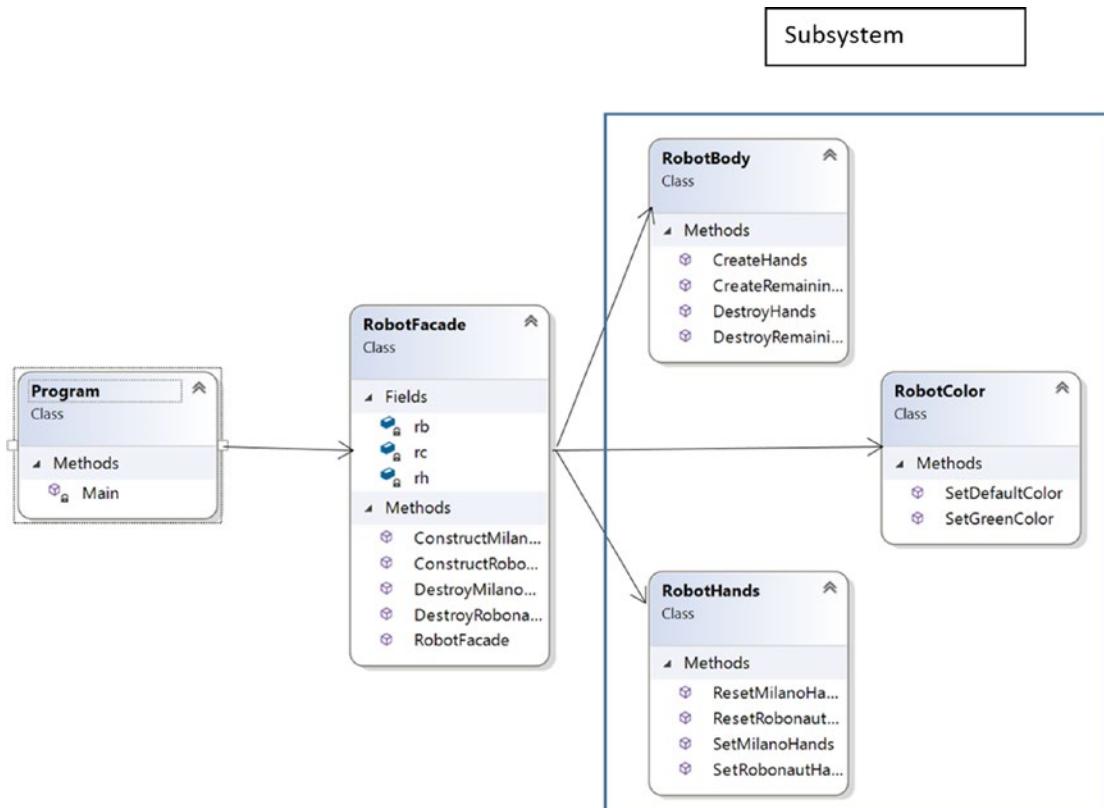


Figure 9-1. Class diagram

Directed Graph Document

Figure 9-2 shows the directed graph document.

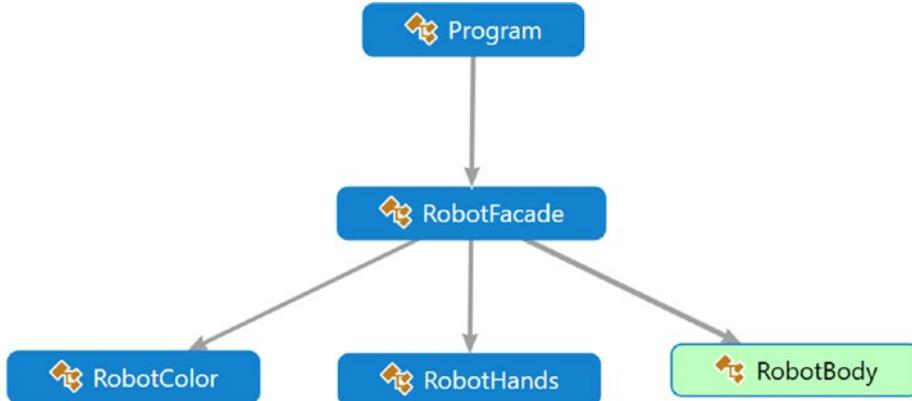


Figure 9-2. Directed Graph Document

Solution Explorer View

Figure 9-3 shows the high-level structure of the parts of the program.

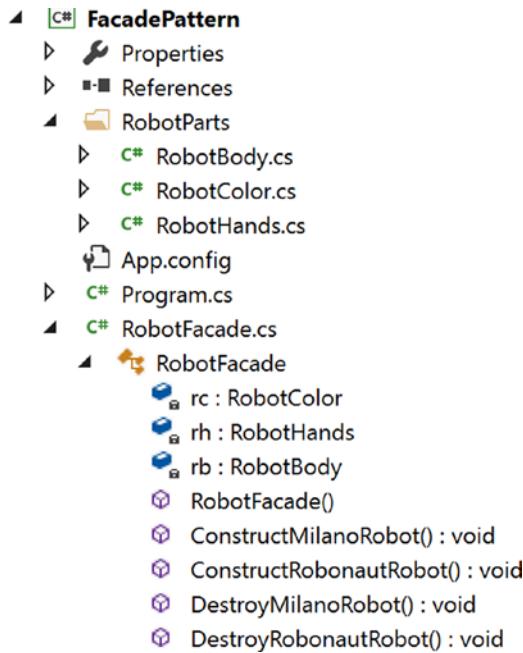


Figure 9-3. Solution Explorer View

Implementation

Here's the implementation:

```
// RobotBody.cs

using System;

namespace FacadePattern.RobotParts
{
    public class RobotBody
    {
        public void CreateHands()
        {
            Console.WriteLine("Hands manufactured");
        }

        public void CreateRemainingParts()
        {
            Console.WriteLine("Remaining parts (other than hands) are
created");
        }

        public void DestroyHands()
        {
            Console.WriteLine("The robot's hands are destroyed");
        }

        public void DestroyRemainingParts()
        {
            Console.WriteLine("The robot's remaining parts are destroyed");
        }
    }
}

//RobotColor.cs

using System;

namespace FacadePattern.RobotParts
{
```

```
public class RobotColor
{
    public void SetDefaultColor()
    {
        Console.WriteLine("This is steel color robot.");
    }
    public void SetGreenColor()
    {
        Console.WriteLine("This is a green color robot.");
    }
}

// RobotHands.cs

using System;

namespace FacadePattern.RobotParts
{
    public class RobotHands
    {
        public void SetMilanoHands()
        {
            Console.WriteLine("The robot will have EH1 Milano hands");
        }
        public void SetRobonautHands()
        {
            Console.WriteLine("The robot will have Robonaut hands");
        }
        public void ResetMilanoHands()
        {
            Console.WriteLine("EH1 Milano hands are about to be
destroyed");
        }
        public void ResetRobonautHands()
        {
            Console.WriteLine("Robonaut hands are about to be destroyed");
        }
    }
}
```

```
        }
    }
}

// RobotFacade.cs
using System;
using FacadePattern.RobotParts;

namespace FacadePattern
{
    public class RobotFacade
    {
        RobotColor rc;
        RobotHands rh ;
        RobotBody rb;
        public RobotFacade()
        {
            rc = new RobotColor();
            rh = new RobotHands();
            rb = new RobotBody();

        }
        public void ConstructMilanoRobot()
        {
            Console.WriteLine("Creation of a Milano Robot Start");
            rc.SetDefaultColor();
            rh.SetMilanoHands();
            rb.CreateHands();
            rb.CreateRemainingParts();
            Console.WriteLine("Milano Robot Creation End");
            Console.WriteLine();
        }
        public void ConstructRobonautRobot()
        {
            Console.WriteLine("Initiating the creational process of a
Robonaut Robot");
            rc.SetGreenColor();
            rh.SetRobonautHands();
```

```
rb.CreateHands();
rb.CreateRemainingParts();
Console.WriteLine("A Robonaut Robot is created");
Console.WriteLine();
}

public void DestroyMilanoRobot()
{
    Console.WriteLine("Milano Robot's destruction process is
started");
    rh.ResetMilanoHands();
    rb.DestroyHands();
    rb.DestroyRemainingParts();
    Console.WriteLine("Milano Robot's destruction process is over");
    Console.WriteLine();
}
public void DestroyRobonautRobot()
{
    Console.WriteLine("Initiating a Robonaut Robot's destruction
process.");
    rh.ResetRobonautHands();
    rb.DestroyHands();
    rb.DestroyRemainingParts();
    Console.WriteLine("A Robonaut Robot is destroyed");
    Console.WriteLine();
}
}

// Program.cs
using System;

namespace FacadePattern
{
    class Program
    {
        static void Main(string[] args)
```

```

{
    Console.WriteLine("/**Facade Pattern Demo**\n");
    //Creating Robots
    RobotFacade rf1 = new RobotFacade();
    rf1.ConstructMilanoRobot();
    RobotFacade rf2 = new RobotFacade();
    rf2.ConstructRobonautRobot();
    //Destroying robots
    rf1.DestroyMilanoRobot();
    rf2.DestroyRobonautRobot();
    Console.ReadLine();
}
}
}

```

Output

Here's the output:

Facade Pattern Demo

Creation of a Milano Robot Start

This is steel color robot.

The robot will have EH1 Milano hands

Hands manufactured

Remaining parts (other than hands) are created

Milano Robot Creation End

Initiating the creational process of a Robonaut Robot

This is a green color robot.

The robot will have Robonaut hands

Hands manufactured

Remaining parts (other than hands) are created

A Robonaut Robot is created

Milano Robot's destruction process is started

EH1 Milano hands are about to be destroyed

The robot's hands are destroyed

The robot's remaining parts are destroyed
 Milano Robot's destruction process is over

Initiating a Robonaut Robot's destruction process.
 Robonaut hands are about to be destroyed
 The robot's hands are destroyed
 The robot's remaining parts are destroyed
 A Robonaut Robot is destroyed

Q&A Session

1. What are the key advantages of using the Facade pattern?

Answer:

- If your system consists of many subsystems, managing those subsystems becomes tough, and clients will find it difficult to communicate separately with each of these subsystems. In this scenario, Facade patterns are handy. Instead of presenting complex subsystems, you present one simplified interface to clients. This approach also supports weak coupling by separating a client from the subsystems.
- It can also help reduce the number of objects that a client needs to deal with.

2. The facade class is using compositions in this example. Is this necessary?

Answer:

Yes. With this approach, you can access the intended methods in each subsystem.

3. Can you now access each of the subsystems directly?

Answer:

Obviously, you can. The Facade pattern does not restrict you from doing this.

4. How is Facade different from the Adapter design pattern?

Answer:

In the Adapter pattern, you are trying to alter an interface so that your clients do not see any difference between the interfaces. By contrast, the Facade pattern simplifies the interface. It presents the client with a simple interface to interact with (instead of a complex subsystem).

5. There should be only one facade for a complex subsystem. Is this understanding correct?

Answer:

Not at all. You can create any number of facades for a particular subsystem.

6. Can you add new stuffs/additional code with a facade?

Answer:

Yes, you can.

7. What are the challenges associated with the Facade pattern?

Answer:

- Subsystems are connected with the facade layer. So, you need to take care of an additional layer of coding (increasing your codebase).
- When the internal structure of a subsystem changes, you need to incorporate the changes in the facade layer also.
- Developers need to learn about this new layer, whereas some of them may already aware of how to use the subsystems/APIs efficiently.

8. It appears that facades do not restrict you from directly connecting with subsystems. Is this understanding correct?

Answer:

Yes. But in that case, the code looks dirty, and you may lose the benefits associated with the Facade pattern.

CHAPTER 10

Flyweight Pattern

This chapter covers the Flyweight pattern.

GoF Definition

Use sharing to support large numbers of fine-grained objects efficiently.

Concept

The GoF said the following about flyweights:

A flyweight is a shared object that can be used in multiple contexts simultaneously. The flyweight acts as an independent object in each context—it's indistinguishable from an instance of the object that's not shared. Flyweights cannot make assumptions about the context in which they operate. The key concept here is the distinction between intrinsic and extrinsic state. Intrinsic state is stored in the flyweight; it consists of information that's independent of the flyweight's context, thereby making it sharable. Extrinsic state depends on and varies with the flyweight's context and therefore can't be shared. Client objects are responsible for passing extrinsic state to the flyweight when it needs it.

So, you can conclude the following:

- A flyweight is an object. It tries to minimize memory usage by sharing data as much as possible with other similar objects. Shared objects may try to allow their usage at fine granularities with minimum costs.
- Two common terms are used in the previous extract: intrinsic and extrinsic. Intrinsic state is stored/shared in the flyweight object. On the other hand, client objects store the extrinsic state, and these objects are passed to a flyweight object when they invoke the operations.

Real-Life Example

Suppose you have a pen. You can use different ink refills to write with different colors. So, the pen without the refill can be considered the flyweight with intrinsic data, and the refills can be considered the extrinsic data in this example.

Computer World Example

This pattern helps you to save memory by reducing the number of object instances at runtime. Suppose in a computer game you have a large number of participants whose core structures are the same, but their appearances vary (for example, they may have different states, colors, weapons, and so on). Therefore, if you want to store all of these objects with all these variations/states, the memory requirement will be huge. So, instead of storing all these objects, you can design the application in such way that you will create one of these instances, and your client object will maintain all of these variations/states. If you can successfully implement the concept in the design phase, then you have followed the Flyweight pattern in the application.

Consider another example. Suppose a company needs to print business cards for its employees. In this case, what will be the starting point? The business can create a common template where the company logo, address, and so on, is already printed (intrinsic), and later the company will put the particular employee details (extrinsic) on the cards.

Another common use of this pattern is seen in the graphical representation of characters in a word processor.

Illustration

The following example uses two types of objects only, small and large robots. But you may need many of them. So, you want to use a Flyweight pattern in this case.

Since these objects are similar, you do not want to create a new type of small (or large) robot if you already have one of them. In other words, you will try to reuse the existing one to serve your needs. Go through the code comments for more clarity.

Class Diagram

Figure 10-1 shows the class diagram.

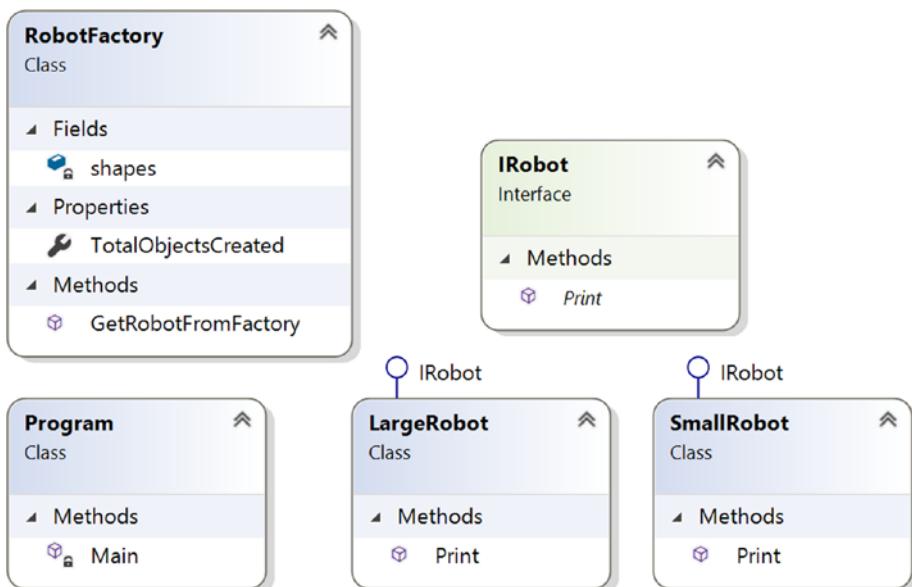


Figure 10-1. Class diagram

Directed Graph Document

Figure 10-2 shows the directed graph document.

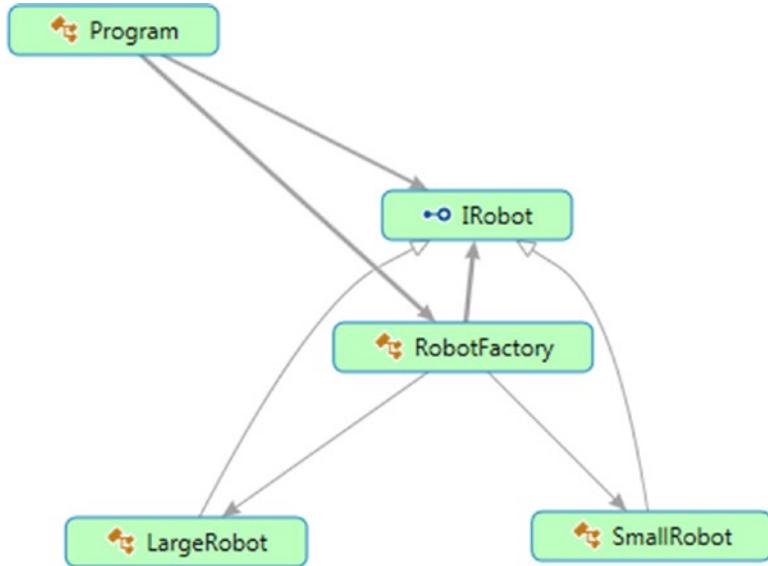


Figure 10-2. Directed Graph Document

Solution Explorer View

Figure 10-3 shows the high-level structure of the parts of the program.

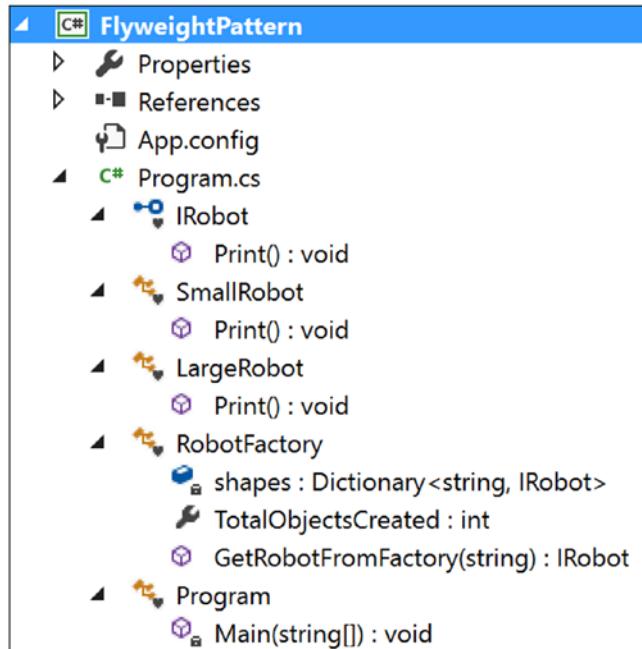


Figure 10-3. Solution Explorer View

Implementation

Here's the implementation:

```
using System;
using System.Collections.Generic;//Dictionary is used here

namespace FlyweightPattern
{
    /// <summary>
    /// The 'Flyweight' interface
    /// </summary>
    interface IRobot
    {
        void Print();
    }
}
```

CHAPTER 10 FLYWEIGHT PATTERN

```
/// <summary>
/// A 'ConcreteFlyweight' class
/// </summary>
class SmallRobot : IRobot
{
    public void Print()
    {
        Console.WriteLine("This is a small Robot");
    }
}

/// <summary>
/// A 'ConcreteFlyweight' class
/// </summary>
class LargeRobot : IRobot
{
    public void Print()
    {
        Console.WriteLine("I am a large Robot");
    }
}

/// <summary>
/// The 'FlyweightFactory' class
/// </summary>
class RobotFactory
{
    Dictionary<string, IRobot> shapes = new Dictionary<string,
IRobot>();

    public int TotalObjectsCreated
    {
        get {return shapes.Count;}
    }

    public IRobot GetRobotFromFactory(string robotType)
    {
        IRobot robotCategory = null;
```

```

        if (shapes.ContainsKey(robotType))
        {
            robotCategory = shapes[robotType];
        }
        else
        {
            switch (robotType)
            {
                case "Small":
                    robotCategory = new SmallRobot();
                    shapes.Add("Small", robotCategory);
                    break;
                case "Large":
                    robotCategory = new LargeRobot();
                    shapes.Add("Large", robotCategory);
                    break;
                default:
                    throw new Exception("Robot Factory can create only
                                         small and large robots");
            }
        }
        return robotCategory;
    }

}

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***Flyweight Pattern Demo***\n");
        RobotFactory myfactory = new RobotFactory();
        IRobot shape = myfactory.GetRobotFromFactory("Small");
        shape.Print();
        /*Now we are trying to get the 2 more Small robots.
        Note that: now onwards we need not create additional small
        robots because
    }
}

```

```

        we have already created one of this category*/
        for (int i = 0; i < 2; i++)
        {
            shape = myfactory.GetRobotFromFactory("Small");
            shape.Print();
        }
        int NumOfDistinctRobots = myfactory.TotalObjectsCreated;
        Console.WriteLine("\n Now, total numbers of distinct robot
objects is = {0}\n", NumOfDistinctRobots);

/*Here we are trying to get the 5 more Large robots.
Note that: now onwards we need not create additional small
robots because we have already created one of this category */
        for (int i = 0; i < 5; i++)
        {
            shape = myfactory.GetRobotFromFactory("Large");
            shape.Print();
        }

        NumOfDistinctRobots = myfactory.TotalObjectsCreated;
        Console.WriteLine("\n Distinct Robot objects created till
now = {0}", NumOfDistinctRobots);
        Console.ReadKey();
    }
}
}

```

Output

Here's the output:

Flyweight Pattern Demo

This is a small Robot
This is a small Robot
This is a small Robot

Now, total numbers of distinct robot objects is = 1

I am a large Robot
I am a large Robot

Distinct Robot objects created till now = 2

Improvements to the Program

Now take a close look at the previous program. It may seem that this pattern works like a Singleton pattern because you cannot create two distinct types of small (or large) robots. But there may be situations where you want to create different types of small (or large) robots with the same basic structures but different colors (you can add any special states or characteristics you like).

To keep this program simple, you are dealing with robots that can be either small type or large type (just like previously). In addition, each of these types can have either green or red color associated with them. So, before making any such robot, you will consult your factory. If you already have a small or large type of robot, you will not re-create them. You will collect the basic structure from your factory. Then you will color them only. Note that color is extrinsic data here, but the category of robot (Small or Large) is intrinsic.

Note To minimize the code, I have made some minor changes to the previous implementation. Instead of creating two separate classes (`SmallRobot` and `LargeRobot`), I have used a single class, `Robot`.

Modified Class Diagram

Figure 10-4 shows the modified class diagram.

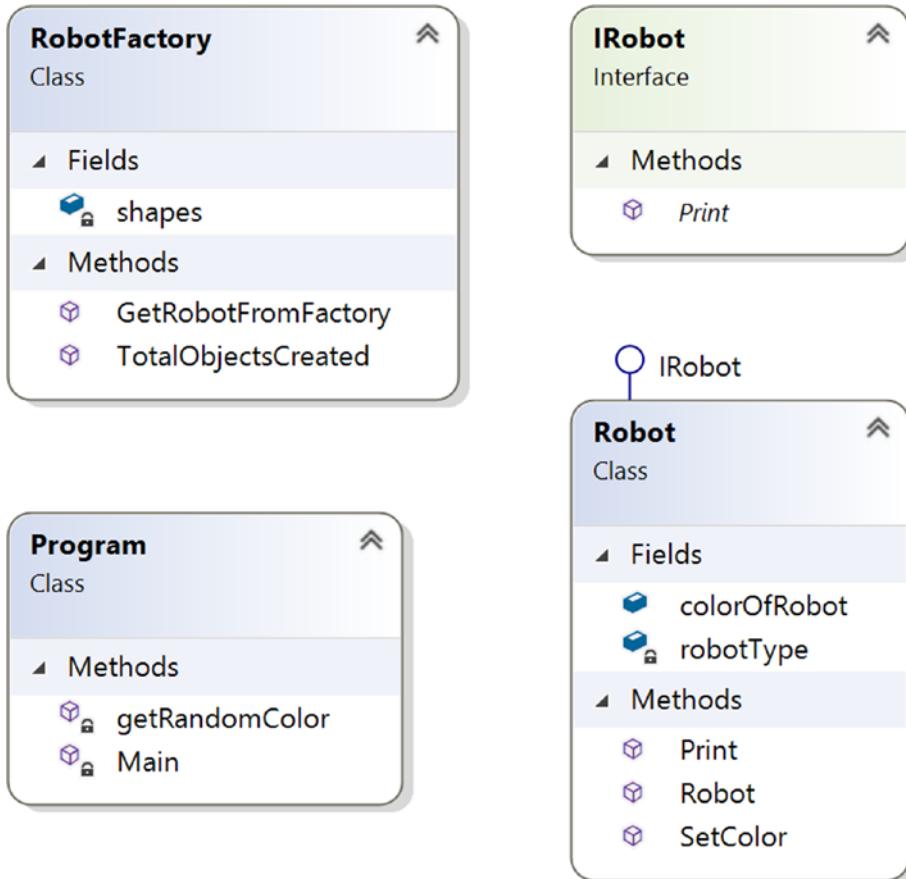


Figure 10-4. Modified class diagram

Modified Solution Explorer View

Figure 10-5 shows the modified Solution Explorer view.

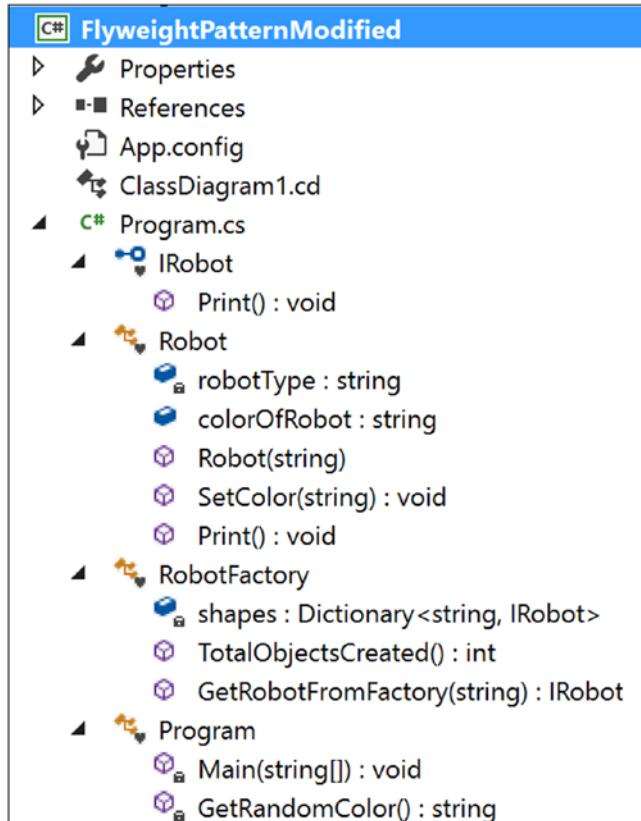


Figure 10-5. Modified Solution Explorer View

Modified Implementation

Here's the modified implementation:

```
using System;
using System.Collections.Generic;
using System.Threading;

namespace FlyweightPatternModified
{
    //Our interface
    interface IRobot
```

CHAPTER 10 FLYWEIGHT PATTERN

```
{  
    void Print();  
}  
  
/**  
 * The 'ConcreteFlyweight' class-SmallRobot  
 */  
class Robot : IRobot  
{  
    String robotType;  
    public String colorOfRobot;  
    public Robot(String robotType)  
    {  
        this.robotType = robotType;  
    }  
    public void SetColor(String colorOfRobot)  
    {  
        this.colorOfRobot = colorOfRobot;  
    }  
    public void Print()  
    {  
        Console.WriteLine("This is a" + robotType + "type robot with"  
            + colorOfRobot + "color");  
    }  
}  
  
// The 'FlyweightFactory' class  
  
class RobotFactory  
{  
    Dictionary<string, IRobot> shapes = new Dictionary<string,  
IRobot>();  
    public int TotalObjectsCreated()  
    {  
        return shapes.Count;  
    }  
}
```

```
public IRobot GetRobotFromFactory(String robotType)
{
    IRobot robotCategory = null;
    if (shapes.ContainsKey(robotType))
    {
        robotCategory = shapes[robotType];
    }
    else
    {
        switch (robotType)
        {
            case "Small":
                Console.WriteLine("We do not have Small Robot at
present. So we are creating a Small Robot now.");
                robotCategory = new Robot("Small");
                shapes.Add("Small", robotCategory);
                break;
            case "Large":
                Console.WriteLine("We do not have Large Robot at
present. So we are creating a Large Robot now.");
                robotCategory = new Robot("Large");
                shapes.Add("Large", robotCategory);
                break;
            default:
                throw new Exception("Robot Factory can create only
king and queen type robots");
        }
    }
    return robotCategory;
}

//FlyweightPattern is in action.
```

```
class Program
{
    static void Main(string[] args)
    {
        RobotFactory myfactory = new RobotFactory();
        Console.WriteLine("\n***Flyweight Pattern Example
Modified***\n");
        Robot shape;
        /*Here we are trying to get 3 Small type robots*/
        for (int i = 0; i < 3; i++)
        {
            shape = (Robot)myfactory.GetRobotFromFactory("Small");
            /*Not required to add sleep(). But it is included to
            increase the probability of getting a new random number
            to see the variations in the output.*/
            Thread.Sleep(1000);
            shape.SetColor(getRandomColor());
            shape.Print();
        }
        /*Here we are trying to get 3 Large type robots*/
        for (int i = 0; i < 3; i++)
        {
            shape = (Robot)myfactory.GetRobotFromFactory("Large");
            /*Not required to add sleep().But it is included to
            increase the probability of getting a new random number
            to see the variations in the output.*/
            Thread.Sleep(1000);
            shape.SetColor(getRandomColor());
            shape.Print();
        }
        int NumOfDistinctRobots = myfactory.TotalObjectsCreated();
        //System.out.println("\nDistinct Robot objects created till
now= " + NumOfDistinctRobots);
        Console.WriteLine("\n Finally no of Distinct Robot objects
created: " + NumOfDistinctRobots);
    }
}
```

```

        Console.ReadKey();
    }
    static string getRandomColor()
    {
        Random r = new Random();
        /*You can supply any number of your choice in nextInt argument.
         * we are simply checking the random number generated is an
         * even number or an odd number. And based on that we are
         * choosing the color. For simplicity, we'll use only two
         * color-red and green
        */
        int random = r.Next(100);
        if (random % 2 == 0)
        {
            return "red";
        }
        else
        {
            return "green";
        }
    }
}
}

```

Modified Output

Here's the first run of the modified output:

Flyweight Pattern Example Modified

We do not have Small Robot at present. So we are creating a Small Robot now.

This is a Small type robot with redcolor

This is a Small type robot with redcolor

This is a Small type robot with greencolor

We do not have Large Robot at present. So we are creating a Large Robot now.

This is a Large type robot with redcolor
This is a Large type robot with redcolor
This is a Large type robot with greencolor

Finally no of Distinct Robot objects created: 2

Here's the second run of the modified output:

Flyweight Pattern Example Modified

We do not have Small Robot at present. So we are creating a Small Robot now.

This is a Small type robot with greencolor
This is a Small type robot with redcolor
This is a Small type robot with redcolor

We do not have Large Robot at present. So we are creating a Large Robot now.

This is a Large type robot with redcolor
This is a Large type robot with greencolor
This is a Large type robot with redcolor

Finally no of Distinct Robot objects created: 2

Note The outputs vary because I am picking colors randomly in this example.

Q&A Session

1. **Can you highlight the key differences between a Singleton pattern and a Flyweight pattern?**

Answer:

Singleton helps you to maintain almost one object that is required in the system. In other words, once the required object is created, you cannot create more of that. You need to reuse the existing object.

The Flyweight pattern generally concerns with heavy but similar objects because they may occupy big blocks of memory. So, you try to create a smaller set of template objects that can be configured on the fly to make these heavy objects. These smaller and configurable objects are called flyweights. You can reuse them in your application when you deal with many large objects. This approach helps you to reduce the consumption of big chunks of memories. Basically flyweights make *one look like many*. This is why GoF tells us the following: “A flyweight is a shared object that can be used in multiple contexts simultaneously. The flyweight acts as an independent object in each context—it’s indistinguishable from an instance of the object that’s not shared.”

Figure 10-6 shows you how to visualize the core concepts of the Flyweight pattern before using flyweights.

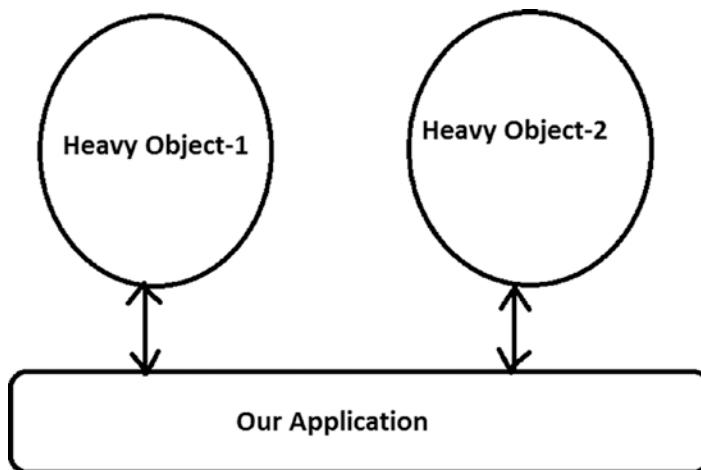


Figure 10-6. Before using flyweights

Figure 10-7 shows the design after using flyweights.

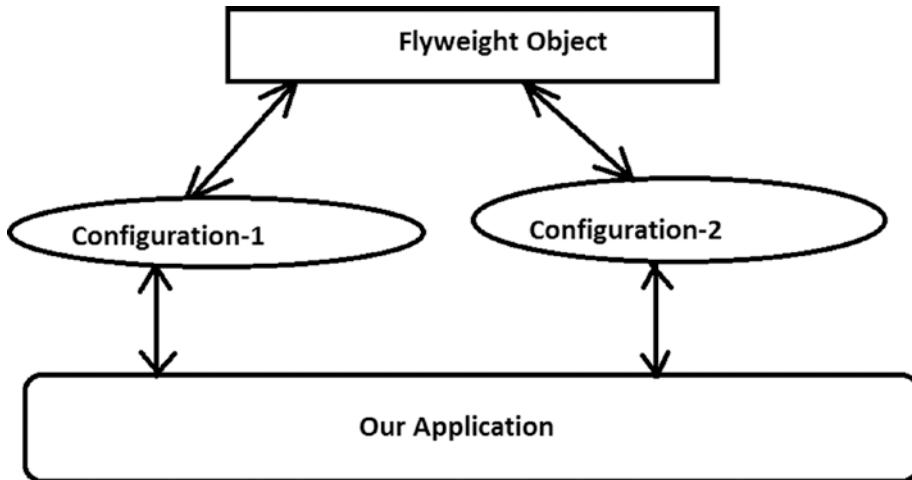


Figure 10-7. After using flyweights

So, from the Figure 10-7, we can see Heavy-Object1 is created when we apply Configuration-1 to the Flyweight Object and similarly, Heavy-Object2 is created when we apply Configuration-2 to the Flyweight Object. You assume that instance specific contents can be passed to the flyweights to make these heavy objects. So, you can notice that in this example, the flyweight object is acting like a common template that can be configured as per the need.

2. If there any impact because of multithreading?

Answer:

If you are creating objects with new operators, definitely in a multithreaded environment you may end up with creating multiple unwanted objects. This is similar to the Singleton pattern, and the remedy is also similar.

3. What are the advantages of using the Flyweight design pattern?

Answer:

- You can reduce memory consumptions of heavy objects that can be controlled identically.

- You can reduce the total number of the objects in the system.
- You can maintain centralized states of many “virtual” objects.

4. What are the challenges associated with using the Flyweight design pattern?

Answer:

- In this pattern, you need to spend some time to configure these flyweights. These configuration times can impact the overall performance of the application.
- To create flyweights, you are extracting a common template class from existing objects. This additional layer of programming can be tricky and sometimes hard to debug and maintain.

5. Can you have nonshareable flyweight interface?

Answer:

Yes, a flyweight interface does not enforce that it needs to be shareable always. So, in some cases, you may have nonshareable flyweights with concrete flyweight objects as children.

6. Since the intrinsic data of flyweights is the same, you can try to share them. Is this understanding correct?

Answer:

Yes.

7. How can clients handle the extrinsic data of these flyweights?

Answer:

They need to pass that information (the states) to the flyweights when they need to use this concept.

8. So, extrinsic data is not shareable. Is this understanding correct?

Answer:

Yes.

CHAPTER 11

Composite Pattern

This chapter covers the Composite pattern.

GoF Definition

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Concept

This pattern is useful to represent part-whole hierarchies of objects. In object-oriented programming, a composite is an object with a composition of one or more similar objects, where each of these objects has similar functionality. (This is also known as a “has-a” relationship among objects.) So, the usage of this pattern is common in tree-structured data. If you can apply the concept properly, you do not need to discriminate between a branch and the leaf nodes. In simple words, you can achieve these two key goals with this pattern:

- You can compose objects into a tree structure to represent a part-whole hierarchy.
- You can access both the composite objects (branches) and the individual objects (leaf nodes) uniformly. As a result, you can reduce the complexity of the code and make the application less prone to errors.

Real-Life Example

Think of an organization that consists of many **departments**. In general, each of these departments consists of multiple employees (in other words, all these participants are basically employees in the organization). Some employees are grouped together to form a department, and those departments can be further grouped together to build the whole organization.

Computer World Example

I already mentioned that any tree data structure can follow this concept. In that case, clients can treat the leaves of the tree and the nonleaves (or, branches of the tree) in the same way.

Note When you traverse the tree, you may need to use the concept of an Iterator design pattern, covered in Chapter 18.

Illustration

In this example, I am representing a college organization. Let's assume we have a principal and **two heads of departments** (HODs), one for computer science and engineering (CSE) and one for mathematics (Maths). Suppose that in the mathematics department, we have two lecturers (teachers), and in the computer science and engineering department you have three lecturers (teachers). So, the tree structures for this organization look like Figure 11-1.

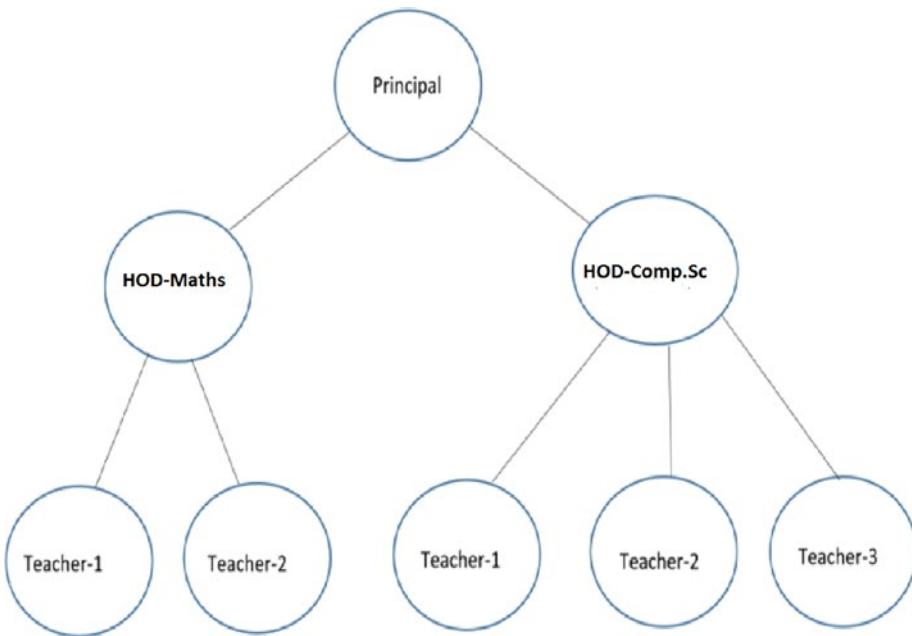


Figure 11-1. A college organization

Let's also assume that at the end of the year, one lecturer from the CSE department retires. The following illustration will deal with all of these scenarios.

Class Diagram

Figure 11-2 shows the class diagram.

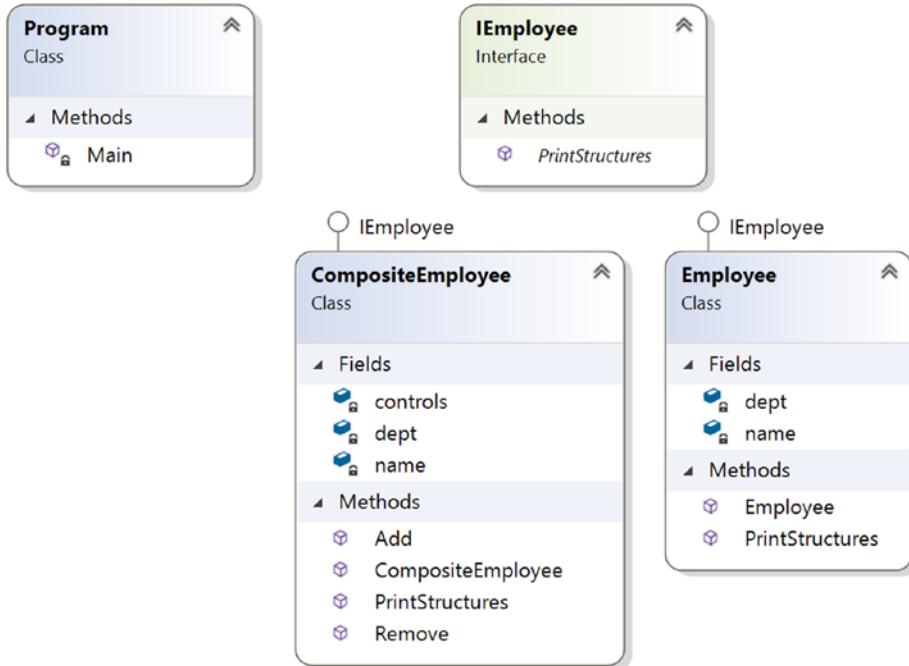


Figure 11-2. Class diagram

Solution Explorer View

Figure 11-3 shows the high-level structure of the parts of the program.

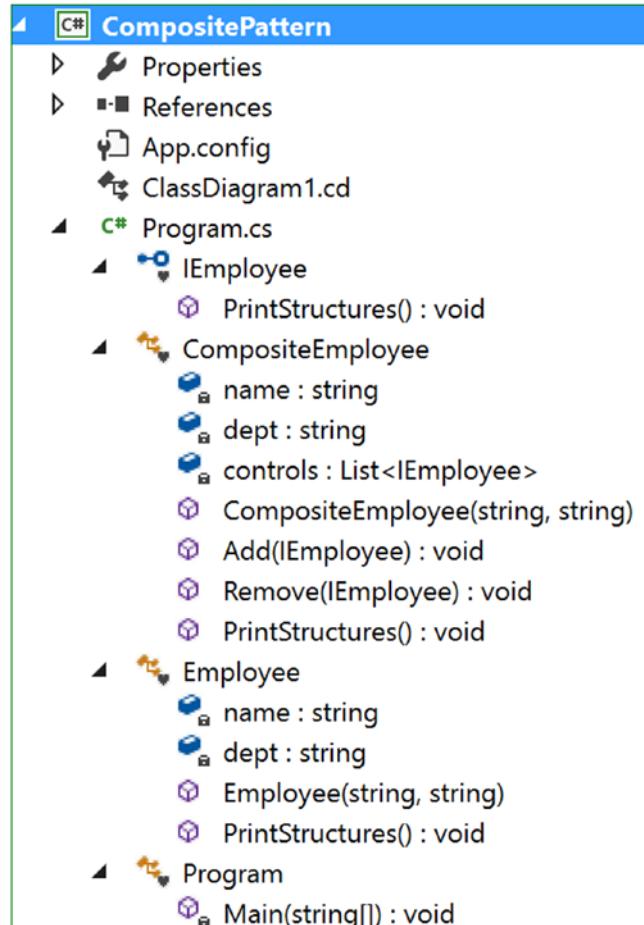


Figure 11-3. Solution Explorer View

Implementation

Here's the implementation:

```
using System;
using System.Collections.Generic;//For List<Employee> here

namespace CompositePattern
{
    interface IEmployee
    {
        void PrintStructures();
    }

    class CompositeEmployee : IEmployee
    {
        private string name;
        private string dept;
        //The container for child objects
        private List<IEmployee> controls;

        // constructor
        public CompositeEmployee(string name, string dept)
        {
            this.name = name;
            this.dept = dept;
            controls = new List<IEmployee>();
        }

        public void Add(IEmployee e)
        {
            controls.Add(e);
        }

        public void Remove(IEmployee e)
        {
            controls.Remove(e);
        }

        public void PrintStructures()
```

```

{
    Console.WriteLine("\t" + this.name + " works in " + this.dept);
    foreach (IEmployee e in controls)
    {
        e.PrintStructures();
    }
}
class Employee : IEmployee
{
    private string name;
    private string dept;
    // constructor
    public Employee(string name, string dept)
    {
        this.name = name;
        this.dept = dept;
    }
    public void PrintStructures()
    {
        Console.WriteLine("\t\t" + this.name + " works in " + this.dept);
    }
}
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine(" *** Composite Pattern Demo *** ");
        //Principal of the college
        CompositeEmployee Principal = new CompositeEmployee("Dr.S.Som",
        (Principal), "Planning-Supervising-Managing");
        //The College has 2 Head of Departments-One from Mathematics,
        One from Computer Sc.
        CompositeEmployee hodMaths = new
        CompositeEmployee("Mrs.S.Das(HOD-Maths)", "Maths");
}

```

```
CompositeEmployee hodCompSc = new CompositeEmployee
("Mr. V.Sarcar(HOD-CSE)", "Computer Sc.");

//2 other teachers works in Mathematics department
Employee mathTeacher1 = new Employee("Math Teacher-1", "Maths");
Employee mathTeacher2 = new Employee("Math Teacher-2", "Maths");

//3 other teachers works in Computer Sc. department
Employee cseTeacher1 = new Employee("CSE Teacher-1", "Computer Sc.");
Employee cseTeacher2 = new Employee("CSE Teacher-2", "Computer Sc.");
Employee cseTeacher3 = new Employee("CSE Teacher-3", "Computer Sc.");

//Teachers of Mathematics directly reports to HOD-Maths
hodMaths.Add(mathTeacher1);
hodMaths.Add(mathTeacher2);

//Teachers of Computer Sc. directly reports to HOD-CSE
hodCompSc.Add(cseTeacher1);
hodCompSc.Add(cseTeacher2);
hodCompSc.Add(cseTeacher3);

//Principal is on top of college
//HOD -Maths and Comp. Sc directly reports to him
Principal.Add(hodMaths);
Principal.Add(hodCompSc);

//Printing the leaf-nodes and branches in the same way.
//i.e. in each case, we are calling PrintStructures() method
Console.WriteLine("\n Testing the structure of a Principal
object");
//Prints the complete structure
Principal.PrintStructures();

Console.WriteLine("\n Testing the structure of a HOD object:");
Console.WriteLine("Teachers working at Computer Science
department:");
//Prints the details of Computer Sc, department
```

```

        hodCompSc.PrintStructures();

        //Leaf node
        Console.WriteLine("\n Testing the structure of a leaf node:");
        mathTeacher1.PrintStructures();

        //Suppose, one computer teacher is leaving now from the
        organization.
        hodCompSc.Remove(cseTeacher2);
        Console.WriteLine("\n After CSE Teacher-2 resigned, the
        organization has following members:");
        Principal.PrintStructures();

        Console.ReadKey();
    }

}
}

```

Output

Here's the output:

Composite Pattern Demo

Testing the structure of a Principal object

Dr.S.Som(Principal) works in Planning-Supervising-Managing

Mrs.S.Das(HOD-Maths) works in Maths

 Math Teacher-1 works in Maths

 Math Teacher-2 works in Maths

Mr. V.Sarcar(HOD-CSE) works in Computer Sc.

 CSE Teacher-1 works in Computer Sc.

 CSE Teacher-2 works in Computer Sc.

 CSE Teacher-3 works in Computer Sc.

Testing the structure of a HOD object:

Teachers working at Computer Science department:

Mr. V.Sarcar(HOD-CSE) works in Computer Sc.

 CSE Teacher-1 works in Computer Sc.

CSE Teacher-2 works in Computer Sc.
CSE Teacher-3 works in Computer Sc.

Testing the structure of a leaf node:

Math Teacher-1 works in Maths

After CSE Teacher-2 resigned, the organization has following members:

Dr.S.Som(Principal) works in Planning-Supervising-Managing

Mrs.S.Das(HOD-Maths) works in Maths

Math Teacher-1 works in Maths

Math Teacher-2 works in Maths

Mr. V.Sarcar(HOD-CSE) works in Computer Sc.

CSE Teacher-1 works in Computer Sc.

CSE Teacher-3 works in Computer Sc.

Q&A Session

1. What are the advantages of using the Composite design pattern?

Answer:

- In a tree-like structure, you can treat both the composite objects (branches) and the individual objects (leaf nodes) uniformly.

Notice that in this example, I have used a common method called PrintStructures to print both the composite object structure (the principal or department heads) and the single objects (the leaf nodes like Math Teacher-1.)

- It is common to implement a part-whole hierarchy using this design pattern.
- You can easily add a new component to the architecture or delete an existing component from the architecture.

2. What are the challenges associated with using the Composite design pattern?

Answer:

- If you want to maintain the ordering of child nodes (for example, if the parse trees are represented as components), you may need to take special care.
- If you are dealing with immutable objects, you cannot simply delete them.
- You can easily add a new component, but maintenance can become difficult over a period of time. Sometimes you may want to deal with a composite that has special components. This kind of constraint may cause additional costs to the development because you may need to implement a dynamic checking mechanism to support the concept.

3. In this example, you used a list data structure. Are other data structures OK to use?

Answer:

Absolutely. There is no universal rule. You are free to use your preferred data structure. The GoF also confirms that it is not necessary to use a general-purpose data structure.

4. How can you connect the Iterator design pattern to a Composite design pattern?

Answer:

If you go through the example, you will notice that if you want to examine a composite object architecture, you may need to iterate over the objects. In other words, if you want to do some special activities with some branches, you may need to iterate over its leaf nodes and nonleaf nodes.

5. In your implementation, in the interface, you defined only one method, **PrintStructures**. But you are using additional methods for the addition and removal of objects in the composite class (**CompositeEmployee**). Why are you not putting these methods in the interface?

Answer:

Nice observation. Even the GoF discussed this. Let's check what will happen if you put the **Add(...)** and **Remove(...)** methods in the interface. In that case, the leaf nodes need to implement these addition and removal operations. In this case, it may appear that you lose transparency, but I think that you have more safety because you have blocked the meaningless operations in leaf nodes. This is why the GoF also mentioned that this kind of decision involves a trade-off between safety and transparency.

6. I want to use an abstract class instead of an interface. Is this allowed?

Answer:

In most cases, the simple answer is yes. But you need to understand the difference between an **abstract class** and an **interface**. In a typical scenario, you will find one of them may be more useful than the other one. Since throughout the book I am presenting only simple and easy-to-understand examples, you may not see much difference between these two.

Note In the “Q&A Session” section of Chapter 3, which covered the Builder pattern, I discussed how to decide between an abstract class and an interface.

CHAPTER 12

Bridge Pattern

This chapter covers the Bridge pattern.

GoF Definition

Decouple an abstraction from its implementation so that the two can vary independently.

Concept

This pattern is also known as the Handle/Body pattern. With it, you decouple an implementation class from an abstract class by providing a bridge between them.

This bridge interface makes the functionality of concrete classes independent from the interface implementer classes. You can alter different kinds of classes structurally without affecting each other.

Real-Life Example

In a software product development company, the development team and the marketing team both play crucial roles. The marketing team does a market survey and gathers the customer requirements. The development team implements those requirements in the product to fulfill the customer needs. Any change (say, in the operational strategy) in one team should not have a direct impact on the other team. In this case, you can think of the marketing team as playing the role of the bridge between the clients of the product and the development team of the software organization.

Computer World Example

GUI frameworks can use the Bridge pattern to separate abstractions from the platform-specific implementation. For example, using this pattern, you can separate a window abstraction from a window implementation for Linux or macOS.

Illustration

Suppose you are a maker of remote control for different electronic items. For simplicity, let's assume you are currently getting orders to make remote controls for televisions and VCD (though nowadays, we use DVD) players and your remote control has two major functionalities: on and off.

Suppose you want to start with the design shown in Figure 12-1 or the one in Figure 12-2.

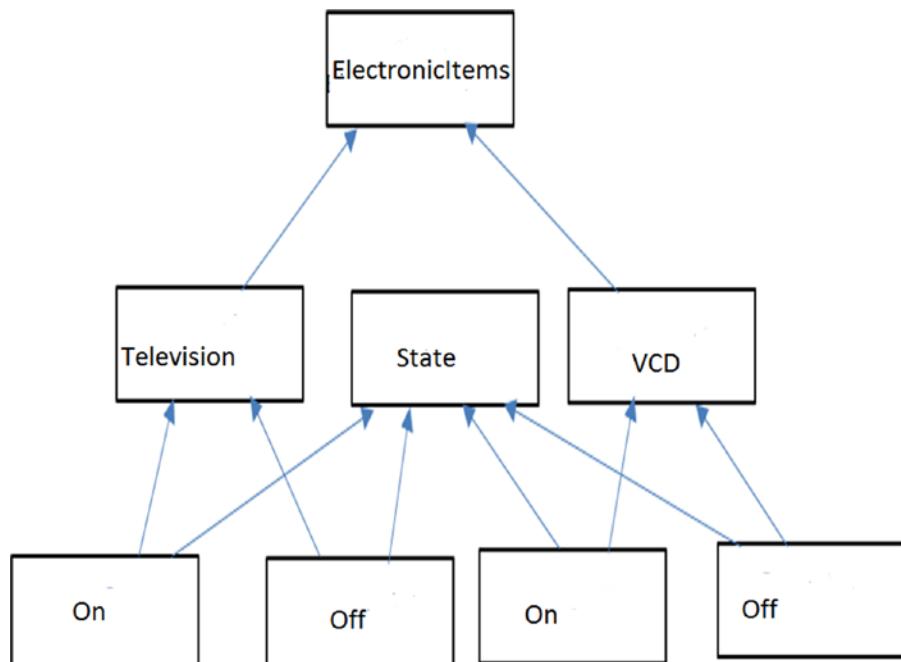


Figure 12-1. Approach 1

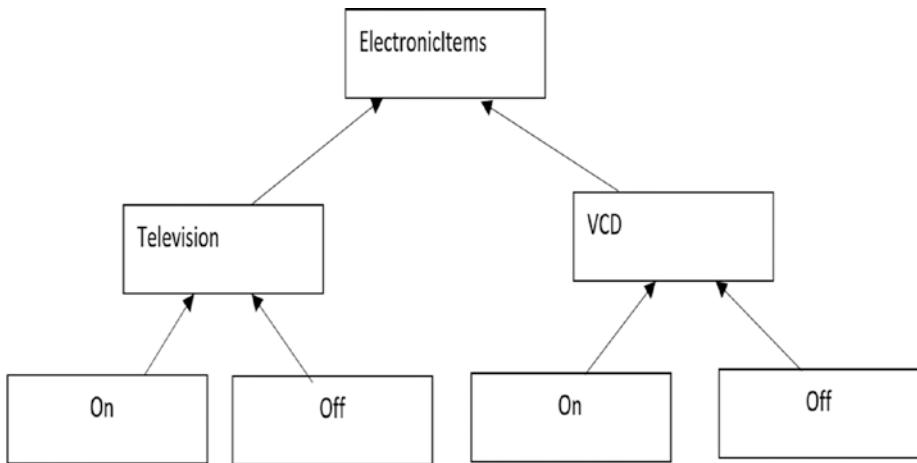


Figure 12-2. Approach 2

On further analysis, you discover that approach 1 is messy and will be difficult to maintain.

At the beginning, approach 2 looks cleaner, but if you want to include new states such as Sleep, Mute, and so on, or if you want to include new electronic items like Air Conditioner (AC), DVD, and so on, you will face new challenges because the elements are tightly coupled in this design. But in a real-world scenario, this kind of enhancement is often required.

So, you need to start with a **loosely coupled** system for future enhancements so that either of these two hierarchies (electronics items and their states) can grow independently. The Bridge pattern is perfect for such a scenario.

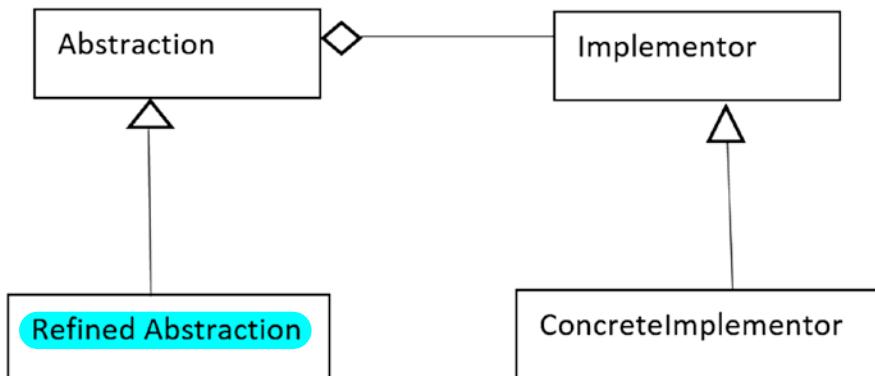


Figure 12-3. A classical Bridge pattern

Let's start from the most common class diagram of a Bridge pattern (see Figure 12-3). In this class diagram:

- Abstraction (an abstract class) defines the abstract interface and maintains the Implementor reference.
- RefinedAbstraction (a concrete class) extends the interface defined by Abstraction.
- Implementor (an interface) defines the interface for implementation classes.
- ConcreteImplementor (a concrete class) implements the Implementor interface.

I have followed a similar architecture in the following implementation. For your ready reference, I have pointed out all of the participants in the implementation with comments.

Class Diagram

Figure 12-4 shows the class diagram.

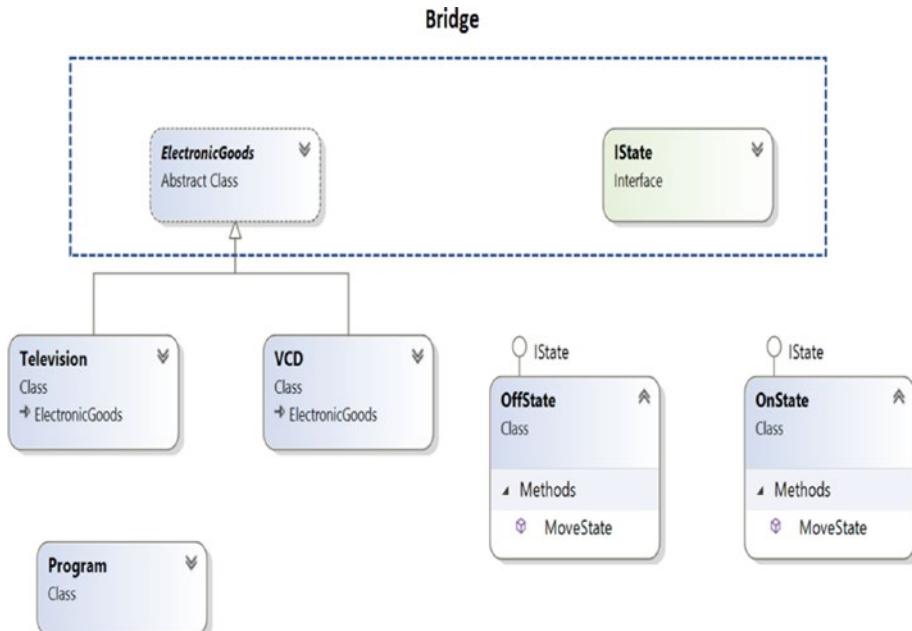


Figure 12-4. Class diagram

Solution Explorer View

Figure 12-5 shows the structure of the parts of the program.

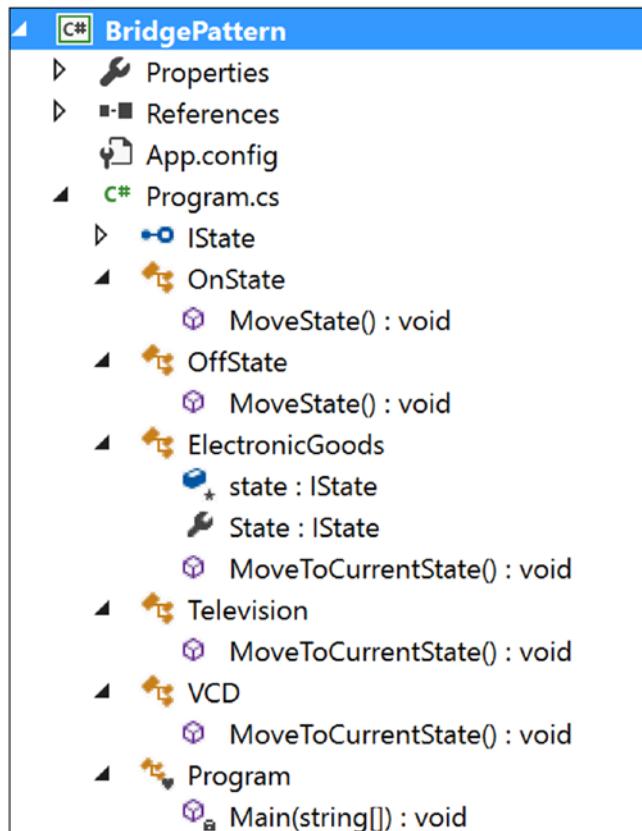


Figure 12-5. Solution Explorer View

Implementation

Here's the implementation:

```

using System;

namespace BridgePattern
{
    //Implementor
    public interface IState
    {

```

CHAPTER 12 BRIDGE PATTERN

```
    void MoveState();
}
//ConcreteImplementor-1
public class OnState : IState
{
public void MoveState()
{
    Console.WriteLine("On State");
}
}
//ConcreteImplementor-2
public class OffState : IState
{
    public void MoveState()
{
    Console.WriteLine("Off State");
}
}
//Abstraction
public abstract class ElectronicGoods
{
    //Composition - implementor
    protected IState state;

    //Alternative approach to properties:
    //we can also pass an implementor (as input argument) inside a
    constructor.
    //public ElectronicGoods(IState state)
    //{
    //    this.state = state;
    //}
    public IState State
    {
        get
        {
            return state;
        }
    }
}
```

```
    set
    {
        state = value;
    }
}
abstract public void MoveTocurrentState();
}

//Refined Abstraction
public class Television : ElectronicGoods
{
    //public Television(IState state) : base(state)
    //{
    //}
    /*Implementation specific:
     * We are delegating the implementation to the Implementor object*/
    public override void MoveTocurrentState()
    {
        Console.WriteLine("\n Television is functioning at : ");
        state.MoveState();
    }
}
public class VCD : ElectronicGoods
{
    //public VCD(IState state) : base(state)
    //{
    //}
    /*Implementation specific:
     * We are delegating the implementation to the Implementor object*/
    public override void MoveTocurrentState()
    {
        Console.WriteLine("\n VCD is functioning at : ");
        state.MoveState();
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***Bridge Pattern Demo***");
        Console.WriteLine("\nDealing with a Television:");

        //ElectronicGoods eItem = new Television(presentState);
        ElectronicGoods eItem = new Television();
        IState presentState = new OnState();
        eItem.State = presentState;
        eItem.MoveToCurrentState();
        //Verifying Off state of the Television now
        presentState = new OffState();
        //eItem = new Television(presentState);
        eItem.State = presentState;
        eItem.MoveToCurrentState();

        Console.WriteLine("\n \n Dealing with a VCD:");
        presentState = new OnState();
        //eItem = new VCD(presentState);
        eItem = new VCD();
        eItem.State = presentState;
        eItem.MoveToCurrentState();

        presentState = new OffState();
        //eItem = new VCD(presentState);
        eItem.State = presentState;
        eItem.MoveToCurrentState();
        Console.ReadLine();
    }
}
```

Output

Here's the output:

```
***Bridge Pattern Demo***
```

Dealing with a Television:

```
Television is functioning at : On State
Television is functioning at : Off State
```

Dealing with a VCD:

```
VCD is functioning at : On State
VCD is functioning at : Off State
```

Q&A Session

- 1. It looks like this pattern is similar to the State pattern. Is this understanding correct?**

Answer:

No. The State pattern is a behavioral pattern, and its intent is different. In this chapter, I showed an example where the electronics items can be in different states, but the key intent was to show the following:

- How you avoid the tight coupling between the items and their states
- How you maintain two different hierarchies where both of them can be extended without impacting each other
- How you deal with multiple objects where implementations are shared among themselves

2. You could use simple subclassing instead of using this kind of design. Is this understanding correct?

Answer:

No. With simple subclassing, your implementations cannot vary dynamically. Your implementations may seem to behave differently, but actually they are bound to the abstraction at compile time.

3. In this example, there is a lot of dead code. Why are you keeping it?

Answer:

Some developers prefer **constructors over properties** (or, **getter-setter methods**). You can see both variations in different implementations. I am keeping them for your ready reference.

4. What are the key advantages of using a Bridge design pattern?

Answer:

- **Implementations are not bound to the abstractions.**
- **Both the abstractions and implementations can grow independently.**
- **Concrete classes are independent from the interface** implementer classes. In other words, changes in one of them do not affect the other. So, you can also vary the interface and the concrete implementations in different ways.

5. What are the challenges associated with this pattern?

Answer:

- **The overall structure may become complex.**
- **Sometimes the Bridge pattern is confused with the Adapter pattern.** (Remember that the key purpose of an Adapter pattern is to deal with incompatible interfaces only.)

I.C: Behavioral Patterns

CHAPTER 13

Visitor Pattern

This chapter covers the Visitor pattern.

GoF Definition

Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Concept

With this pattern, you separate an algorithm from an object structure. So, you can add new operations without modifying the existing architecture. This pattern supports the open/close principle (which says extension is allowed but modification is disallowed for entities such as class, function, modules, and so on).

Note You can experience the true power of this design pattern when you combine it with the Composite pattern, as shown in the modified implementation later in this chapter.

Real-Life Example

Think of a taxi-booking scenario. When the taxi arrives at your door and you enter the vehicle, the visiting taxi takes control of the transportation. It can take you to your destination through a new route that you are not familiar with, and in the worst case, it can alter the destination.

Computer World Example

This pattern is useful when public APIs need to support *plug-in* operations. Clients can then perform their intended operations on a class (with the visiting class) without modifying the source.

Illustration

This is a simple example to represent a Visitor pattern. Figure 13-1 shows two hierarchies; the interface `IOriginalInterface` and the concrete class `MyClass` (which implements the interface `IOriginalInterface`) are placed in an original/existing hierarchy. The interface `IVisitor` and the concrete class `Visitor` are placed in a new hierarchy. The aim is that any modification/update operations through this new class hierarchy should not alter the code in the `IOriginalInterface` (existing) hierarchy.

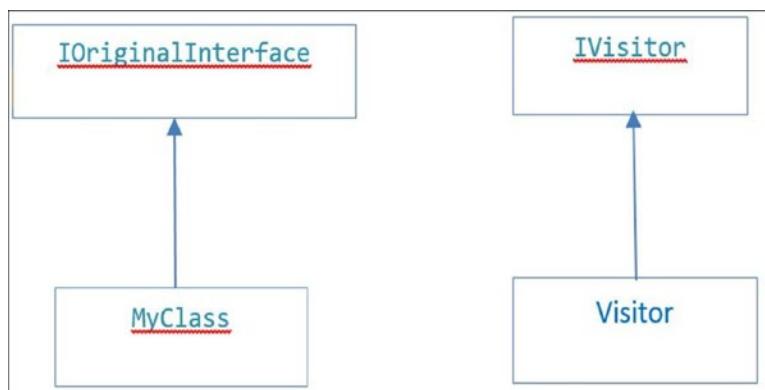


Figure 13-1. Visitor pattern

So, now assume that in this example you want to modify the initial integer value in `MyClass` but your constraint is that you cannot change the original code in `IOriginalInterface`. You can use a Visitor pattern in such a scenario.

So, in the following example, you are separating the functionality implementations (in other words, the algorithms) from the class hierarchy and fulfilling your intention.

Class Diagram

Figure 13-2 shows the class diagram.

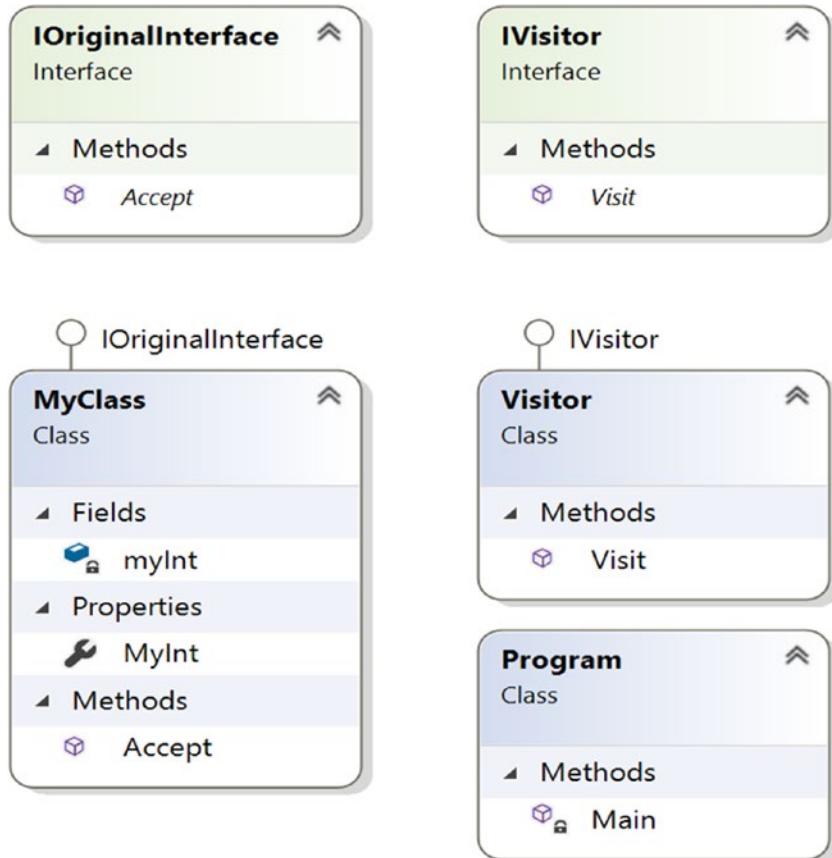


Figure 13-2. Class diagram

Solution Explorer View

Figure 13-3 shows the high-level structure of the parts of the program.

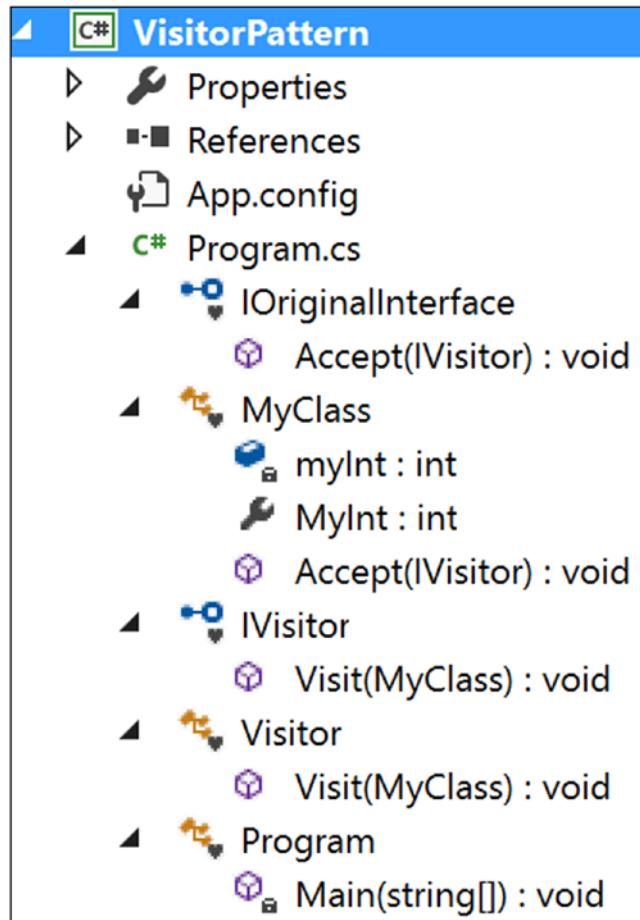


Figure 13-3. Solution Explorer View

Implementation

Here's the implementation:

```
using System;

namespace VisitorPattern
{
    interface IOriginalInterface
```

```
{  
    void Accept(IVisitor visitor);  
}  
class MyClass : IOriginalInterface  
{  
    private int myInt = 5;//Initial or default value  
  
    public int MyInt  
    {  
        get  
        {  
            return myInt;  
        }  
        set  
        {  
            myInt = value;  
        }  
    }  
    public void Accept(IVisitor visitor)  
    {  
        Console.WriteLine("Initial value of the integer:{0}", myInt);  
        visitor.Visit(this);  
        Console.WriteLine("\nValue of the integer now:{0}", myInt);  
    }  
}  
  
interface IVisitor  
{  
    void Visit(MyClass myClassElement);  
}  
class Visitor : IVisitor  
{  
    public void Visit(MyClass myClassElement)  
    {  
        Console.WriteLine("Visitor is trying to change the integer  
        value.");  
    }  
}
```

```

        myClassElement.MyInt = 100;
        Console.WriteLine("Exiting from Visitor.");
    }
}
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***Visitor Pattern Demo***\n");
        IVisitor visitor = new Visitor();
        MyClass myClass = new MyClass();
        myClass.Accept(visitor);
        Console.ReadLine();
    }
}
}

```

Output

Here's the output:

```

***Visitor Pattern Demo***

Initial value of the integer:5
Visitor is trying to change the integer value.
Exiting from Visitor.

Value of the integer now:100

```

Modified Illustration

You already saw a simple example of the Visitor design pattern. But you can exercise the true power of this design pattern when you combine it with the Composite pattern. So, let's examine a scenario where you may need to use both the Composite pattern and the Visitor pattern. Notice that in the following implementation, I am using "foreach" statements which can consume an iterator from client code. So, we can say that actually all 3 patterns- Visitor pattern, Composite pattern and Iterator pattern are combined in this example.

Key Characteristics of the Modified Example

Let's consider the example of the Composite design pattern from Chapter 11. In that example, there is a college with two different departments. Each of these departments has one head of department (HOD) and multiple professors/lecturers. All of these HODs report to the principal of the college. Figure 13-4 shows the tree structure for this example.

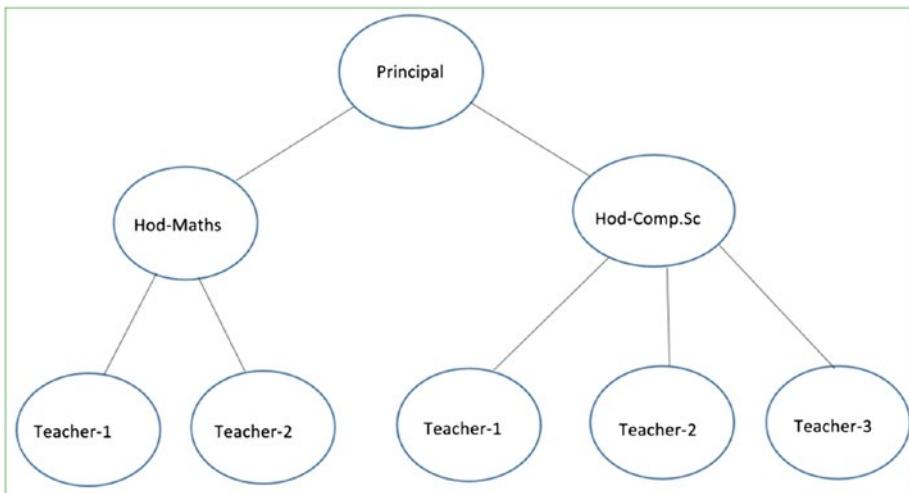


Figure 13-4. Tree structure of the Composite design example

Now suppose the principal of the college wants to promote a few employees. Let's say that teaching experience is the only criteria to promote someone, but the criteria should vary among senior teachers and junior teachers. For a junior teacher, the minimum criteria for promotion is 12 years, and for senior teachers, it is 15 years.

So, you need to introduce a new field, say `yearsOfExperience`. When a visitor gathers the necessary details from this college, it will show the eligible candidates for promotion.

Notice that the visitor is collecting the data one piece at a time from the original college structure without making any modifications to it, and once the collection process is over, the visitor analyzes the data to display the intended results. To understand this visually, you can follow the arrows in the upcoming figures. *The principal is at the top of the organization, so you can assume that no promotion is required for that person.*

Step 1

Figure 13-5 shows step 1.

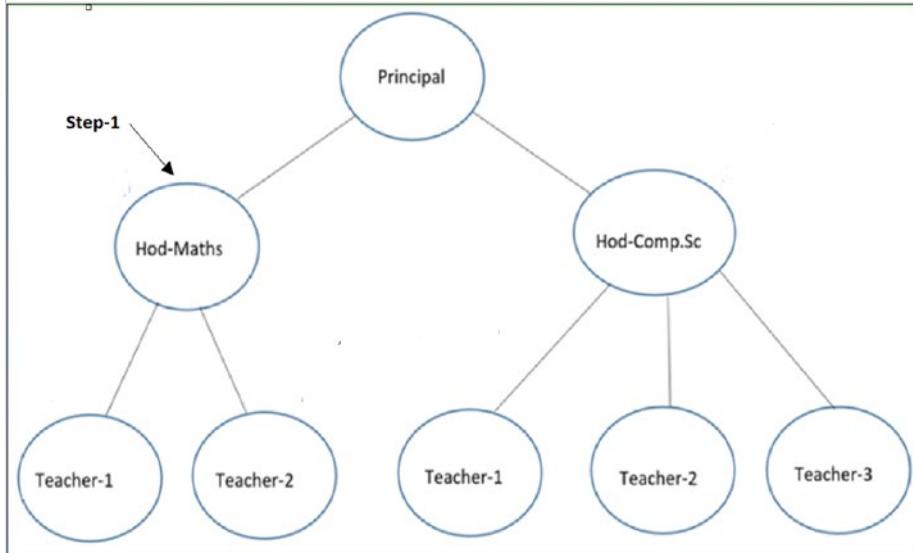


Figure 13-5. Step 1

Step 2

Figure 13-6 shows step 2.

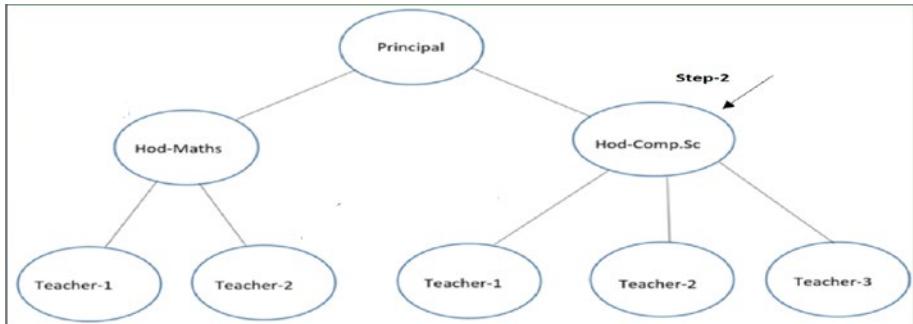


Figure 13-6. Step 2

Step 3

Figure 13-7 shows step 3.

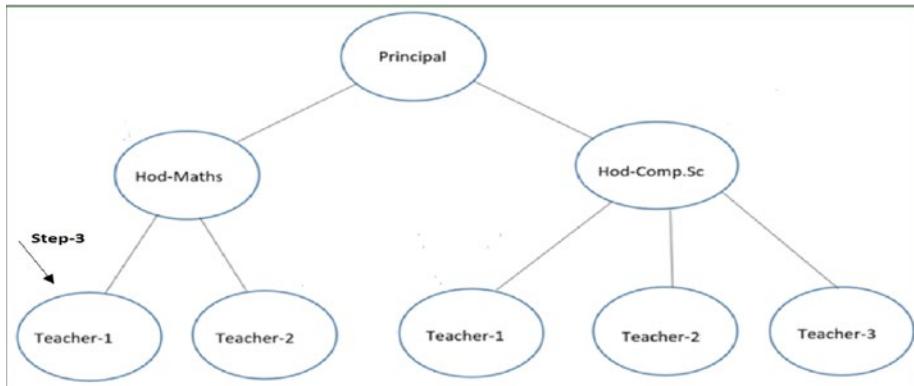


Figure 13-7. Step 3

Step 4

Figure 13-8 shows step 4.

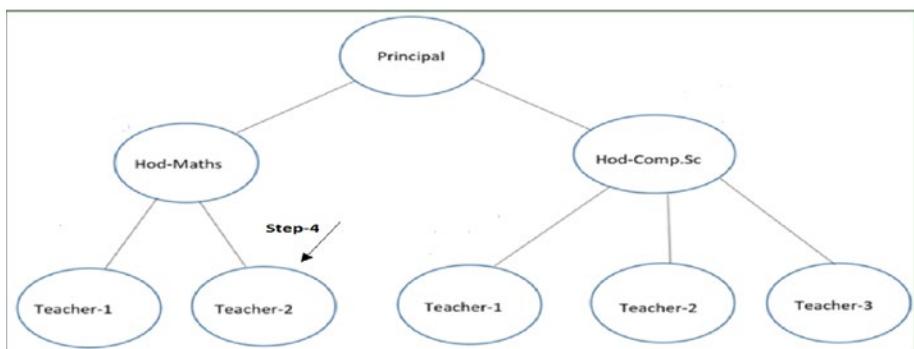


Figure 13-8. Step 4

Step 5

Figure 13-9 shows step 5.

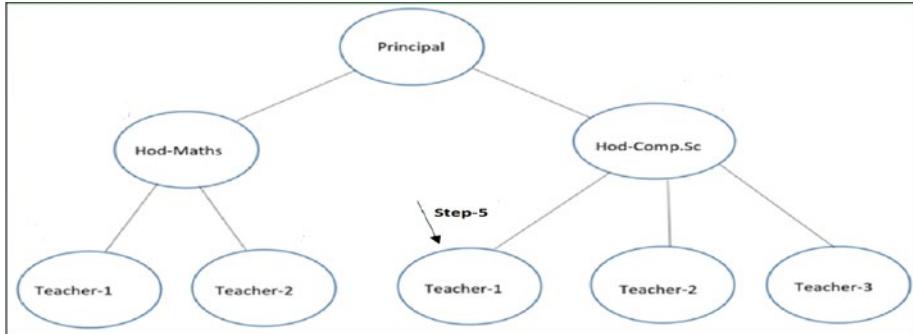


Figure 13-9. Step 5

And so on...

Now let us go through the following implementation.

Modified Solution Explorer View

Figure 13-10 shows the high-level structure of the parts of the program.



Figure 13-10. Modified Solution Explorer View

Modified Implementation

Here's the implementation:

```
using System;
using System.Collections.Generic;

namespace VisitorPatternWithCompositePatternEx
{
    interface IEmployee
    {
        void PrintStructures();
        void Accept(IVisitor visitor);
    }

    //Employees who have Subordinates
    class CompositeEmployee : IEmployee
    {
        private string name;
        private string dept;
        //New field for this example
        private int yearsOfExperience;
        //The container for child objects
        private List<IEmployee> controls;
        // constructor
        public CompositeEmployee(string name, string dept, int experience)
        {
            this.name = name;
            this.dept = dept;
            this.yearsOfExperience = experience;
            controls = new List<IEmployee>();
        }
        public void Add(IEmployee e)
        {
            controls.Add(e);
        }
    }
}
```

```
public void Remove(IEmployee e)
{
    controls.Remove(e);
}
// Gets the name
public string Name
{
    get {return this.name;}
    //set {_name = value;}
}
// Gets the department name
public string Dept
{
    get {return this.dept;}
}
// Gets the yrs. of experience
public int Experience
{
    get {return this.yearsOfExperience;}
}
public List<IEmployee> Controls
{
    get {return this.controls;}
}
public void PrintStructures()
{
    Console.WriteLine("\t" + this.name + " works in " + this.dept
    + " Experience :" + this.yearsOfExperience + " years");
    foreach (IEmployee e in controls)
    {
        e.PrintStructures();
    }
}
```

```
public void Accept(IVisitor visitor)
{
    visitor.VisitCompositeElement(this);
}
}

class Employee : IEmployee
{
    private string name;
    private string dept;
    //New field for this example
    private int yearsOfExperience;
    // constructor
    public Employee(string name, string dept, int experience)
    {
        this.name = name;
        this.dept = dept;
        this.yearsOfExperience = experience;
    }
    public void PrintStructures()
    {
        Console.WriteLine("\t\t" + this.name + " works in " + this.
            dept + " Experience :" + this.yearsOfExperience + " years");
    }
    // Gets the name
    public string Name
    {
        get {return this.name;}
    }
    // Gets the department name
    public string Dept
    {
        get {return this.dept;}
        //set {_name = value;}
    }
}
```

```
// Gets the yrs. of experience
public int Experience
{
    get {return this.yearsOfExperience;}
}
public void Accept(IVisitor visitor)
{
    visitor.VisitLeafNode(this);
}
}

interface IVisitor
{
    void VisitCompositeElement(CompositeEmployee employees);
    void VisitLeafNode(Employee employee);
}
class Visitor : IVisitor
{
    public void VisitCompositeElement(CompositeEmployee employee)
    {
        //We'll promote them if experience is greater than 15 years
        bool eligibleForPromotion = employee.Experience > 15 ? true :
        false;
        Console.WriteLine("\t\t" + employee.Name + " from"
        + employee.Dept + " is eligible for promotion?" +
        eligibleForPromotion);
    }
    public void VisitLeafNode(Employee employee)
    {
        //We'll promote them if experience is greater than 12 years
        bool eligibleForPromotion = employee.Experience > 12 ? true :
        false;
```

```

Console.WriteLine("\t\t" + employee.Name + " from"
+ employee.Dept + " is eligible for promotion?" +
eligibleForPromotion);
}

}

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("****Visitor Pattern combined with Composite
Pattern Demo****\n");
        #region Similar code structure taken from Composite Pattern
        demo
        //Prinipal of the college
        CompositeEmployee Principal = new CompositeEmployee("Dr.S.Som
(Principal)", "Planning-Supervising-Managing",20);
        //The college has 2 Head of Departments-One from Maths, one
        from Computer Sc.
        CompositeEmployee hodMaths = new
        CompositeEmployee("Mrs.S.Das(HOD-Maths)", "Maths",14);
        CompositeEmployee hodCompSc = new CompositeEmployee("Mr.
V.Sarcar(HOD-CSE)", "Computer Sc.",16);

        //2 other teachers works in Mathematics department
        Employee mathTeacher1 = new Employee("Math Teacher-1",
        "Maths", 14);
        Employee mathTeacher2 = new Employee("Math Teacher-2",
        "Maths", 6);

        //3 other teachers works in Computer Sc. department
        Employee cseTeacher1 = new Employee("CSE Teacher-1",
        "Computer Sc.", 10);
        Employee cseTeacher2 = new Employee("CSE Teacher-2",
        "Computer Sc.", 13);
        Employee cseTeacher3 = new Employee("CSE Teacher-3",
        "Computer Sc.", 7);
    }
}

```

```
//Teachers of Mathematics directly reports to HOD-Maths  
hodMaths.Add(mathTeacher1);  
hodMaths.Add(mathTeacher2);  
  
//Teachers of Computer Sc directly reports to HOD-Comp.Sc  
hodCompSc.Add(cseTeacher1);  
hodCompSc.Add(cseTeacher2);  
hodCompSc.Add(cseTeacher3);  
  
//Principal is on top of college  
//HOD -Maths and Comp. Sc directly reports to him  
Principal.Add(hodMaths);  
Principal.Add(hodCompSc);  
  
Console.WriteLine("\n Testing the overall structure");  
//Prints the complete structure  
Principal.PrintStructures();  
#endregion  
  
Console.WriteLine("\n***Visitor starts visiting our composite  
structure***\n");  
IVisitor aVisitor = new Visitor();  
/*Principal is already holding the highest position.  
We are not checking whether he is eligible for promotion or  
not*/  
//Principal.Accept(aVisitor);  
  
//For employees who directly reports to Principal  
foreach (IEmployee e in Principal.Controls)  
{  
    e.Accept(aVisitor);  
}  
//For employees who directly reports to HOD-Maths  
foreach (IEmployee e in hodMaths.Controls)  
{  
    e.Accept(aVisitor);  
}
```

```

//For employees who directly reports to HOD-Comp.Sc
foreach (IEmployee e in hodCompSc.Controls)
{
    e.Accept(aVisitor);
}

Console.ReadLine();
}
}
}
}

```

Modified Output

Here's the output:

```

***Visitor Pattern combined with Composite Pattern Demo***

Testing the overall structure
Dr.S.Som(Principal) works in Planning-Supervising-Managing Experience :20
years
Mrs.S.Das(HOD-Maths) works in Maths Experience :14 years
    Math Teacher-1 works in Maths Experience :14 years
    Math Teacher-2 works in Maths Experience :6 years
Mr. V.Sarcar(HOD-CSE) works in Computer Sc. Experience :16 years
    CSE Teacher-1 works in Computer Sc. Experience :10 years
    CSE Teacher-2 works in Computer Sc. Experience :13 years
    CSE Teacher-3 works in Computer Sc. Experience :7 years

```

Visitor starts visiting our composite structure

```

Mrs.S.Das(HOD-Maths) from Maths is eligible for promotion? False
Mr. V.Sarcar(HOD-CSE) from Computer Sc. is eligible for promotion? True
Math Teacher-1 from Maths is eligible for promotion? True
Math Teacher-2 from Maths is eligible for promotion? False
CSE Teacher-1 from Computer Sc. is eligible for promotion? False
CSE Teacher-2 from Computer Sc. is eligible for promotion? True
CSE Teacher-3 from Computer Sc. is eligible for promotion? False

```

Q&A Session

1. When should you consider implementing the Visitor design pattern?

Answer:

You should use Visitor when you need to add capabilities without modifying the existing architecture. This is one of the primary aims of implementing a Visitor pattern. In this pattern, encapsulation is not the primary concern.

2. Are there any drawbacks associated with this pattern?

Answer:

- I mentioned earlier that encapsulation is not its key concern. So, in many cases, you can destroy the power of encapsulation using visitors.
- If you need to add new concrete classes to an existing architecture frequently, the visitor hierarchy becomes difficult to maintain. For example, suppose you want to add another concrete class in the original hierarchy. In this case, you need to modify the visitor class hierarchy accordingly to fulfill your purposes.

3. Why are you saying that a visitor class can violate encapsulation?

Answer:

In the illustration, I tested a simple Visitor design pattern and updated the integer value of myInt through the visitor class. Also, in many cases, you may see that the visitor needs to move around a composite structure to gather information from that, and then you can play with those information. To provide this kind of support, you are surely violating the core aim of encapsulation.

CHAPTER 14

Observer Pattern

This chapter covers the Observer pattern.

GoF Definition

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Concept

In this pattern, there are many observers (objects) that are observing a particular subject (also an object). Observers want to be notified when there is a change made inside the subject. So, they register themselves to that subject. When they lose interest in the subject, they simply unregister from the subject. Sometimes this model is referred as the Publisher-Subscriber model. The whole idea can be summarized as follows: using this pattern, an object (subject) can send notifications to multiple observers (a set of objects) at the same time.

You can visualize the scenarios with the following diagrams.

In step 1, observers are requesting to get notifications from a subject (see Figure 14-1).

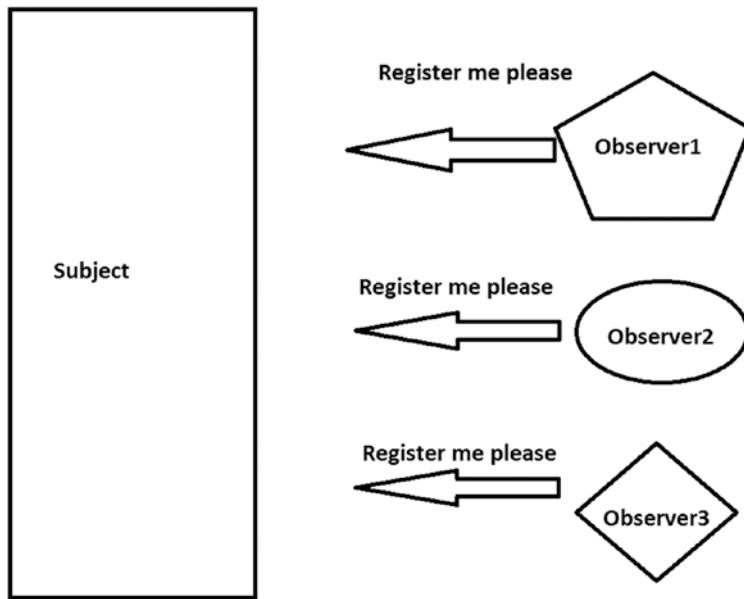


Figure 14-1. Step 1

In step 2, the subject grants the requests; in other words, a connection is established (Figure 14-2).

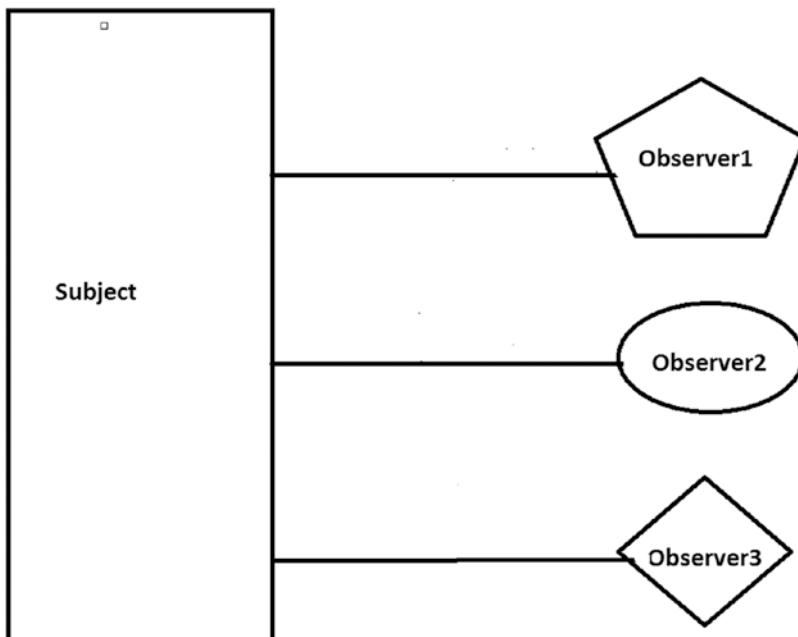


Figure 14-2. Step 2

In step 3, the subject is sending notifications to the registered users (see Figure 14-3).

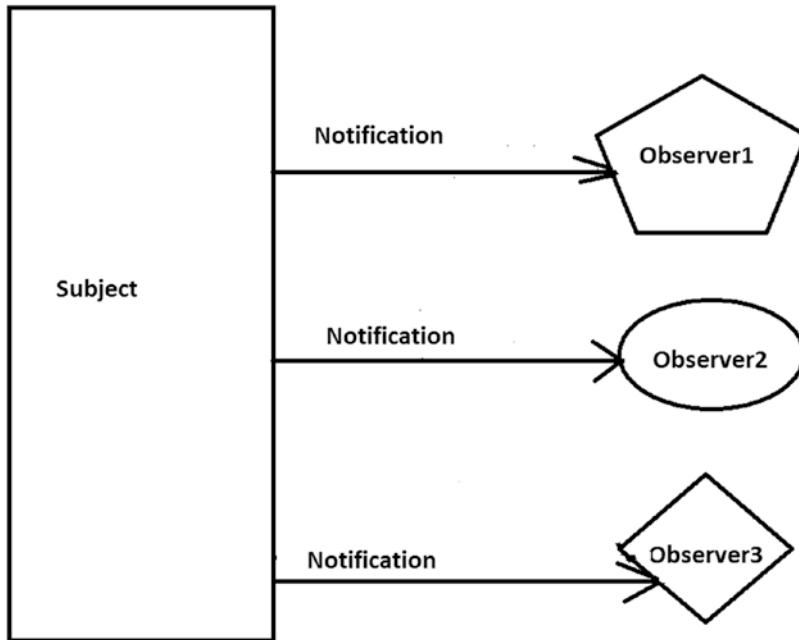


Figure 14-3. Step 3

In step 4 (optional), Observer2 does not want to get further notification. So, the subject has unregistered itself (see Figure 14-4).

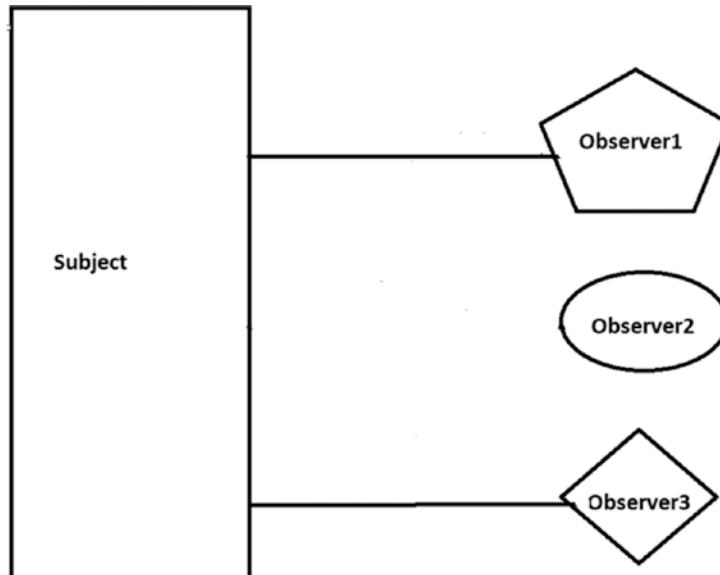


Figure 14-4. Step 4

In step 5, from now on, only Observer1 and Observer3 are getting notifications from the subject (see Figure 14-5).

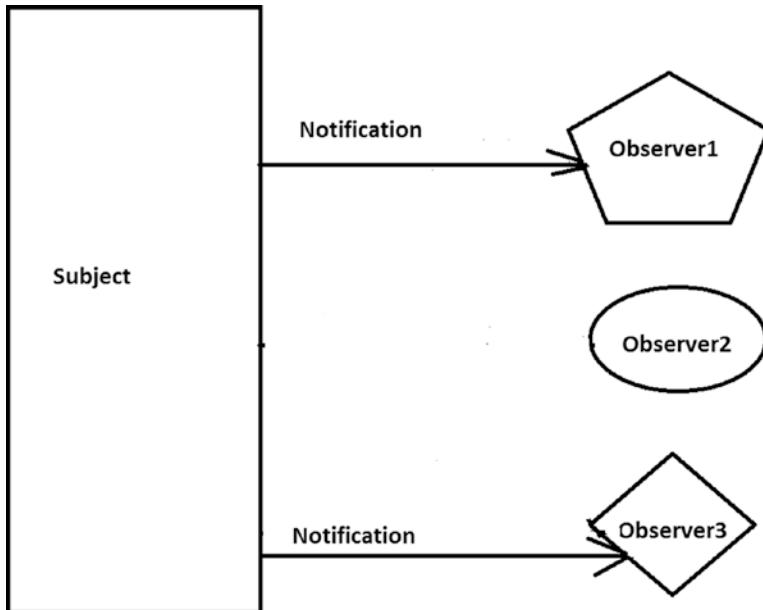


Figure 14-5. Step 5

Real-Life Example

Think about a celebrity who has many followers on social media. Each of these followers wants to get all the latest updates from their favorite celebrity. So, they follow the celebrity until their interest wanes. When they lose interest, they simply do not follow that celebrity. Think of each of these fans or followers as an observer and the celebrity as the subject.

Computer World Example

In the world of computer science, consider a simple UI-based example. This UI is connected to some database. A user can execute some query through that UI, and after searching the database, the result is returned in the UI. With this pattern, you segregate the UI from the database. If a change occurs in the database, the UI should be notified so that it can update its display according to the change.

To simplify this scenario, assume that you are the person responsible for maintaining a particular database in your organization. Whenever there is a change made to the database, you want to get a notification so that you can take action if necessary.

Illustration

For this example, I have created three observers and one subject. The subject maintains a list of all of its registered users. The observers want to receive a notification when the flag value changes in the subject. In the output, you will discover that these observers are getting notifications when the flag values are changed to 5, 50, and 100, respectively. But one of them does not receive any notification when the flag value changes to 50 because at that moment he was not a registered user in the subject. But at the end, he starts getting notifications again because he registers again.

Class Diagram

Figure 14-6 shows the class diagram.

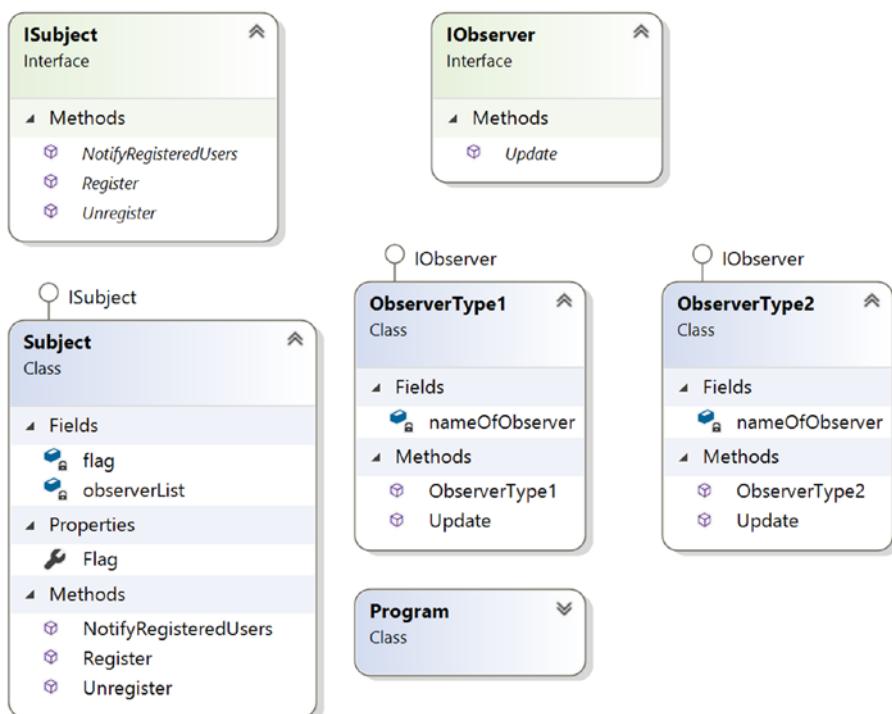


Figure 14-6. The class diagram

Solution Explorer View

Figure 14-7 shows the high-level structure of the parts of the program.

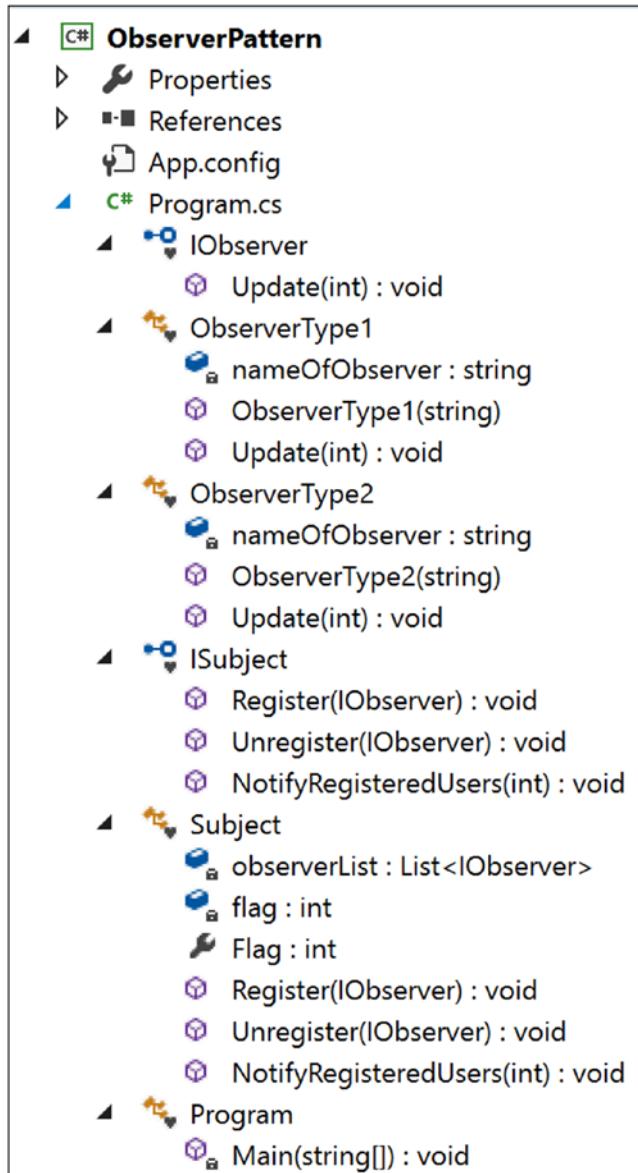


Figure 14-7. Solution Explorer View

Implementation

Here's the implementation:

```
using System;
using System.Collections.Generic;//We have used List<Observer> here
namespace ObserverPattern
{
    interface IObserver
    {
        void Update(int i);
    }
    class ObserverType1 : IObserver
    {
        string nameOfObserver;
        public ObserverType1(String name)
        {
            this.nameOfObserver = name;
        }
        public void Update(int i)
        {
            Console.WriteLine("{0} has received an alert: Someone has
                updated myValue in Subject to: {1}", nameOfObserver,i);
        }
    }
    class ObserverType2 : IObserver
    {
        string nameOfObserver;
        public ObserverType2(String name)
        {
            this.nameOfObserver = name;
        }
    }
}
```

CHAPTER 14 OBSERVER PATTERN

```
public void Update(int i)
{
    Console.WriteLine("{0} notified: myValue in Subject at
present: {1}", nameOfObserver, i);
}

interface ISubject
{
    void Register(IObserver o);
    void Unregister(IObserver o);
    void NotifyRegisteredUsers(int i);
}

class Subject:ISubject
{
    List<IObserver> observerList = new List<IObserver>();
    private int flag;
    public int Flag
    {
        get
        {
            return flag;
        }
        set
        {
            flag = value;
            //Flag value changed. So notify observer/s.
            NotifyRegisteredUsers(flag);
        }
    }
    public void Register(IObserver anObserver)
    {
        observerList.Add(anObserver);
    }
}
```

```
public void Unregister(I0bserver anObserver)
{
    observerList.Remove(anObserver);
}
public void NotifyRegisteredUsers(int i)
{
    foreach (I0bserver observer in observerList)
    {
        observer.Update(i);
    }
}
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("/**Observer Pattern Demo**\n");
        //We have 3 observers-2 of them are ObserverType1, 1 of them
        is of ObserverType2
        I0bserver my0bserver1 = new ObserverType1("Roy");
        I0bserver my0bserver2 = new ObserverType1("Kevin");
        I0bserver my0bserver3 = new ObserverType2("Bose");
        Subject subject = new Subject();
        //Registering the observers-Roy, Kevin, Bose
        subject.Register(my0bserver1);
        subject.Register(my0bserver2);
        subject.Register(my0bserver3);
        Console.WriteLine(" Setting Flag = 5 ");
        subject.Flag = 5;
        //Unregistering an observer(Roy))
        subject.Unregister(my0bserver1);
        //No notification this time Roy. Since it is unregistered.
        Console.WriteLine("\n Setting Flag = 50 ");
        subject.Flag = 50;
```

```

//Roy is registering himself again
subject.Register(myObserver1);
Console.WriteLine("\n Setting Flag = 100 ");
subject.Flag = 100;
Console.ReadKey();
}
}
}

```

Output

Here's the output:

*****Observer Pattern Demo*****

Setting Flag = 5

Roy has received an alert: Someone has updated myValue in Subject to: 5
 Kevin has received an alert: Someone has updated myValue in Subject to: 5
 Bose notified: myValue in Subject at present: 5

Setting Flag = 50

Kevin has received an alert: Someone has updated myValue in Subject to: 50
 Bose notified: myValue in Subject at present: 50

Setting Flag = 100

Kevin has received an alert: Someone has updated myValue in Subject to: 100
 Bose notified: myValue in Subject at present: 100

Roy has received an alert: Someone has updated myValue in Subject to: 100

Analysis of the Output

Notice that initially all three observers (Roy, Kevin, and Bose) registered themselves to get notifications from the subject. So, in the initial phase, all of them received notifications. But then, Roy was not interested in getting further notifications, so he unregistered himself. At this time, only Kevin and Bose were receiving notifications (when the flag value is 50). But Roy changed his mind and he again registers to get notifications. So, at the end, all of them are receiving notifications from the subject.

Q&A Session

- If I have only one observer, then I may not need to set up the interface. Is this understanding correct?**

Answer:

Yes. But if you want to follow the pure object-oriented programming guidelines, you may always prefer interfaces (or abstract classes) instead of using a concrete class. Aside from this point, usually you will have multiple observers, and you will want to keep the methods they implement consistent. That's where you will get benefit from this kind of design.

- Can you have different observers that may vary?**

Answer:

Yes. You should not think that for each observer you need to create a different class. Also, think about this in a real-world scenario. When anyone is making a crucial change in the organization's database, multiple groups of people from different departments may want to know about the change (such as your boss and the owner of the database, who work at different levels) and act accordingly. So, if you create separate classes for each of them, it will be hard to maintain and at the same time, it will be difficult to identify the source of the changes at a given point of time.

- Can you add or remove observers at runtime?**

Answer:

Yes. Notice that at the beginning of the program, to get notifications, Roy registers himself. Then he unregisters himself and reregisters.

4. It appears to me that there are similarities between the Observer pattern and the Chain of Responsibility pattern. Is this understanding correct?

Answer:

In an Observer pattern, all registered users get notifications at the same time, but in the case of the Chain of Responsibility pattern, objects in the chain are notified one by one, which will happen until an object handles the notification fully. Figure 14-8 and Figure 14-9 summarize the difference.

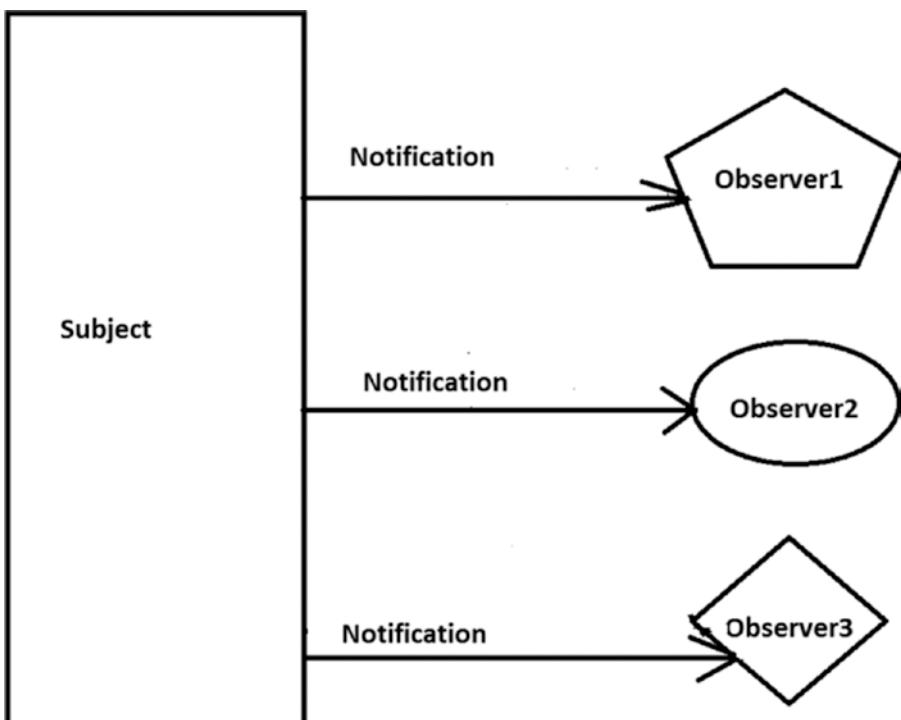


Figure 14-8. Observer pattern

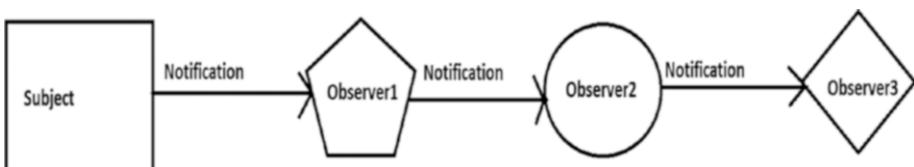


Figure 14-9. Chain of Responsibility pattern

In Figure 14-9, I assume that Observer3 was able to process the notification completely. So, it is the end node of the chain.

5. Does this model support one-to-many relationships?

Answer:

Yes. Since a subject can send notifications to multiple observers, this kind of dependency is clearly depicting a one-to-many relationship.

Note the following:

- In general, you will see the presence of this pattern in event-driven software. Modern languages like C# have built-in support for handling these events following this pattern. These constructs will make your life easier.
- If you are already familiar with the .NET Framework, you will see that in C#, you already have generic `System.IEnumerable<T>` and `System.IObserver<T>` interfaces where the generic type parameter is used to provide notifications. For a detailed reading, you can refer to this page:

<https://docs.microsoft.com/en-us/dotnet/standard/events/observer-design-pattern>

6. If these ready-made constructs exist, why are you writing your own code?

Answer:

You cannot change ready-made functionalities, but I believe that when you try to implement the concept yourself, you gain an understanding that can help you to better use those ready-made constructs.

7. What are the key benefits of the Observer pattern?

Answer:

Subjects and their registered users (observers) make up a loosely coupled system. They do not need to know each other explicitly. Also, you can add or remove observers at runtime independently.

8. What are the key challenges associated with an Observer pattern?

Answer:

Undoubtedly, memory leak is the greatest concern when you deal with events in C# (also referred as a *lapsed listener problem* in some cases). An automatic garbage collector may not always help you in this context. I have dedicated an entire chapter in this book to handle leaks in your applications; see Chapter 29.

CHAPTER 15

Strategy (Policy) Pattern

This chapter covers the Strategy pattern.

GoF Definition

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Concept

You can select the behavior of an algorithm dynamically at runtime.

Real-Life Example

In a soccer match, if Team A is leading 1–0 over Team B toward the end of the game, instead of attacking, Team A becomes defensive to maintain the lead. At the same time, Team B goes for an all-out attack to score the equalizer.

Computer World Example

Suppose you have a backup memory slot. If your primary memory is full and you need to store more data, you can store it in the backup memory slot. If you do not have this backup memory slot and you try to store additional data into your primary memory (when it is full), then that data will be discarded, you will receive exceptions, or you will encounter some peculiar behavior (based on the architecture of the program). So, a runtime check is necessary before storing the data, and then you can proceed.

Illustration

In this example's `Program.cs` file, I have tested for two arbitrary choices of users (though you can have as many as you want). Depending on the user's input, your context objects will decide what choice should be set and displayed.

Class Diagram

Figure 15-1 shows the class diagram.

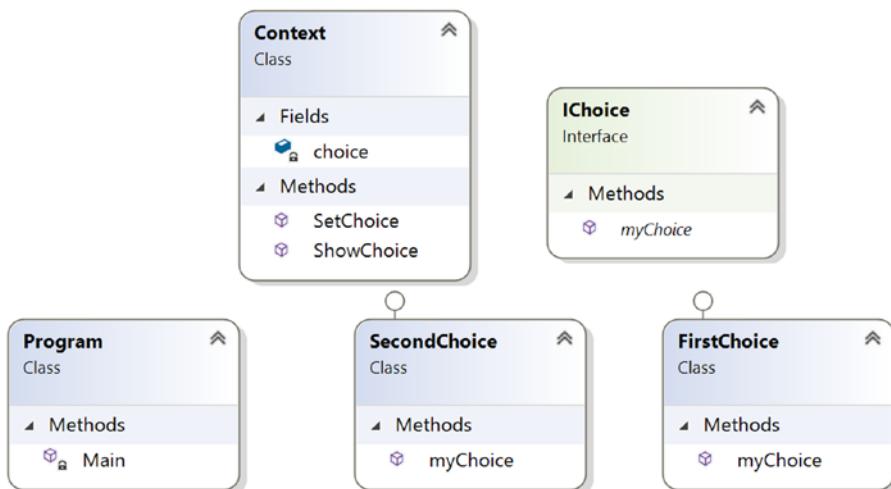


Figure 15-1. Class diagram

Directed Graph Document

Figure 15-2 shows the directed graph documents.

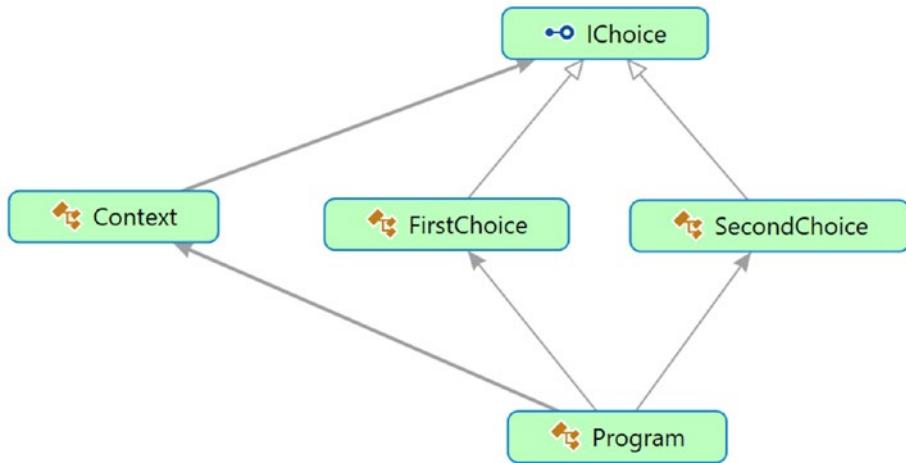


Figure 15-2. *Directed Graph Document*

Solution Explorer View

Figure 15-3 shows the high-level structure of the parts of the program.

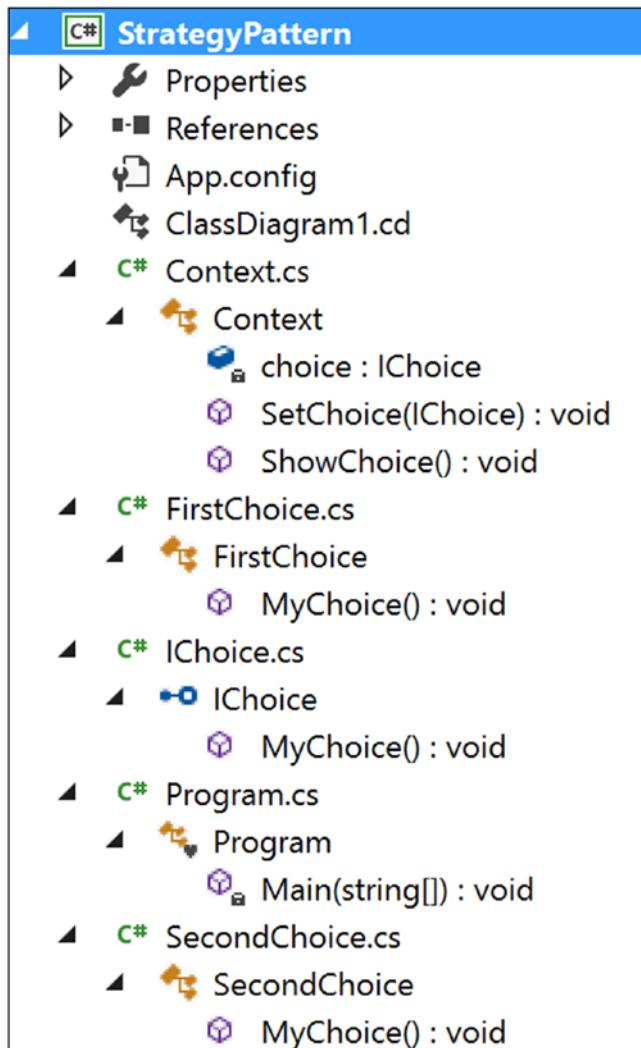


Figure 15-3. Solution Explorer View

Implementation

Here's the implementation:

```
// IChoice.cs

using System;

namespace StrategyPattern
{
    public interface IChoice
    {
        void MyChoice();
    }
}

// FirstChoice.cs

using System;

namespace StrategyPattern
{
    public class FirstChoice:IChoice
    {
        public void MyChoice()
        {
            Console.WriteLine("Traveling to Japan");
        }
    }
}

// SecondChoice.cs

using System;

namespace StrategyPattern
{
    public class SecondChoice:IChoice
    {
        public void MyChoice()
```

CHAPTER 15 STRATEGY (POLICY) PATTERN

```
{  
    Console.WriteLine("Traveling to India");  
}  
}  
}  
  
// Context.cs  
using System;  
  
namespace StrategyPattern  
{  
    public class Context  
    {  
        IChoice choice;  
        /*It's our choice. We prefer to use a setter method instead of  
         *using a constructor. We can call this method whenever we want to  
         *change the "choice behavior" on the fly*/  
        public void SetChoice(IChoice choice)  
        {  
            this.choice = choice;  
        }  
        /* This method will help us to delegate the particular  
         * object's choice behavior/characteristic*/  
        public void ShowChoice()  
        {  
            choice.SetChoice();  
        }  
    }  
}  
  
// Program.cs  
using System;  
  
namespace StrategyPattern  
{  
    class Program  
    {
```

```

static void Main(string[] args)
{
    Console.WriteLine("/**Strategy Pattern Demo**\n");
    IChoice ic = null;
    Context ctxt = new Context();
    //For simplicity, we are considering 2 user inputs only.
    for (int i = 1; i <= 2; i++)
    {
        Console.WriteLine("\nEnter ur choice(1 or 2)");
        string c = Console.ReadLine();
        if (c.Equals("1"))
        {
            ic = new FirstChoice();
        }
        else
        {
            ic = new SecondChoice();
        }
        ctxt.SetChoice(ic);
        ctxt.ShowChoice();
    }
    Console.ReadKey();
}
}
}

```

Output

Here's the output:

```
***Strategy Pattern Demo***
```

```
Enter ur choice(1 or 2)
```

```
2
```

```
Traveling to India
```

Enter ur choice(1 or 2)

1

Traveling to Japan

Q&A Session

1. **Why are you complicating everything? Why do you need the Context class? You could simply use the inheritance mechanism and proceed.**

Answer:

Let's examine what happens if you use the concept of inheritance only. Let's assume that you have the following structure:

```
public interface IChoice
{
    void MyChoice();
}

public class FirstChoice : IChoice
{
    public void MyChoice()
    {
        Console.WriteLine("Traveling to Japan");
    }
}

public class SecondChoice : IChoice
{
    public void MyChoice()
    {
        Console.WriteLine("Traveling to India");
    }
}
```

Now suppose you need to introduce another class in your system that has a special choice. How should you proceed? Say you place a method called `MySpecialChoice()` in the interface like this:

```
public interface IChoice
{
    void MyChoice();
    void MySpecialChoice();
}
```

All the existing classes need to implement this additional method. But it is not over yet. Further assume that there is a constraint on the class `FirstChoice`. It cannot have any special choices, so you will encounter a deadlock situation. If you implement this special method, you are violating the constraint, and if you do not implement it, the system architecture will be broken because as per the language construct, you need to implement all the interface methods. (Or, you need to mark the class with the `abstract` keyword, but at the same time, remember that you cannot create an instance from an abstract class.)

- 2. You could create a separate interface, say `ISpecialChoice`, and place the `MySpecialChoice()` method in that interface. Now any class that wants to get the method can implement that interface also.**

Answer:

Yes, you can do that, but this is what the code will look like:

```
public interface IChoice
{
    void MyChoice();
    //void MySpecialChoice();
}

public interface ISpecialChoice
{
    void MySpecialChoice();
}
```

```

public class FirstChoice : IChoice
{
    public void MyChoice()
    {
        Console.WriteLine("Traveling to Japan");
    }
}
public class SecondChoice : IChoice, ISpecialChoice
{
    public void MyChoice()
    {
        Console.WriteLine("Traveling to India");
    }
    public void MySpecialChoice()
    {
        Console.WriteLine("This is my special choice");
    }
}

```

If you have many classes in the system and you are coding in this way, it will be difficult to maintain the code because if the characteristic/behavior of the special choice changes in the future, you will need to track down all the classes that implement the behavior.

3. Can you add some special characteristic to the default behavior at runtime in your program?

Answer:

Yes, you can. Let's modify the program little bit. Rename the method `MyChoice()` to `SetChoice()` and add some default behavior in the choice hierarchy through the default constructor as follows (the changes are highlighted in bold):

```

public interface IChoice
{
    void SetChoice();
}

```

```

public class FirstChoice:IChoice
{
    public FirstChoice()
    {
        Console.WriteLine(" I do not want to travel");
    }
    public void SetChoice()
    {
        Console.WriteLine("Now I am traveling to Japan");
    }
}
public class SecondChoice:IChoice
{
    public SecondChoice()
    {
        Console.WriteLine(" I am arranging money to travel");
    }
    public void SetChoice()
    {
        Console.WriteLine("Now I am traveling to India");
    }
}

```

If you execute this modified program, you will get the following output:

Strategy Pattern Demo

Enter ur choice(1 or 2)

2

I am arranging money to travel

Now I am traveling to India

Enter ur choice(1 or 2)

1

I do not want to travel

Now I am traveling to Japan

Notice that the default behaviors are modified dynamically by the SetChoice() method.

4. Can you use an abstract class instead of an interface?

Answer:

Yes, and it is suitable also in some cases where you may want to put some common behavior in the abstract class. I discussed this in detail in the “Q&A Session” section of the Builder pattern in Chapter 3.

5. What are the key advantages of using a Strategy design pattern?

Answer:

- This design pattern makes your classes independent from algorithms. Here a class delegates the algorithms to the Strategy object (that encapsulates the algorithm) dynamically at runtime. So, the choice of algorithms is not bound at compile time.
- It's easier to maintain your codebase.
- It's easily extendable. (Refer to question 2 and 3 and the corresponding answers.)

6. What are the key challenges associated with a Strategy design pattern?

Answer:

- The addition of context classes causes more objects in your application.
- Users of the application must be aware of different strategies; otherwise, the outputs may surprise them.

CHAPTER 16

Template Method Pattern

This chapter covers the Template Method pattern.

GoF Definition

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. The Template Method pattern lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

Concept

With the Template Method pattern, you define the minimum or essential structure of an algorithm. Then you defer some responsibility to the subclasses. The key idea is that you can redefine certain steps of an algorithm, but those changes will not impact the core architecture.

This design pattern is useful if you have to implement a multistep algorithm and you want to allow customization through subclasses.

Real-Life Example

Suppose you want to make a pizza. You use the basic mechanism each time you make a pizza, but you include the final materials based on your customer's choices. For example, the customer may want a vegetarian pizza or a meat pizza. A customer can even opt for different toppings such as bacon, onions, extra cheese, mushrooms, and so on.

Computer World Example

Suppose you have been hired to design an online engineering degree course. You know that, in general, the first semester of the course will be the same for all courses. For subsequent semesters, you need to add new papers or subjects to the application based on the course opted by a student. The Template Method pattern makes sense when you want to avoid duplicate code in your application but allow subclasses to change some specific details of the base class workflow to bring varying behavior to the application. (However, you may not want to override the base methods entirely to make radical changes in the subclasses. In this way, the pattern differs from simple polymorphism.)

Illustration

Assume that each engineering student needs to pass mathematics and also demonstrate soft skills (such as communication skills, people management skills, and so on) in their initial semesters to obtain their degrees. Later you will add some special papers to their courses based on their chosen paths (computer science or electronics).

Class Diagram

Figure 16-1 shows the class diagram.

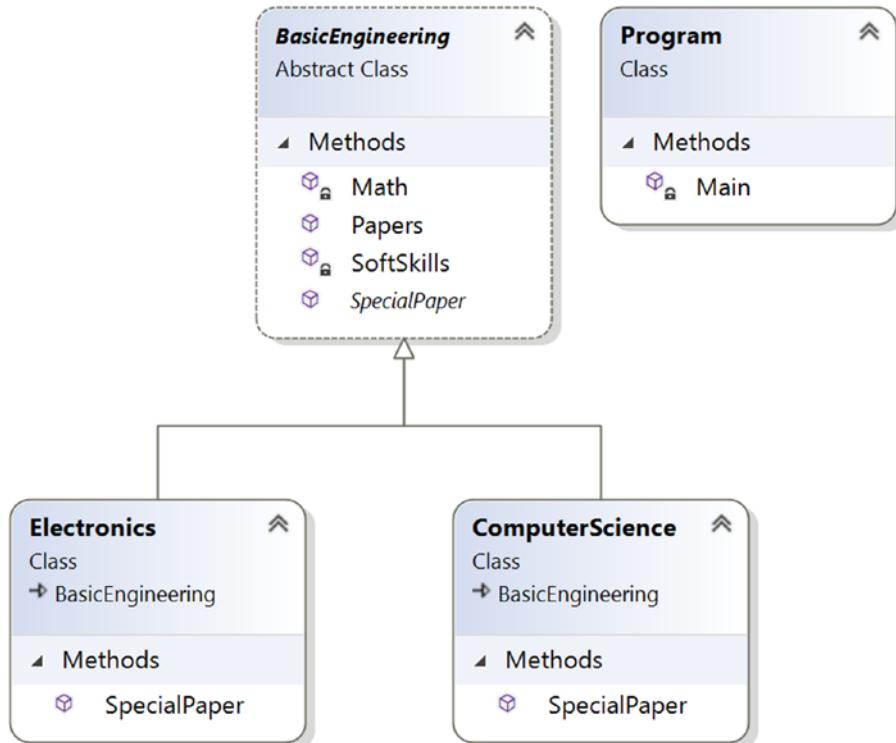


Figure 16-1. Class diagram

Solution Explorer View

Figure 16-2 shows the high-level structure of the parts of the program.

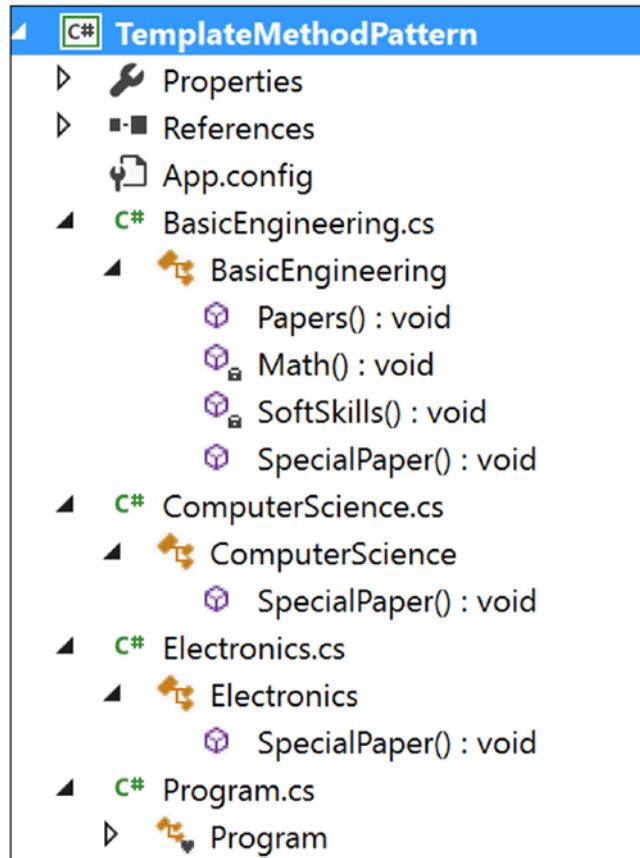


Figure 16-2. Solution Explorer View

Implementation

Here's the implementation:

```
// BasicEngineering.cs
using System;

namespace TemplateMethodPattern
{
    public abstract class BasicEngineering
    {
```

```
public void Papers()
{
    //Common Papers:
    Math();
    SoftSkills();
    //Specialized Paper:
    SpecialPaper();
}
private void Math()
{
    Console.WriteLine("Mathematics");
}
private void SoftSkills()
{
    Console.WriteLine("SoftSkills");
}
public abstract void SpecialPaper();
}

//ComputerScience.cs
using System;

namespace TemplateMethodPattern
{
    public class ComputerScience:BasicEngineering
    {
        public override void SpecialPaper()
        {
            Console.WriteLine("Object-Oriented Programming");
        }
    }
}
```

CHAPTER 16 TEMPLATE METHOD PATTERN

```
//Electronics.cs

using System;

namespace TemplateMethodPattern
{
    public class Electronics:BasicEngineering
    {
        public override void SpecialPaper()
        {
            Console.WriteLine("Digital Logic and Circuit Theory");
        }
    }
}

// Program.cs
using System;

namespace TemplateMethodPattern
{
    class Program
    {
        static void Main(string[] args)
        {

Console.WriteLine("***Template Method Pattern Demo***\n");
        BasicEngineering bs = new ComputerScience();
        Console.WriteLine("Computer Sc Papers:");
        bs.Papers();
        Console.WriteLine();
        bs = new Electronics();
        Console.WriteLine("Electronics Papers:");
        bs.Papers();
        Console.ReadLine();
    }
}
}
```

Output

Here's the output:

```
***Template Method Pattern Demo***
```

Computer Sc Papers:

Mathematics

SoftSkills

Object-Oriented Programming

Electronics Papers:

Mathematics

SoftSkills

Digital Logic and Circuit Theory

Q&A Session

1. In this pattern, subclasses can simply redefine the methods based on their needs. Is this understanding correct?

Answer:

Yes.

2. In the abstract class **BasicEngineering**, only one method is abstract, and the other two methods are concrete methods. What is the reason behind this?

Answer:

This is a simple example with only three methods, and you want the subclasses to override only the `SpecialPaper()` method here. Other methods are common to both courses, and they do not need to be overridden by the subclasses.

3. Suppose you want to add some more methods in the **BasicEngineering** class, but you want to work on those methods if and only if your child classes need them. Otherwise, you will ignore them. This type of situation is common in some PhD programs where some courses are mandatory, but if a student has certain qualifications, the student may not need to attend the lectures for those subjects. Can you design this kind of situation with the Template Method pattern?

Answer:

Yes, you can. Basically, you want to use a hook, which is a method that can help you to control some kind of flow in an algorithm.

To show an example of this kind of design, I am adding one more method in **BasicEngineering** called **AdditionalPapers()**. Let's assume that computer science students need to complete this mandatory course, but electronics students can opt out. Let's go through the program and output.

Modified Implementation

```
// BasicEngineering.cs [Changes are in bold letters]
using System;

namespace TemplateMethodPatternQAs
{
    public abstract class BasicEngineering
    {
        public void Papers()
        {
            //Common Papers:
            Math();
            SoftSkills();
            if (IsAdditionalPapersNeeded())
            {
                AdditionalPapers();
            }
        }
    }
}
```

```
//Specialized Paper:  
    SpecialPaper();  
}  
private void Math()  
{  
    Console.WriteLine("Mathematics");  
}  
private void SoftSkills()  
{  
    Console.WriteLine("SoftSkills");  
}  
private void AdditionalPapers()  
{  
    Console.WriteLine("AdditionalPapers are needed in this stream.");  
}  
public abstract void SpecialPaper();  
//A hook method-Additional Papers not needed.  
public virtual bool IsAdditionalPapersNeeded()  
{  
    return true;  
}  
}  
}  
  
//ComputerScience.cs [No Change]  
  
using System;  
  
namespace TemplateMethodPatternQAs  
{  
    public class ComputerScience:BasicEngineering  
    {  
        public override void SpecialPaper()  
        {  
            Console.WriteLine("Object-Oriented Programming");  
        }  
    }  
}
```

```

//Not tested the hook method:
//Additional papers are needed
}

}

//Electronics.cs [Changes are in bold letters]
using System;

namespace TemplateMethodPatternQAs
{
    public class Electronics:BasicEngineering
    {
        public override void SpecialPaper()
        {
            Console.WriteLine("Digital Logic and Circuit Theory");
        }

//Using the hook method
        public override bool IsAdditionalPapersNeeded()
        {
            return false;
        }
    }
}

//Program.cs [No Change]
using System;

namespace TemplateMethodPatternQAs
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("***Template Method Pattern QAs***\n");
            BasicEngineering bs = new ComputerScience();
            Console.WriteLine("Computer Sc Papers:");
            bs.Papers();
            Console.WriteLine();
            bs = new Electronics();
        }
    }
}

```

```

        Console.WriteLine("Electronics Papers:");
        bs.Papers();
        Console.ReadLine();
    }
}
}

```

Modified Output

Here's the modified output:

Template Method Pattern QAs

```

Computer Sc  Papers:
Mathematics
SoftSkills
AdditionalPapers are needed in this stream.
Object Oriented Programming

Electronics Papers:
Mathematics
SoftSkills
Digital Logic and Circuit Theory

```

Note You may prefer an alternative approach. For example, you could override the method in `Electronics.cs`, and then you could make the method body empty. But this approach does not work well if you compare it to the previous approach.

4. It looks like this pattern is similar to the Builder pattern. Is this understanding correct?

Answer:

No. Don't forget the core intent; the Template Method pattern is a behavioral design pattern, and Builder is a creational design pattern. In the Builder pattern, the clients/customers are the boss. They can control the order of the algorithm. In the Template

Method pattern, you are the boss. You put your code in a central location (for example, the abstract class `BasicEngineering.cs` in this example), and you have absolute control over the flow of the execution. You can alter your template as per your needs, and other participants need to follow you.

5. What are the key advantages of using a Template Method design pattern?

Answer:

- You can control the flow of the algorithms. Clients cannot change them.
- Common operations will be in a centralized location. For example, in an abstract class, the subclasses can redefine only the varying parts so that you can avoid redundant code.

6. What are the key challenges associated with a Template Method design pattern?

Answer:

- The client code cannot direct the sequence of steps. If you want that type of functionality, use the Builder pattern.
- A subclass can override a method defined in the parent class (in other words, hiding the original definition in the parent class), which can go against the Liskov substitution principle that basically says that if S is a subtype of T, then objects of type T can be replaced with objects of type S. You can learn more here: https://en.wikipedia.org/wiki/Liskov_substitution_principle.
- Having more subclasses means more scattered code, which is difficult to maintain.

CHAPTER 17

Command Pattern

This chapter covers the Command pattern.

GoF Definition

Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

Concept

With this pattern, you encapsulate a method invocation process. In general, four terms are important here: *invoker*, *client*, *command*, and *receiver*. A *command* object can invoke a method of the receiver in a way that is specific to that receiver's class. The *receiver* then starts processing the job. A command object is separately passed to the *invoker* object to invoke the command. The *client* object holds the invoker object and the command objects. The client only makes the decision (which commands to execute) and then passes the command to the invoker object (for the execution).

Real-Life Example

When you are drawing a picture, you may need to redraw(undo) some parts of it to make it better.

Computer World Example

In general, you can observe this pattern in the menu system of an editor or integrated development environment (IDE). For example, you can use the Command pattern to support undos, multiple undos, or similar operations in a software application.

Microsoft used this pattern in Windows Presentation Foundation (WPF). The article at <https://visualstudiomagazine.com/articles/2012/04/10/command-pattern-in-.net.aspx> describes it in detail.

The command pattern is well suited for handling GUI interactions. It works so well that Microsoft has integrated it tightly into the Windows Presentation Foundation (WPF) stack. The most important piece is the ICommand interface from the System.Windows.Input namespace. Any class that implements the ICommand interface can be used to handle a keyboard or mouse event through the common WPF controls. This linking can be done either in XAML or in a code-behind."

In addition, if you are familiar with Java and Swing, you will notice that Action is also a command object.

Illustration

In this example, I am using similar class names to the concept described earlier.

Class Diagram

Figure 17-1 shows the class diagram.

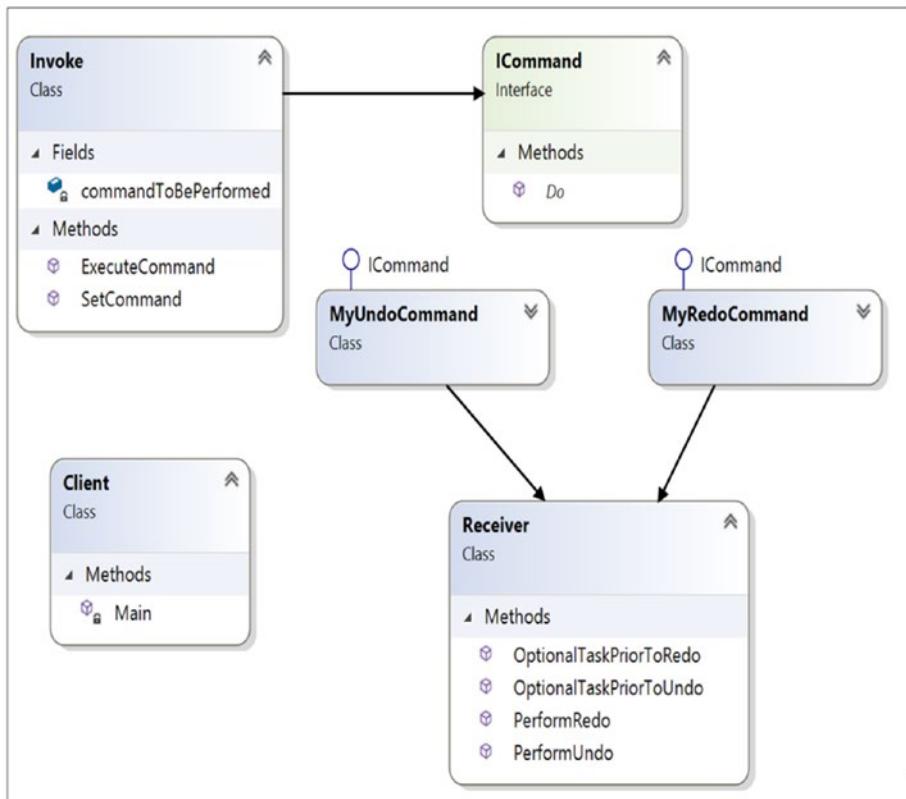


Figure 17-1. Class diagram

Directed Graph Document

Figure 17-2 shows the directed graph document.

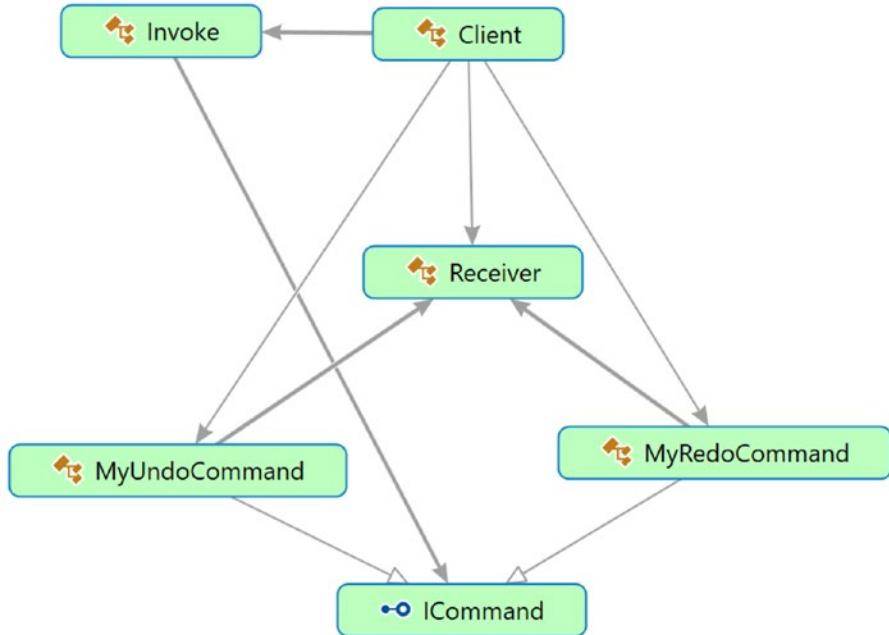


Figure 17-2. Directed Graph Document

Solution Explorer View

Figure 17-3 shows the high-level structure of the parts of the program.

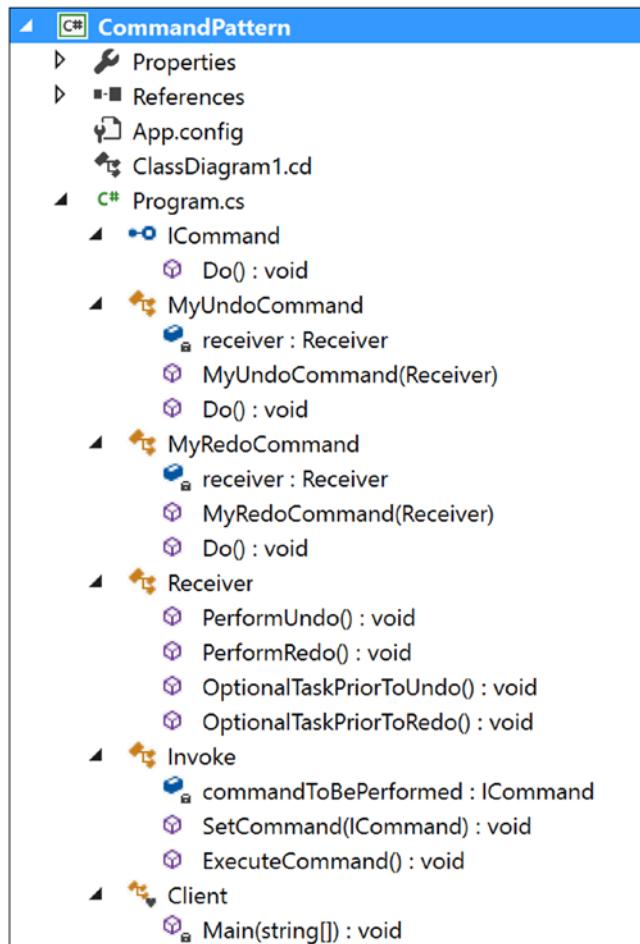


Figure 17-3. Solution Explorer View

Implementation

Here's the implementation:

```
using System;

namespace CommandPattern
{
    public interface ICommand
    {
        void Do();
    }

    public class MyUndoCommand: ICommand
    {
        private Receiver receiver;
        public MyUndoCommand(Receiver recv)
        {
            receiver=recv;
        }
        public void Do()
        {
            //Perform any optional task prior to UnDo
            receiver.OptionalTaskPriorToUndo();
            //Call UnDo in receiver now
            receiver.PerformUndo();
        }
    }
    public class MyRedoCommand : ICommand
    {
        private Receiver receiver;
        public MyRedoCommand(Receiver recv)
        {
            receiver=recv;
        }
    }
}
```

```
public void Do()
{
    //Perform any optional task prior to ReDo
    receiver.OptionalTaskPriorToRedo();
    //Call ReDo in receiver now
    receiver.PerformRedo();
}

//Receiver Class
public class Receiver
{
    public void PerformUndo()
    {
        Console.WriteLine("Executing-MyUndoCommand");
    }

    public void PerformRedo()
    {
        Console.WriteLine("Executing-MyRedoCommand");
    }

    //Optional method-If you want to perform any prior tasks before
    //Undo.
    public void OptionalTaskPriorToUndo()
    {
        Console.WriteLine("Executing-Optional Tasks prior to execute
            undo command");
    }

    //Optional method-If you want to perform any prior tasks before
    //Redo.
    public void OptionalTaskPriorToRedo()
    {
        Console.WriteLine("Executing-Optional Tasks prior to execute
            redo command");
    }
}
```

CHAPTER 17 COMMAND PATTERN

```
//Invoker class
public class Invoke
{
    ICommand commandToBePerformed;
    public void SetCommand(ICommand command)
    {
        this.commandToBePerformed = command;
    }
    public void ExecuteCommand()
    {
        commandToBePerformed.Do();
    }
}

class Client
{
    static void Main(string[] args)
    {
        Console.WriteLine("/**Command Pattern Demo**\n");
        /*Client holds both the Invoker and Command Objects*/
        Invoke invoker = new Invoke ();
        Receiver intendedreceiver = new Receiver();

        MyUndoCommand undoCmd = new MyUndoCommand(intendedreceiver);
        invoker.SetCommand(undoCmd);
        invoker.ExecuteCommand();

        MyRedoCommand redoCmd = new MyRedoCommand(intendedreceiver);
        invoker.SetCommand(redoCmd);
        invoker.ExecuteCommand();
        Console.ReadKey();
    }
}
```

Output

Here's the output:

Command Pattern Demo

Executing-Optional Tasks prior to execute undo command

Executing-MyUndoCommand

Executing-Optional Tasks prior to execute redo command

Executing-MyRedoCommand

Q&A Session

1. In this example, you are dealing with a single receiver only.

How can you deal with multiple receivers?

Also, the GoF definition says that this pattern supports undoable operations. Can you experiment with a true undo operation using this pattern?

Answer:

Consider the following program. The key characteristics of this program are as follows:

- The program has two receivers (Receiver1 and Receiver2). Each of them implements the IReceiver interface methods.
- The program introduces a postprocessing task, which is similar to the preprocessing task described earlier. (In many cases, you may need to implement both a preprocessing task and a postprocessing task. For example, if you want to get data from a particular server, you may need to connect to the server first, then process the data, and finally close the connecting gracefully.)

- To show a simple illustration with undo operations, I am instantiating the receivers with an integer value. A Receiver1 object is instantiated with the value 10, and a Receiver2 object is instantiated with the value 75. Each of the receivers can see the impact of the additional operations. As the names suggest, each of them can keep adding two with the instantiated value. For simplicity, I have put check marks on the values 10 and 75. In other words, while processing an undo operation, if you see that the Receiver1 object's myNumber is 10, you will not go beyond that (because you started at 10). Similarly, a Receiver2 object cannot set a value less than 75.

Modified Class Diagram

Figure 17-4 shows the modified class diagram.



Figure 17-4. Modified class diagram

Modified Solution Explorer View

Figure 17-5 shows the modified Solution Explorer view.

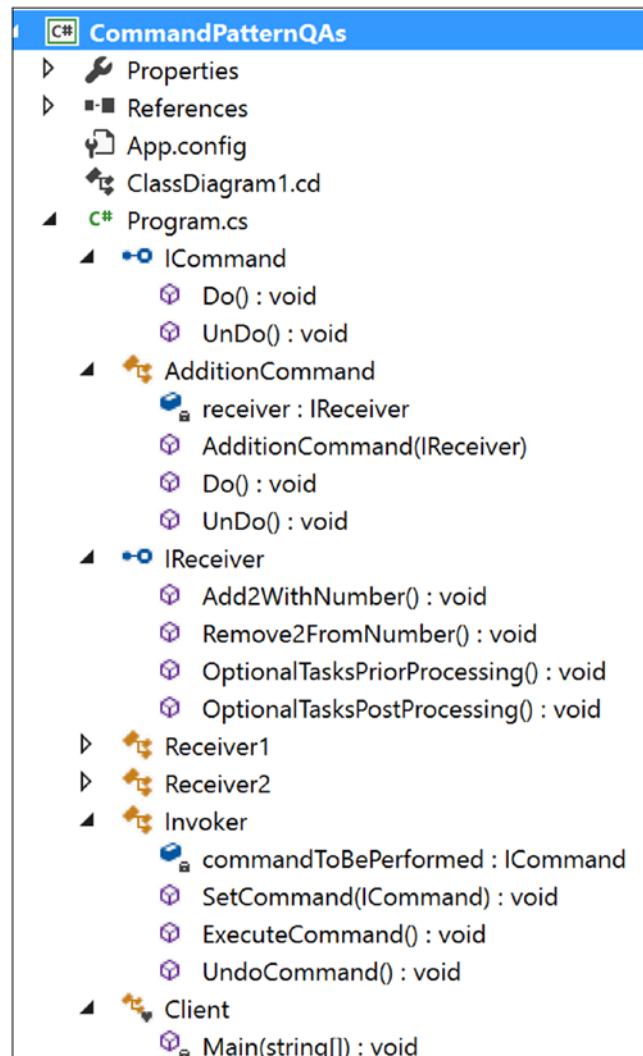


Figure 17-5. Modified Solution Explorer View

Modified Implementation

Here's the modified implementation:

```
using System;

namespace CommandPatternQAs
{
    public interface ICommand
    {
        void Do();
        void Undo();
    }

    public class AdditionCommand : ICommand
    {
        private IReceiver receiver;
        public AdditionCommand(IReceiver recv)
        {
            receiver = recv;
        }

        public void Do()
        {
            receiver.OptionalTasksPriorProcessing();
            receiver.Add2WithNumber();
            receiver.OptionalTasksPostProcessing();
        }

        public void Undo()
        {
            Console.WriteLine("Trying undoing addition...");
            receiver.Remove2FromNumber();
            Console.WriteLine("Undo request processed.");
        }
    }
}
```

```
//To deal with multiple receivers, we are using interfaces here
public interface IReceiver
{
    //It will add 2 with a number
    void Add2WithNumber();
    //It will subtract 2 from a number
    void Remove2FromNumber();
    //Optional methods
    //PreProcessing tasks
    void OptionalTasksPriorProcessing();
    //PostProcessing tasks
    void OptionalTasksPostProcessing();
}

//Receiver Class
public class Receiver1 : IReceiver
{
    int myNumber;
    public int MyNumber
    {
        get
        {
            return myNumber;
        }
        set
        {
            myNumber = value;
        }
    }
    public Receiver1()
    {
        myNumber = 10;
        Console.WriteLine("Receiver1 initialized with {0}", myNumber);
        Console.WriteLine("The objects of receiver1 cannot set beyond {0}", myNumber);
    }
}
```

```
//PreProcessing tasks
public void OptionalTasksPriorProcessing()
{
    Console.WriteLine("Receiver1.OptionalTaskPriorProcessing");
}

//PostProcessing tasks
public void OptionalTasksPostProcessing()
{
    Console.WriteLine("Receiver1.OptionalTaskPostProcessing\n");
}

public void Add2WithNumber()
{
    int presentNumber = this.MyNumber;
    this.MyNumber = this.MyNumber + 2;
    Console.WriteLine("{0}+2={1}", presentNumber, this.MyNumber);
}

public void Remove2FromNumber()
{
    int presentNumber = this.MyNumber;
    //We started with number 10. We'll not decrease further.
    if (presentNumber > 10)
    {
        this.MyNumber = this.MyNumber - 2;
        Console.WriteLine("{0}-2={1}", presentNumber, this.MyNumber);
    }
    else
    {
        Console.WriteLine("Nothing more to undo...");
    }
}

//Receiver Class
public class Receiver2 : IReceiver
{
    int myNumber;
```

```
public int MyNumber
{
    get
    {
        return myNumber;
    }
    set
    {
        myNumber = value;
    }
}
public Receiver2()
{
    myNumber = 75;
    Console.WriteLine("Receiver2 initialized with {0}", myNumber);
    Console.WriteLine("The objects of receiver2 cannot set beyond
{0}", myNumber);
}
//PreProcessing tasks
public void OptionalTasksPriorProcessing()
{
    Console.WriteLine("Receiver2.OptionalTaskPriorProcessing");
}
//PostProcessing tasks
public void OptionalTasksPostProcessing()
{
    Console.WriteLine("Receiver2.OptionalTaskPostProcessing");
}
public void Add2WithNumber()
{
    int presentNumber = this.MyNumber;
    this.MyNumber = this.MyNumber + 2;
    Console.WriteLine("{0}+2={1}", presentNumber, this.MyNumber);
}
```

CHAPTER 17 COMMAND PATTERN

```
public void Remove2FromNumber()
{
    int presentNumber = this.MyNumber;
    //We started with number 75. We'll not decrease further.
    if (presentNumber > 75)
    {
        this.MyNumber = this.MyNumber-2;
        Console.WriteLine("{0}-2={1}", presentNumber, this.MyNumber);
    }
    else
    {
        Console.WriteLine("Nothing more to undo...");
    }
}

//Invoker class
public class Invoker
{
    ICommand commandToBePerformed;
    public void SetCommand(ICommand command)
    {
        this.commandToBePerformed = command;
    }
    public void ExecuteCommand()
    {
        commandToBePerformed.Do();
    }
    public void UndoCommand()
    {
        commandToBePerformed.UnDo();
    }
}
```

```
class Client
{
    static void Main(string[] args)
    {
        Console.WriteLine("/**Command Pattern Q&As**");
        Console.WriteLine("/**A simple demo with undo supported
operations**\n");
        /*Client holds both the Invoker and Command Objects*/
        Invoker invoker = new Invoker();
        //Testing receiver-Receiver1
        IReceiver intendedreceiver = new Receiver1();
        ICommand currentCmd = new AdditionCommand(intendedreceiver);
        invoker.SetCommand(currentCmd);
        //Executed the command 2 times
        invoker.ExecuteCommand();
        invoker.ExecuteCommand();
        //Trying to undo 3 times
        invoker.UndoCommand();
        invoker.UndoCommand();
        invoker.UndoCommand();

        Console.WriteLine("\nTesting receiver-Receiver2");
        intendedreceiver = new Receiver2();
        currentCmd = new AdditionCommand(intendedreceiver);
        invoker.SetCommand(currentCmd);
        //Executed the command 1 time
        invoker.ExecuteCommand();
        //Trying to undo 2 times
        invoker.UndoCommand();
        invoker.UndoCommand();
        Console.ReadKey();
    }
}
```

Modified Output

Here's the modified output:

Command Pattern Q&As

A simple demo with undo supported operations

Receiver1 initialized with 10

The objects of receiver1 cannot set beyond 10

Receiver1.OptionalTaskPriorProcessing

10+2=12

Receiver1.OptionalTaskPostProcessing

Receiver1.OptionalTaskPriorProcessing

12+2=14

Receiver1.OptionalTaskPostProcessing

Trying undoing addition...

14-2=12

Undo request processed.

Trying undoing addition...

12-2=10

Undo request processed.

Trying undoing addition...

Nothing more to undo...

Undo request processed.

Testing receiver-Receiver2

Receiver2 initialized with 75

The objects of receiver2 cannot set beyond 75

Receiver2.OptionalTaskPriorProcessing

75+2=77

Receiver2.OptionalTaskPostProcessing

Trying undoing addition...

77-2=75

Undo request processed.

Trying undoing addition...

Nothing more to undo...

Undo request processed.

2. In this modified program, I don't see much difference between the receivers. Was that intentional?

Answer:

Yes, for simplicity, the initialized values were different, 10 and 75. You can simply replace this concept with different states/methods as per your needs among the receivers.

3. Why do you need the invoker?

Answer:

Most of the time, programmers try to encapsulate data and the corresponding methods in object-oriented programming. But you will find that in this pattern you are trying to encapsulate command objects. In other words, you are implementing encapsulation from a different perspective.

This approach makes sense when you need to deal with a complex set of commands.

Let's review terms again. You create command objects that you shoot to some receivers to access them. But you execute those commands through an invoker that calls the methods of the command objects (for example, ExecuteCommand in this example). For a simple case, this invoker class is not mandatory. For example, consider a case in which a command object has only one method to execute and you are trying to dispense with the invoker to invoke the method. But invokers may play an important role when you want to keep track of multiple commands in a log file (or in a queue).

4. Why would you want to keep track of these logs?

Answer:

You may want to create undo or redo operations.

5. What are the key advantages associated with the Command pattern?

Answer:

- Requests for creation and the ultimate execution are decoupled. Clients may not know how an invoker is performing the operations.
- You can create macros (sequences of commands).
- New commands can be added without affecting the existing system.
- Most importantly, you can support the much-needed undo/redo operations.

6. What are the challenges associated with the Command pattern?

Answer:

To support more commands, you need to create more classes. So, maintenance can be difficult as time goes on.

CHAPTER 18

Iterator Pattern

This chapter covers the Iterator pattern.

GoF Definition

Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Concept

Iterators are generally used to traverse a container (which is basically an object) to access its elements, but you do not need to deal with the element's internal details. You will frequently use the concepts of iterators when you want to traverse different kinds of collection objects in a standard and uniform way.

- This concept is used frequently to traverse the nodes of a tree-like structure. So, in many scenarios, you may notice the use of the Iterator pattern with the Composite pattern.
- C# has its own iterators that were introduced in Visual Studio 2005. The foreach statement is frequently used in this context. To learn more about these built-in functionalities, you can refer to this web page:

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/iterators>

- If you are familiar with Java, you may have already used Java's built-in Iterator interface: `java.util.Iterator`.

Real-Life Example

Suppose there are two companies, Company A and Company B. Company A stores its employee records (say each employee's name, address, salary details, and so on) in a linked list data structure, and Company B stores its employee data in an array. One day the two companies decide to merge. The Iterator pattern becomes handy in such a situation because you do not need to write any code from scratch. In a situation like this, you can have a common interface through which you can access the data for both companies. So, you can simply call those methods without rewriting the code.

Computer World Example

Similarly suppose, in a college, the arts department is using an array data structure to maintain its student records, and the science department is using a linked list data structure to keep its student records. The administrative department does not care about the different data structures. It is simply interested in getting the data from each department and wants to access the data in a uniform way.

Note In C#, the `LinkedList<T>` class is defined in the `System.Collections.Generic` namespace.

Illustration

This example has many parts, and I have created related folders for each of them. To show an implementation of this design pattern, I'll use the computer world example described earlier. Before you start, refer to the Solution Explorer View and Directed Graph Document.

In this implementation, assume that science subjects are stored in a linked list and arts subjects are stored in an array. You want to print the curriculum using the iterators. Here `IIterator` is the common interface that consists of four methods: `First()`, `Next()`, `IsDone()`, and `CurrentItem()`. So, these methods are implemented in each of the classes `ScienceIterator` and `ArtsIterator` as per your needs. (For example, notice that the `CurrentItem()` method is defined differently in the `ScienceIterator` and `ArtIterator`

classes.) In addition, to print the curriculum, I have used only two of these methods, `IsDone()` and `Next()`. If you want, you can experiment with the remaining methods, `First()` and `currentItem()`. I have provided some sample implementations for these methods so that you can experiment with them.

Class Diagram

Figure 18-1 shows the class diagram.

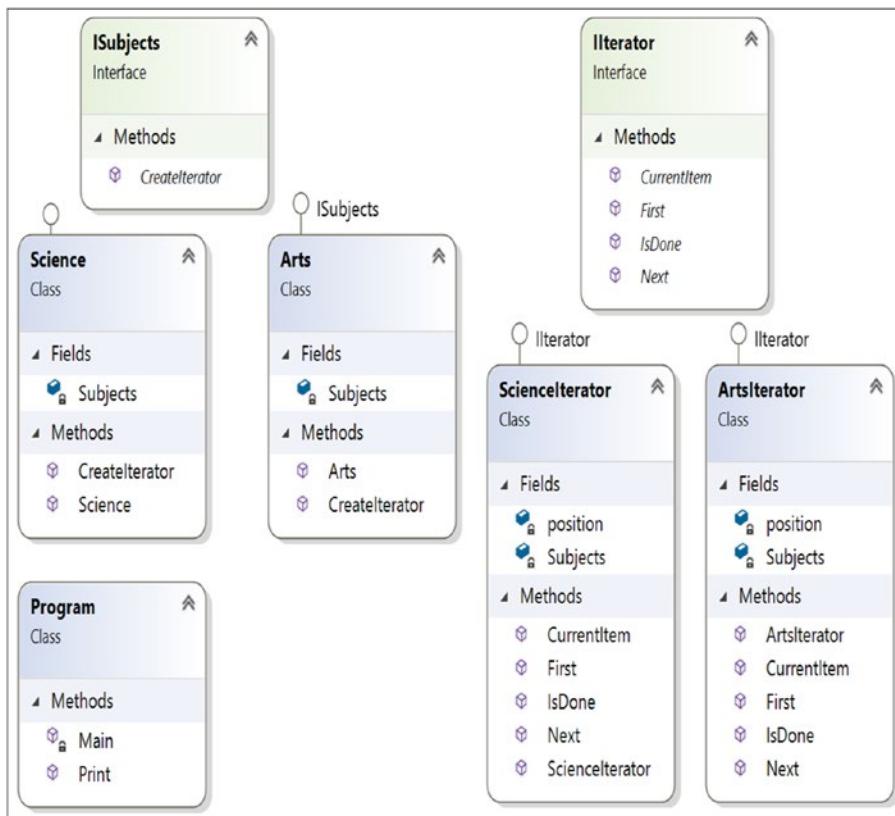


Figure 18-1. Class diagram

Directed Graph Document

Figure 18-2 shows the directed graph document.

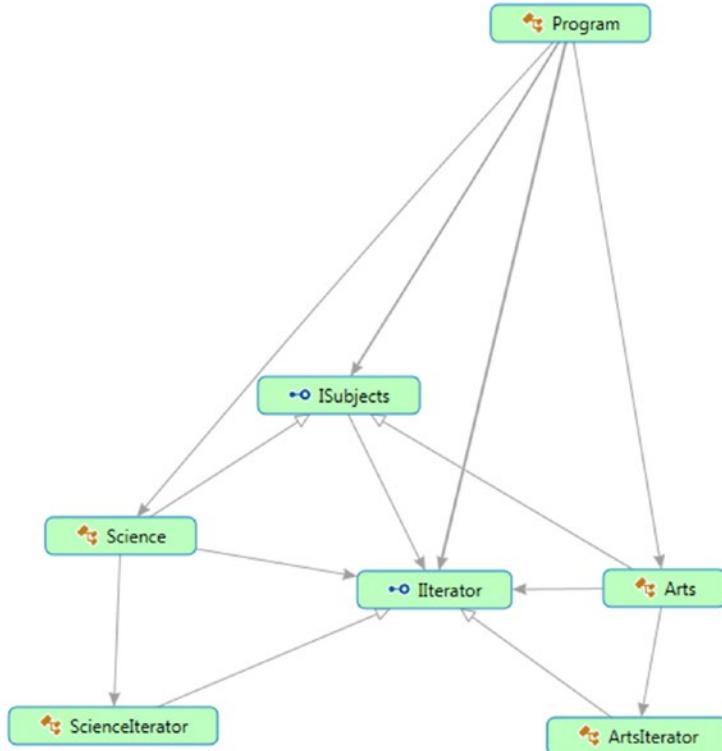


Figure 18-2. Directed Graph Document

Solution Explorer View

Figure 18-3 shows the high-level structure of the parts of the program.

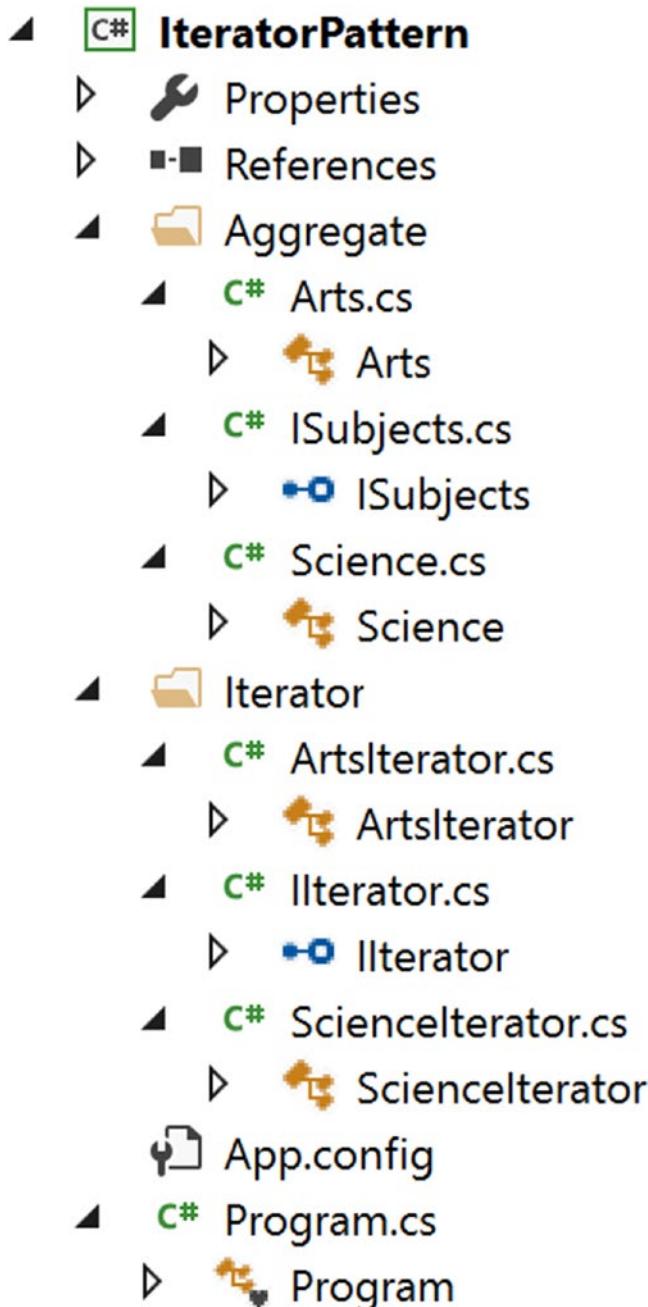


Figure 18-3. Solution Explorer View

Implementation

Here's the implementation:

```
// ISubjects.cs
using System;
using IteratorPattern.Iterator;

namespace IteratorPattern.Aggregate
{
    public interface ISubjects
    {
        IIIterator CreateIterator();
    }
}

// Science.cs
using System;
using IteratorPattern.Iterator;
using System.Collections.Generic;//For Linked List

namespace IteratorPattern.Aggregate
{
    public class Science:ISubjects
    {
        private LinkedList<string> Subjects;

        public Science()
        {
            Subjects = new LinkedList<string>();
            Subjects.AddFirst("Maths");
            Subjects.AddFirst("Comp. Sc.");
            Subjects.AddFirst("Physics");
        }

        public IIIterator CreateIterator()
        {
            return new ScienceIterator(Subjects);
        }
    }
}
```

```
        }
    }
}

//Arts.cs
using System;
using IteratorPattern.Iterator;

namespace IteratorPattern.Aggregate
{
    public class Arts:ISubjects
    {
        private string[] Subjects;

        public Arts()
        {
            Subjects = new[] {"Bengali", "English"};
        }

        public IIIterator CreateIterator()
        {
            return new ArtsIterator(Subjects);
        }
    }
}

using System;

namespace IteratorPattern.Iterator
{
    public interface IIIterator
    {
        void First();//Reset to first element
        string Next();//Get next element
        bool IsDone();//End of collection check
        string CurrentItem();//Retrieve Current Item
    }
}
```

CHAPTER 18 ITERATOR PATTERN

```
// ScienceIterator.cs
using System;
using System.Collections.Generic;
using System.Linq; //For: Subjects.ElementAt(position++);

namespace IteratorPattern.Iterator
{
    public class ScienceIterator:IIterator
    {
        private LinkedList<string> Subjects;
        private int position;

        public ScienceIterator(LinkedList<string> subjects)
        {
            this.Subjects = subjects;
            position = 0;
        }

        public void First()
        {
            position = 0;
        }

        public string Next()
        {
            return Subjects.ElementAt(position++);
        }

        public bool IsDone()
        {
            if (position < Subjects.Count)
            {
                return false;
            }
            else
            {
                return true;
            }
        }
    }
}
```

```
public string CurrentItem()
{
    return Subjects.ElementAt(position);
}
}

// ArtsIterator.cs
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace IteratorPattern.Iterator
{
    public class ArtsIterator:IIterator
    {
        private string[] Subjects;
        private int position;
        public ArtsIterator(string[] subjects)
        {
            this.Subjects = subjects;
            position = 0;
        }
        public void First()
        {
            position = 0;
        }

        public string Next()
        {
            return Subjects[position++];
        }

        public bool IsDone()
        {
            if( position < Subjects.Length)
```

```
        {
            return false;
        }
    else
    {
        return true;
    }
}

public string CurrentItem()
{
    return Subjects[position];
}
}

// Program.cs
using System;
using IteratorPattern.Aggregate;
using IteratorPattern.Iterator;

namespace IteratorPattern
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("***Iterator Pattern Demo***");
            ISubjects ScienceSubjects = new Science();
            ISubjects ArtsSubjects = new Arts();

            IIIterator IteratorForScience = ScienceSubjects.CreateIterator();
            IIIterator IteratorForArts = ArtsSubjects.CreateIterator();

            Console.WriteLine("\nScience subjects :");
            Print(IteratorForScience);

            Console.WriteLine("\nArts subjects :");
            Print(IteratorForArts);
        }
    }
}
```

```

        Console.ReadLine();
    }
    public static void Print(IIterator iterator)
    {
        while (!iterator.IsDone())
        {
            Console.WriteLine(iterator.Next());
        }
    }
}
}

```

Output

Here's the output:

Iterator Pattern Demo

Science subjects :

Physics

Comp. Sc.

Maths

Arts subjects :

Bengali

English

Q&A Session

1. What is the Iterator pattern used for?

Answer:

- You can traverse an object structure without knowing its internal details. As a result, if you have a collection of different subcollections (for example, your container is mixed with arrays,

lists, linked lists, and so on), you can still traverse the overall collection and deal with the elements in a universal way without knowing the internal details or differences among them.

- You can traverse a collection in different ways. If they are designed properly, multiple traversals are also possible in parallel.
- 2. **What are the key challenges associated with this pattern?**

Answer:

You must make sure that no accidental modification has taken place during the traversal procedure.

- 3. **But to deal with the challenge mentioned earlier, can you simply take a backup and then proceed?**

Answer:

Taking a backup and reexamining it later is a costly operation.

- 4. **In the Solution Explorer view, there is a folder called **Aggregate**. Is there any reason behind that naming?**

Answer:

An aggregate is used to define an interface to create an Iterator object. I have adopted the name from the GoF.

- 5. **Throughout the discussion, you have talked about collections. What is a collection?**

Answer:

When you manage (or create) a related group of objects, in C# you have following choices:

- You can consider arrays.
- You can consider collections.

Collections are preferred in many cases because they can grow or shrink dynamically. In some collections, you can even assign keys to objects so that you can retrieve them at a later stage more efficiently

with those keys. (For example, a dictionary is such a collection that is often used for fast lookups.) Lastly, a collection is a class, so before you add elements to it, you need to create instances. Here's an example:

```
LinkedList<string> Subjects = new LinkedList<string>();
Subjects.AddLast("Maths");
Subjects.AddLast("Comp. Sc.");
Subjects.AddLast("Physics");
```

In this example, instead of `AddFirst()` method, I have used `AddLast()` method for a variation. Both methods are in-built in C#. `AddLast()` method adds the node at the end of the `LinkedList<T>` whereas `AddFirst()` method adds the node at the beginning of `LinkedList<T>`.

- 6. In this implementation, you could simply consider using either of the science or arts subjects to demonstrate an implementation of an Iterator pattern and reduce the code size. Is this understanding correct?**

Answer:

Yes. But if you do not deal with at least two different data structures, you may not visualize the real power of the Iterator design pattern. So, I have kept both here.

Note In the Modified Illustration of Visitor pattern (Chapter 13), you have experimented a combination of 3 design patterns-Visitor pattern, Composite pattern and Iterator pattern (using C#'s “foreach” statements that are used to consume an iterator from client code).

CHAPTER 19

Memento Pattern

This chapter covers the Memento pattern.

GoF Definition

Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

Concept

The dictionary definition of the word *memento* is that it's a reminder of past events. When using this pattern, you can restore an object to its previous state. This pattern provides an object-oriented way to save the state of an object.

Real-Life Example

You can see a classic example of the Memento pattern in the states of a finite state machine. This pattern is a mathematical model, but one of its simplest applications is a turnstile. A turnstile has some rotating arms, which initially are locked. When you are allowed to go through it (say by putting some coins in), the locks will open, and the arms will be allowed to rotate. Once you pass through, the arms will return to a locked state.

Computer World Example

In a drawing application, you may need to revert to an older state.

Illustration

In this example, three objects are involved: a memento, an originator, and a caretaker. (These are common naming conventions, which is why I'm using them here.) The originator object has an internal state, and a client can set a state in it. The caretaker first asks the originator for a memento object because it wants to hold (or save) the current state of the originator object. The caretaker object also confirms the save operation by displaying a console message. Now suppose the client wants to revert to the previous state. Since the originator object's state is already changed, to roll back to the previous state, you need to get help from the caretaker object that saved the state earlier. So, the caretaker object now returns the memento object (with the previous state) to the originator. Also, the memento object itself is an opaque object (one that the caretaker is not allowed to make any changes on).

Class Diagram

Figure 19-1 shows the class diagram.

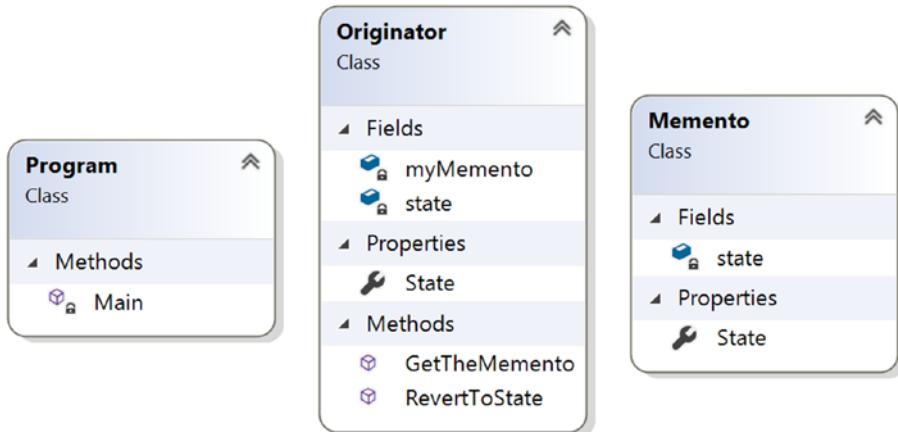


Figure 19-1. Class diagram

Directed Graph Document

Figure 19-2 shows the directed graph document.

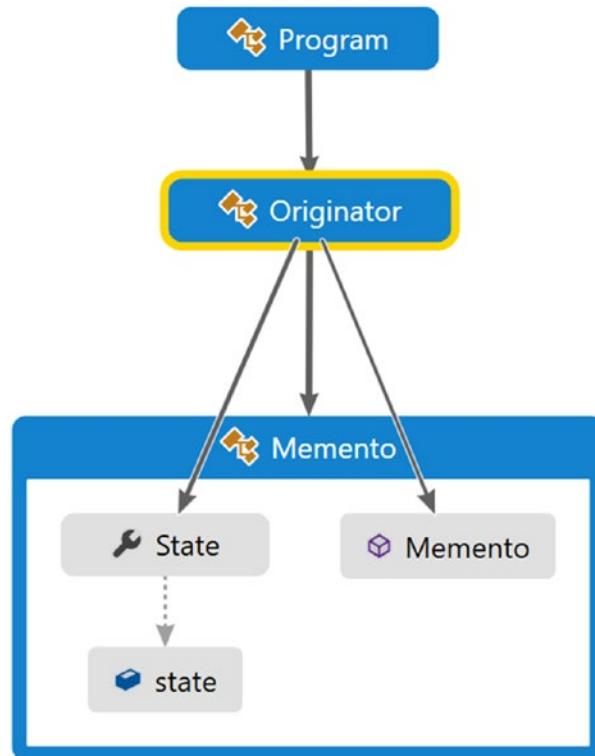


Figure 19-2. Directed Graph Document

Solution Explorer View

Figure 19-3 shows the high-level structure of the parts of the program.

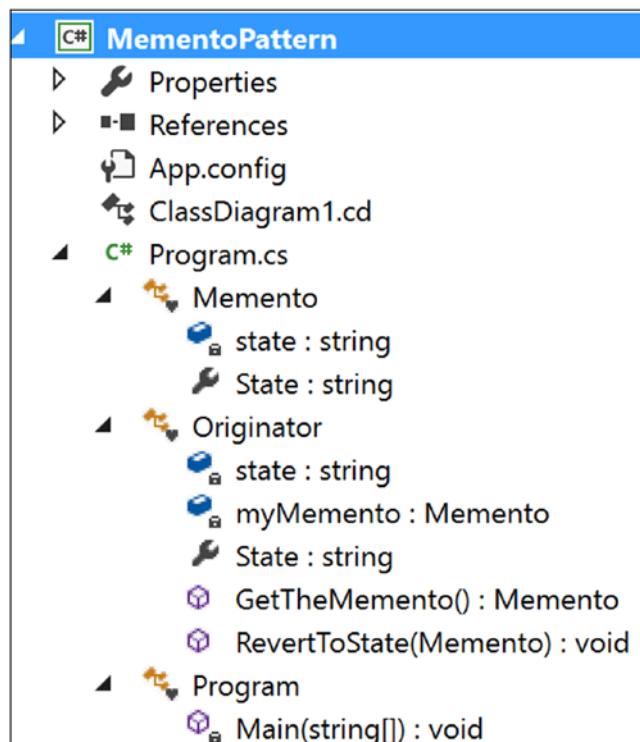


Figure 19-3. Solution Explorer View

Implementation

Here's the implementation:

```
using System;

namespace MementoPattern
{
    /// <summary>
    /// Memento class
    /// </summary>
```

```
class Memento
{
    private string state;
    //If we want the state to be same when a memento is created for the
    //first time, we could prefer a constructor (as shown below)
    //instead of using a setter method in this case.
    //public Memento(string state)
    //{
    //    this.state = state;
    //}
    public string State
    {
        get
        {
            return state;
        }
        set
        {
            state = value;
        }
    }
}

/// <summary>
/// Originator class
/// </summary>
/* Wikipedia notes( for your reference):
Make an object (originator) itself responsible for:
1.Saving its internal state to a(memento) object and
2.Restoring to a previous state from a(memento) object.
Only the originator that created a memento is allowed to access it.*/
```

CHAPTER 19 MEMENTO PATTERN

```
class Originator
{
    private string state;
    Memento myMemento;
    //public Originator()
    //{
    //    //this.State = "Initial state";
    //}
    public string State
    {
        get { return state; }
        set
        {
            state = value;
            Console.WriteLine("Current State : {0}", state);
        }
    }

    // Originator will supply the memento in respond to caretaker's
    //request
    public Memento GetTheMemento()
    {
        //Creating a memento with the current state
        myMemento = new Memento();
        myMemento.State = state;
        return myMemento;
    }

    // Back to old state (Restore)
    public void RevertToState(Memento previousMemento)
    {
        Console.WriteLine("Restoring to previous state...");
        this.state = previousMemento.State;
        Console.WriteLine("Current State : {0}", state);
    }
}
```

```
/// <summary>
/// The 'Caretaker' class.
/// Wikipedia notes( for your reference):
/// A client (caretaker) can request a memento from the originator
/// to save the internal state of the originator) and
/// pass a memento back to the originator (to restore to a previous state)
/// This enables to save and restore the internal state of an originator
/// without violating its encapsulation
/// </summary>
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("/**Memento Pattern Demo**\n");
        //Originator is initialized with a state
        Originator originatorObject = new Originator();
        Memento mementoObject;
        originatorObject.State = "Initial state";

        mementoObject = originatorObject.GetTheMemento();
        //Making a new state
        originatorObject.State="Intermediary state";

        // Restore to the previous state
        originatorObject.RevertToState(mementoObject);

        // Wait for user's input
        Console.ReadKey();
    }
}
```

Output

Here's the output:

```
***Memento Pattern Demo***

Current State : Initial state
Current State : Intermediary state
Restoring to previous state...
Current State : Initial state
```

Q&A Session

- With this pattern, you can restore the state to a particular point in time. But in a real-life scenario, you can have multiple restore points. How can you implement this concept using this design pattern?**

Answer:

You can use the generic `List<T>` collection in such a case.

Consider the following program.

The Originator class and the Memento class are the same as shown previously, so I am presenting the modified Caretaker class here. Since you are using the `List<T>` collection, do not forget to include the namespace `System.Collections.Generic`.

Modified Implementation

Here is the modified Caretaker class:

```
//Caretaker
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine(" ***Memento Pattern QAs ***");
```

```
Console.WriteLine("/**Demonstration-Caretaker is using multiple  
restore points**\n");  
//Originator is initialized with a state  
Originator originatorObject = new Originator();  
Memento mementoObject;  
IList<Memento> savedStates = new List<Memento>();  
Console.WriteLine("A new set of verification");  
//State-1  
originatorObject.State = "State-1";  
savedStates.Add(originatorObject.GetTheMemento());  
//State-2  
originatorObject.State = "State-2";  
savedStates.Add(originatorObject.GetTheMemento());  
//State-3  
originatorObject.State = "State-3";  
savedStates.Add(originatorObject.GetTheMemento());  
//State-4 which is not saved  
originatorObject.State = "State-4";  
  
//Available restore points  
Console.WriteLine("Currently available restore points are :");  
foreach (Memento m in savedStates)  
{  
    Console.WriteLine(m.State);  
}  
  
//Roll back starts...  
Console.WriteLine("Started restoring process...");  
for (int i = savedStates.Count; i > 0; i--)  
{  
    mementoObject = originatorObject.GetTheMemento();  
    mementoObject.State = savedStates[i - 1].State;  
    originatorObject.RevertToState(mementoObject);  
}  
  
// Wait for user  
Console.ReadKey();  
}
```

Modified Output

Here's the modified output:

```
***Memento Pattern QAs***  
***Demonstration-Caretaker is using multiple restore points***  
  
A new set of verification  
Current State : State-1  
Current State : State-2  
Current State : State-3  
Current State : State-4  
Currently available restore points are :  
State-1  
State-2  
State-3  
Started restoring process...  
Restoring to previous state...  
Current State : State-3  
Restoring to previous state...  
Current State : State-2  
Restoring to previous state...  
Current State : State-1
```

2. Can you use a nongeneric version, say **ArrayList**, in the previous example?

Answer:

I like to follow the advice of the experts, who generally prefer generic versions over nongeneric versions. This is why I always like data structures such as **List**, **Dictionary**, and so on, over their counterparts such as **ArrayList** and **HashTable**. I discuss generics in detail in my earlier book *Interactive C#* (Apress, 2017).

3. What are the key advantages of using the Memento design pattern?

Answer:

- The biggest advantage is that you can always discard the unwanted changes and restore to an intended or stable state.
- You do not compromise on the encapsulation associated with the key objects participating in this model.

4. What are the key challenges associated with the Memento design pattern?

Answer:

- Having more mementos requires more storage. In addition, they put additional burden on a caretaker.
- The previous point increases maintenance costs.
- You cannot ignore the time it takes to save these states, which can impact the overall performance of the system.

In a language such as C# or Java, developers may prefer to use serialization/deserialization techniques instead of directly implementing the Memento design pattern. Both techniques have their own pros and cons, but you can combine both techniques in your application. You can learn more about these techniques here:

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/serialization/>

5. In these implementations, if you make the originator's state public, then your clients also could directly access the states. In other words, you do not need to implement the concept of properties. Is this understanding correct?

Answer:

You cannot break the encapsulation. One of the comments in the code says, "Only the originator that created a memento is allowed to access it." Also, refer to the GoF definition, which starts with "Without violating encapsulation...."

6. I am confused. To support undo operations, which pattern should I prefer, Memento or Command?

Answer:

The GoF has said that these are related patterns. It primarily depends on how you want to handle the situation. Suppose you are adding 25 to an integer. After this addition operation, you want to undo the operation by doing the reverse operation. In other words, $50 + 25 = 75$, so to go back, you will do $75 - 25 = 50$. In this type of operation, you do not need to store the previous state.

But consider a situation where you may need to store the state of your objects prior to the operation. In this case, you'd use Memento. For example, in a paint application, you can avoid the cost of undoing some painting operations by storing the list of objects prior to executing the commands. This stored list can be treated as mementos, and you can keep this list with the associated commands. I suggest that you read this article for more information:

[https://www.developer.com/design/article.php/3720566/
Working-With-Design-Patterns-Memento.htm](https://www.developer.com/design/article.php/3720566/Working-With-Design-Patterns-Memento.htm)

So, a particular application can use both of these patterns to support undo operations.

CHAPTER 20

State Pattern

This chapter covers the State pattern.

GoF Definition

Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

Concept

The concept is best described by the following examples.

Real-Life Example

Consider the scenario of a network connection, say a TCP connection. An object can be in various states; for example, a connection might already be established, a connection might be closed, or the object has already started listening through the connection. When this connection receives a request from other objects, it responds as per its present state.

The functionalities of a traffic signal or a television can also be considered to be an example of the State pattern. For example, you can change the channel if the TV is already in switched-on mode. It will not respond to the channel change requests if it is in switched-off mode.

Computer World Example

Suppose you have a job-processing system that can process a certain number of jobs at a time. When a new job appears, either the system processes the job or it signals that the system is busy with the maximum number of jobs that it can process at a time. In other words, the system will send a busy signal if it gets another job-processing request when its total number of job-processing capabilities has been reached.

Illustration

This example models the functionalities related to a television and its remote control. Suppose you have a remote control to support the operations of a TV. You can simply assume that at any given point in time your TV can be in either of these three states: On, Off, or Mute. Initially the TV is in the Off state. When you press the On button on a remote control, the TV will be in the On state, and then if you press the Mute button, it will go into the Mute state. You can also assume that if you press the Off button when the TV is already in the Off state, if you press the On button when the TV is already in the On state, or if you press the Mute button when the TV is already in Mute mode, there will be no state change for the TV. The TV can go into the Off state from the On state or the Mute state (if you press the Off button). Figure 20-1 shows the state diagram that reflects all these possible scenarios.

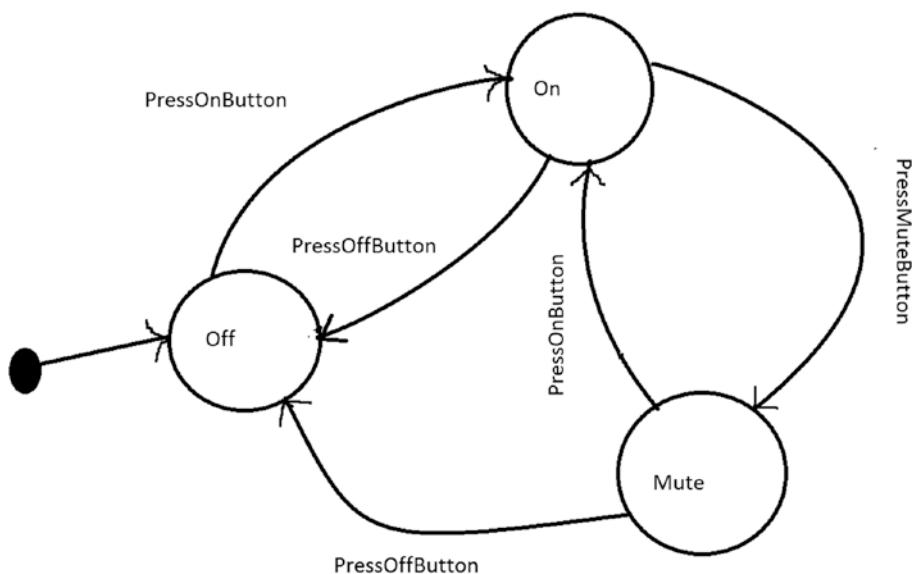


Figure 20-1. Different states of a Television (TV)

For simplicity, in this diagram, I have not marked any state as final, though in the following illustration, you are switching off the TV at the end.

Class Diagram

Figure 20-2 shows the class diagram.

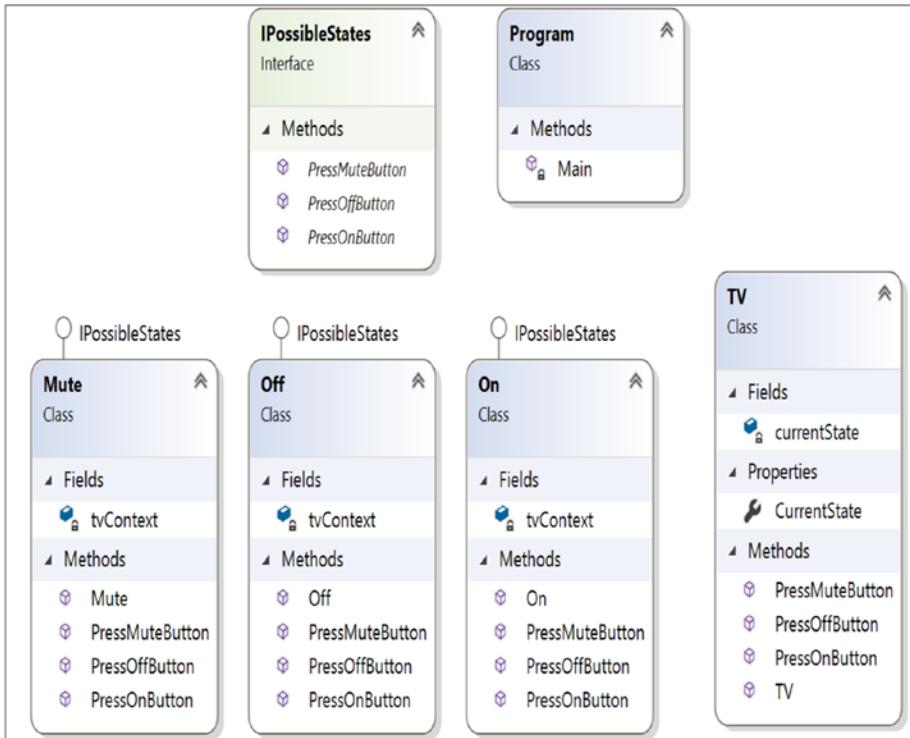


Figure 20-2. Class diagram

Directed Graph Document

Figure 20-3 shows the directed graph document.

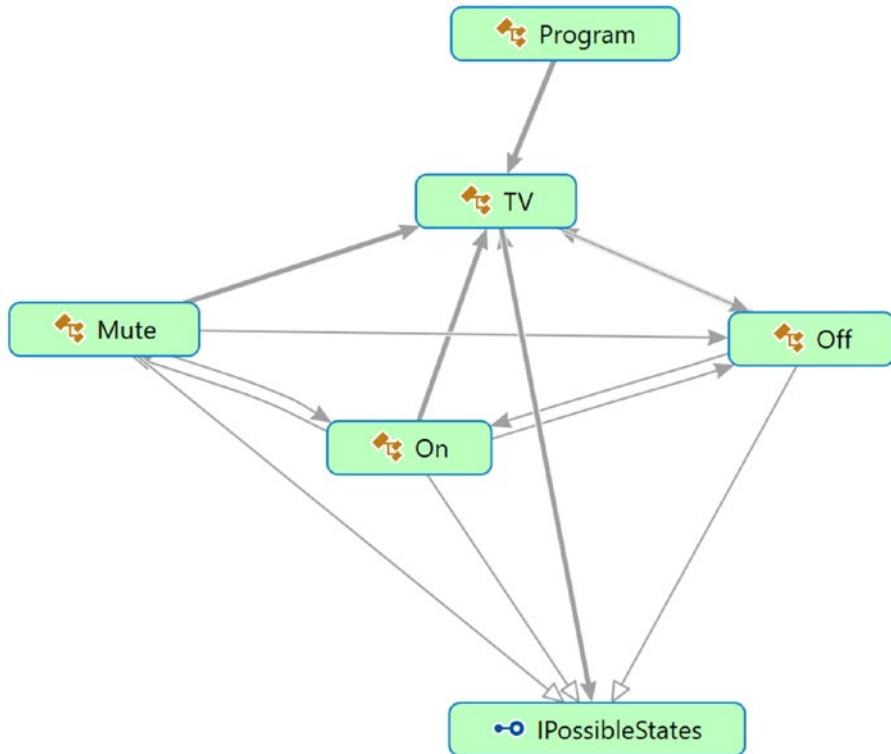


Figure 20-3. *Directed Graph Document*

Solution Explorer View

Figure 20-4 shows the high-level structure of the parts of the program.

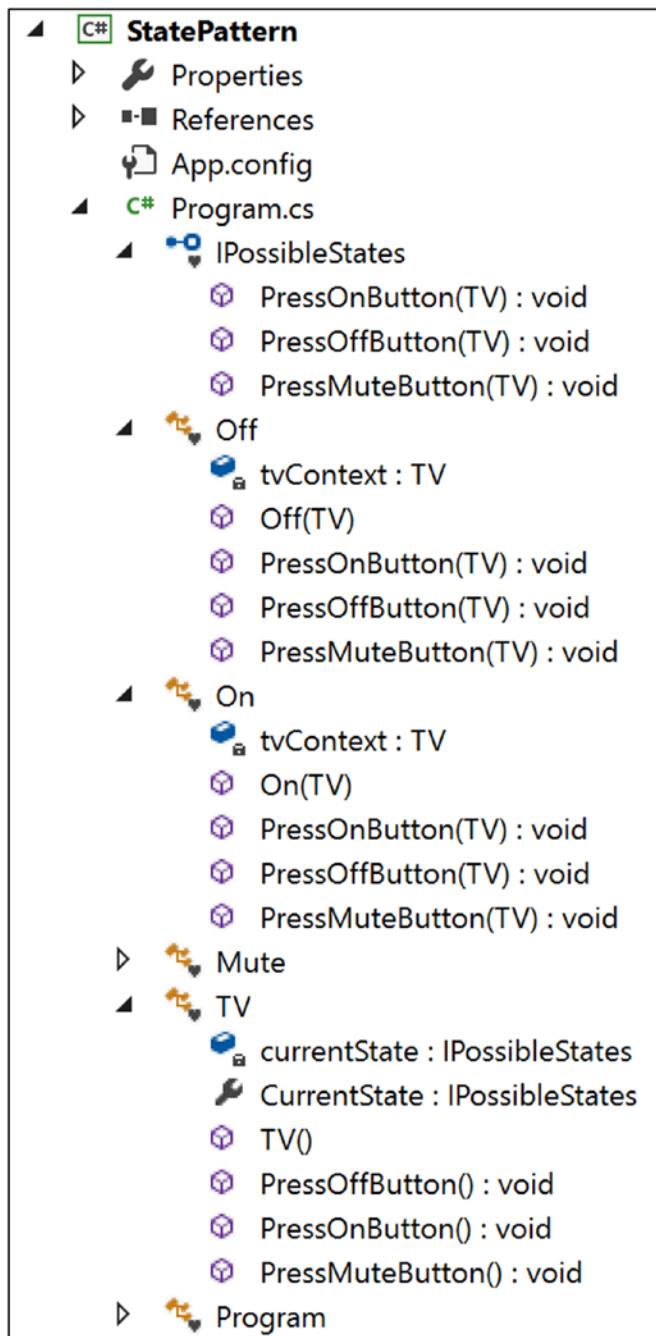


Figure 20-4. Solution Explorer View

Implementation

Here's the implementation:

```
using System;
namespace StatePattern
{
    interface IPossibleStates
    {
        void PressOnButton(TV context);
        void PressOffButton(TV context);
        void PressMuteButton(TV context);
    }

    class Off : IPossibleStates
    {
        TV tvContext;
        //Initially we'll start from Off state
        public Off(TV context)
        {
            Console.WriteLine(" TV is Off now.");
            this.tvContext = context;
        }
        //Users can press any of these buttons at this state-On, Off or Mute
        //TV is Off now, user is pressing On button
        public void PressOnButton(TV context)
        {
            Console.WriteLine("You pressed On button. Going from Off to On
state");
            tvContext.CurrentState = new On(context);
        }
        //TV is Off already, user is pressing Off button again
        public void PressOffButton(TV context)
        {
            Console.WriteLine("You pressed Off button. TV is already in
Off state");
        }
    }
}
```

```
//TV is Off now, user is pressing Mute button
public void PressMuteButton(TV context)
{
    Console.WriteLine("You pressed Mute button. TV is already in
    Off state, so Mute operation will not work.");
}

class On : IPossibleStates
{
    TV tvContext;
    public On(TV context)
    {
        Console.WriteLine("TV is On now.");
        this.tvContext = context;
    }
    //Users can press any of these buttons at this state-On, Off or Mute
    //TV is On already, user is pressing On button again
    public void PressOnButton(TV context)
    {
        Console.WriteLine("You pressed On button. TV is already in On
        state.");
    }
    //TV is On now, user is pressing Off button
    public void PressOffButton(TV context)
    {
        Console.WriteLine("You pressed Off button. Going from On to Off
        state.");
        tvContext.CurrentState = new Off(context);
    }
    //TV is On now, user is pressing Mute button
    public void PressMuteButton(TV context)
    {
        Console.WriteLine("You pressed Mute button. Going from On to
        Mute mode.");
    }
}
```

```
        tvContext.CurrentState = new Mute(context);
    }
}
class Mute : IPossibleStates
{
    TV tvContext;
    public Mute(TV context)
    {
        Console.WriteLine("TV is in Mute mode now.");
        this.tvContext = context;
    }
    //Users can press any of these buttons at this state-On, Off or Mute
    //TV is in mute, user is pressing On button
    public void PressOnButton(TV context)
    {
        Console.WriteLine("You pressed On button. Going from Mute mode
        to On state.");
        tvContext.CurrentState = new On(context);
    }
    //TV is in mute, user is pressing Off button
    public void PressOffButton(TV context)
    {
        Console.WriteLine("You pressed Off button. Going to Mute mode
        to Off state.");
        tvContext.CurrentState = new Off(context);
    }
    //TV is in mute already, user is pressing mute button again
    public void PressMuteButton(TV context)
    {
        Console.WriteLine(" You pressed Mute button. TV is already in
        Mute mode.");
    }
}
```

```
class TV
{
    private IPossibleStates currentState;
    public IPossibleStates CurrentState
    {
        //get;set;//Not working as expected
        get
        {
            return currentState;
        }
        /*Usually this value will be set by the class that
         implements the interface "IPossibleStates"*/
        set
        {
            currentState = value;
        }
    }
    public TV()
    {
        this.currentState = new Off(this);
    }
    public void PressOffButton()
    {
        currentState.PressOffButton(this); //Delegating the state
    }
    public void PressOnButton()
    {
        currentState.PressOnButton(this); //Delegating the state
    }
    public void PressMuteButton()
    {
        currentState.PressMuteButton(this); //Delegating the state
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***State Pattern Demo***\n");
        //Initially TV is Off
        TV tv = new TV();
        Console.WriteLine("User is pressing buttons in the following
sequence:");
        Console.WriteLine("Off->Mute->On->On->Mute->Mute->Off\n");
        //TV is already in Off state
        tv.PressOffButton();
        //TV is already in Off state, still pressing the Mute button
        tv.PressMuteButton();
        //Making the TV on
        tv.PressOnButton();
        //TV is already in On state, pressing On button again
        tv.PressOnButton();
        //Putting the TV in Mute mode
        tv.PressMuteButton();
        //TV is already in Mute, pressing Mute button again
        tv.PressMuteButton();
        //Making the TV off
        tv.PressOffButton();
        // Wait for user
        Console.Read();
    }
}
```

Output

Here's the output:

State Pattern Demo

TV is Off now.

User is pressing buttons in the following sequence:

Off->Mute->On->On->Mute->Mute->Off

You pressed Off button. TV is already in Off state

You pressed Mute button. TV is already in Off state, so Mute operation will not work.

You pressed On button. Going from Off to On state

TV is On now.

You pressed On button. TV is already in On state.

You pressed Mute button. Going from On to Mute mode.

TV is in Mute mode now.

You pressed Mute button. TV is already in Mute mode.

You pressed Off button. Going to Mute mode to Off state.

TV is Off now.

Q&A Session

1. Can you elaborate on how this pattern works in a real-world scenario?

Answer:

Psychologists have repeatedly documented the fact that human beings can perform their best when they are in a relaxed mood but in the reverse scenario, when their minds are filled with tension, they cannot produce great results. That is why they always suggest that we work in a relaxed mood. Now consider a scenario. Suppose, you want to watch the live telecast of the winning moments of your favorite team. To watch and enjoy the moment, you need to power on the TV first. If the TV is not functioning properly at that moment and cannot be in "On" state, you cannot

enjoy the moment. So, if you want to enjoy the moment through your TV, the first criteria is that the TV should change its state from “off” to “on”. The State pattern is helpful if you want to design a similar kind of behavior change in an object when its internal state changes.

2. **In this example, you have considered only three states of a TV: On, Off, and Mute. There can be many other states; for example, there may be a state that deals with connection issues or display conditions. Why have you ignored those issues?**

Answer:

The straightforward answer is because of simplicity. If the number of states increases significantly in the system, then it becomes difficult to maintain the system (and this is one of the key challenges associated with this design pattern). But if you understand this implementation, you can easily add any states you want.

3. **I have noticed that the GoF represented a similar structure for both the State pattern and the Strategy pattern in their famous book. I am confused by that.**

Answer:

Yes, the structures are similar, but you need to remember that their intents are different. Apart from this key distinction, you can think of the patterns like this: With a Strategy pattern, you are getting a better alternative to subclassing. In a State design pattern, different types of behaviors can be encapsulated in a state object, and the context is delegated to any of these states. When a context's internal state changes, its behavior also changes.

State patterns can also help you avoid lots of `if` conditions in some particular contexts. For example, consider the example once again.

If the TV is in the Off state, it cannot go into the Mute state. From this state, it can move to the On state only. So, if you do not like the State design pattern, you may need to write the code like this:

```
class TV
{
    //Some code before
    public void PressOnButton()
    {
        if(currentState==OFF )
        {
            Console.WriteLine("You pressed On button. Going from Off to
            OnState");
        }
        if(currentState==OFF )
        {
            Console.WriteLine("You pressed Off button. TV is already in Off
            state");
        }
        else
        {
            Console.WriteLine("You pressed Mute button. TV is already in Off
            state, so Mute operation will not work.");
        }
        //Some code after
    }
}
```

- 4. How are you implementing the open/close principle in your example?**

Answer:

Each of these TV states is closed for modification, but you can add a new state to the TV class.

5. What are the common characteristics between the Strategy pattern and the State pattern?

Answer:

Both promote composition and delegation.

6. It appears to me that these state objects are acting like singletons. Is this understanding correct?

Answer:

Yes. Most of the times they act in this way.

7. Can you avoid the use of contexts in the method parameters? For example, can you avoid them in statements like this?

```
void PressOnButton(TV context);
```

Answer:

No. Through them you are saving states. They are helping you to evaluate whether you are changing between states or you are already in the same state. Notice the output also. These contexts are helping you to get output like this: "You pressed Mute button. TV is already in Mute mode."

8. What are the pros and cons of the State design pattern?

Answer:

The pros are as follows:

- You have already seen that by following the open/close principle, you can add new states and extend a state's behavior easily.
- It reduces the use of if-else statements. In other words, the conditional complexity is reduced. (Refer to the answer to question 3 earlier.)

You probably already noticed that having more states means having more scattered code, and the obvious side effect is difficult maintenance for you.

CHAPTER 21

Mediator Pattern

This chapter covers the Mediator pattern.

GoF Definition

Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

Concept

A mediator takes responsibility for controlling and coordinating the interactions of a specific group of objects that cannot refer each other explicitly. In other words, a mediator is an intermediary through whom these objects talk to each other. This kind of implementation helps you to reduce the number of interconnections among different objects. As a result, you can reduce the coupling in the system.

Real-Life Example

When an airplane needs to take off, a series of verifications takes place. These kinds of verifications confirm that all components and parts (which are dependent on each other) are in perfect condition.

Another example is when the pilots of different airplanes (who are approaching or departing the terminal area) communicate with the airport towers. They do not explicitly communicate with other pilots in different airlines. They simply send their status to the tower only. These towers send signals to confirm who can take off (or land). You must note that these towers do not control the whole flight. They enforce constraints only in the terminal areas.

Computer World Example

When a client processes a business application, you may need to implement some constraints. For example, suppose you have a form where clients need to supply their user IDs and passwords to access the accounts. In the same form, you may need to supply other mandatory fields such as e-mail ID, communication address, age, and so on. Let's assume you are applying the constraints as follows.

Initially you will check whether the user ID supplied by a user is a valid one. If it is a valid ID, then only the password field will be enabled. After supplying these two fields, you may need to check whether the e-mail ID is provided by the user. Let's assume further that after providing all this information (a valid user ID, a password, a correctly formatted e-mail ID, and so on), your submit button will be enabled. In other words, the submit button will be enabled only if the user supplies a valid user ID, password, a valid email ID, and other mandatory details. You can also ensure that the user ID is an integer, so if a user by mistake provides any character in that field, the submit button will stay in disabled mode. The Mediator pattern becomes handy in such a scenario.

When a program consists of many classes and the logic is distributed among them, the code becomes harder to read and maintain. In those scenarios, if you want to bring new changes into the system's behavior, it can be difficult unless you use the Mediator pattern.

Illustration

Wikipedia describes the Mediator pattern as shown in Figure 21-1 (which is basically adopted from the GoF).

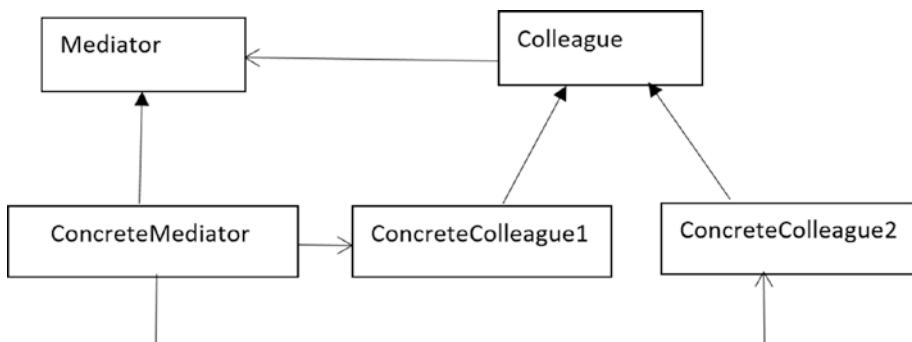


Figure 21-1. Mediator pattern example

The participants are described as follows:

- **Mediator:** This defines the interface that provides the communication among Colleague objects.
- **ConcreteMediator:** This knows and maintains the list of Colleague objects. It implements the Mediator interface and coordinates the communication among the Colleague objects.
- **Colleague:** This defines the interface for communication with other colleagues.
- **ConcreteColleague:** A concrete colleague must implement the Colleague interface. These objects communicate with each other through the mediator.

In this example, I have replaced the word *colleagues* with *friends* (yes, you can assume that it is friendly environment). Let's assume that you have a group of three friends: Amit, Sohel, and Raghu. Raghu is the boss, and he wants to make sure things run smoothly. Let's further assume that the three friends can communicate with each other through a chat server.

At the end of the program, I have introduced another friend, Jack. But he did not register himself with the mediator object, so the mediator is not allowing him to post messages.

Class Diagram

Figure 21-2 shows the class diagram.

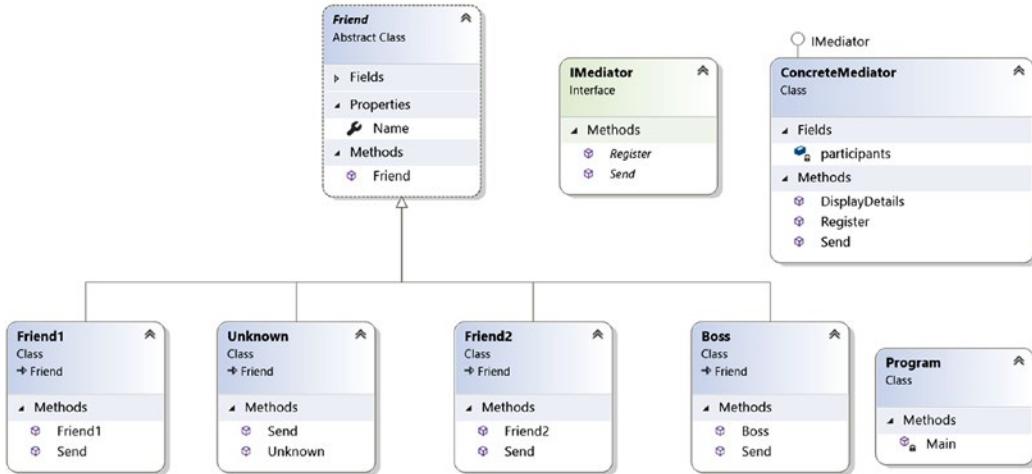


Figure 21-2. Class diagram

Solution Explorer View

Figure 21-3 shows the high-level structure of the parts of the program.

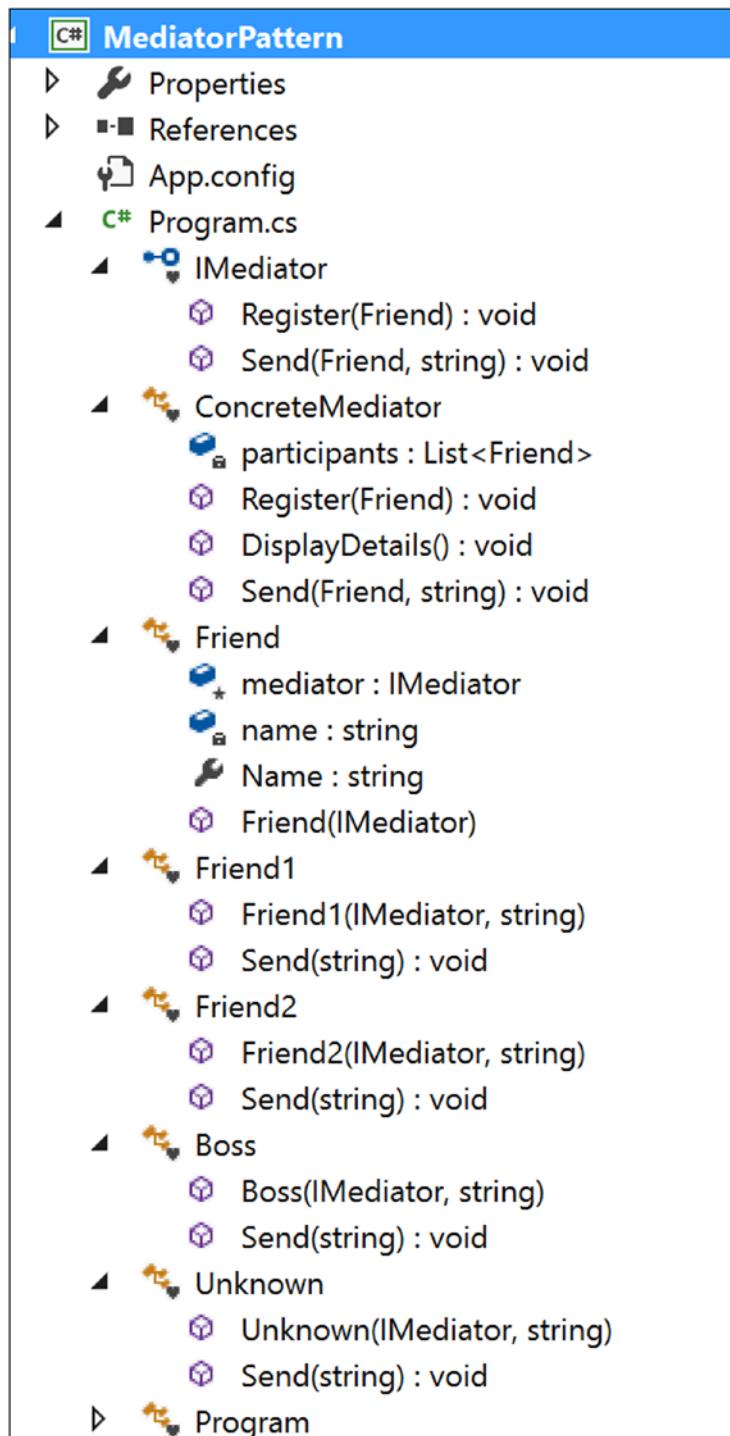


Figure 21-3. Solution Explorer View

Implementation

Here's the implementation:

```
using System;
using System.Collections.Generic;

namespace MediatorPattern
{
    interface IMediator
    {
        void Register(Friend friend);
        void Send(Friend friend, string msg);
    }

    // ConcreteMediator
    class ConcreteMediator : IMediator
    {
        //private Friend friend1,friend2,boss;
        List<Friend> participants = new List<Friend>();
        public void Register(Friend friend)
        {
            participants.Add(friend);
        }
        public void DisplayDetails()
        {
            Console.WriteLine("At present, registered Participants are:");
            foreach (Friend friend in participants)
            {
                Console.WriteLine("{0}", friend.Name);
            }
        }
        public void Send(Friend friend, string msg)
        {
            if (participants.Contains(friend))
            {
```

```
Console.WriteLine(String.Format("[{0}] posts: {1} Last  
message posted {2}", friend.Name, msg, DateTime.Now));  
System.Threading.Thread.Sleep(1000);  
}  
else  
{  
    Console.WriteLine("An outsider named {0} trying to send  
some messages", friend.Name);  
}  
}  
}  
// Friend  
abstract class Friend  
{  
    protected IMediator mediator;  
    private string name;  
    public string Name  
    {  
        get { return name; }  
        set { name = value; }  
    }  
    // Constructor  
    public Friend(IMediator mediator)  
    {  
        this.mediator = mediator;  
    }  
}  
// Friend1-first participant  
class Friend1 : Friend  
{  
    public Friend1(IMediator mediator, string name)  
        : base(mediator)  
    {  
        this.Name = name;  
    }  
}
```

```
public void Send(string msg)
{
    mediator.Send(this, msg);
}
// Friend2-Second participant
class Friend2 : Friend
{
    // Constructor
    public Friend2(IMediator mediator, string name)
        : base(mediator)
    {
        this.Name = name;
    }
    public void Send(string msg)
    {
        mediator.Send(this, msg);
    }
}
/* Friend3-Third Participant. He is the boss.*/
class Boss : Friend
{
    // Constructor
    public Boss(IMediator mediator, string name)
        : base(mediator)
    {
        this.Name = name;
    }
    public void Send(string msg)
    {
        mediator.Send(this, msg);
    }
}
```

```
// Friend4-4th participant who will not register himself to the
// mediator. Still he will try to send a message.
class Unknown: Friend
{
    // Constructor
    public Unknown(IMediator mediator, string name)
        : base(mediator)
    {
        this.Name = name;
    }

    public void Send(string msg)
    {
        mediator.Send(this, msg);
    }
}

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("****Mediator Pattern Demo****\n");

        ConcreteMediator mediator = new ConcreteMediator();

        Friend1 Amit = new Friend1(mediator, "Amit");
        Friend2 Sohel = new Friend2(mediator, "Sohel");
        Boss Raghu = new Boss(mediator, "Raghu");

        //Registering participants
        mediator.Register(Amit);
        mediator.Register(Sohel);
        mediator.Register(Raghu);
        //Displaying the participant's list
        mediator.DisplayDetails();

        Console.WriteLine("Communication starts among participants...");
        Amit.Send("Hi Sohel, can we discuss the mediator pattern?");
    }
}
```

```

        Sohel.Send("Hi Amit,Yup, we can discuss now.");
        Raghu.Send("Please get back to work quickly.");

        //An outsider/unknown person tries to participate
        Unknown unknown = new Unknown(mediator, "Jack");
        unknown.Send("Hello Guys..");

        // Wait for user
        Console.Read();
    }
}
}

```

Output

Here's the output:

Mediator Pattern Demo

At present, registered Participants are:

Amit

Sohel

Raghu

Communication starts among participants....

[Amit] posts: Hi Sohel,can we discuss the mediator pattern? Last message posted 4/6/2018 12:19:17 PM

[Sohel] posts: Hi Amit,Yup, we can discuss now. Last message posted 4/6/2018 12:19:18 PM

[Raghu] posts: Please get back to work quickly. Last message posted 4/6/2018 12:19:19 PM

An outsider named Jack trying to send some messages

Analysis

Note that only registered users can communicate with each other and post messages successfully. The mediator is not allowing any outsider in the system. (Notice the last line of the output.)

Q&A Session

1. Why are you complicating things? In the previous example, each of the participants could talk to each other directly, and you could bypass the use of mediator. Is this understanding correct?

Answer:

In this example, you have only three *registered* participants, so it may appear that they can communicate to each other directly. But think about a more complicated scenario. Suppose a participant can send a message to a target participant if and only if the target participant stays in online mode only (which is the common scenario for a chat server). With this proposed architecture, if the participants try to communicate to each other, each of them needs to check the status of all the other participants before sending a message. And if the number of participants keep growing, can you imagine the complexity of the system?

So, a mediator can rescue you from such a scenario. Figures 21-4 and 21-5 depict the scenario better.

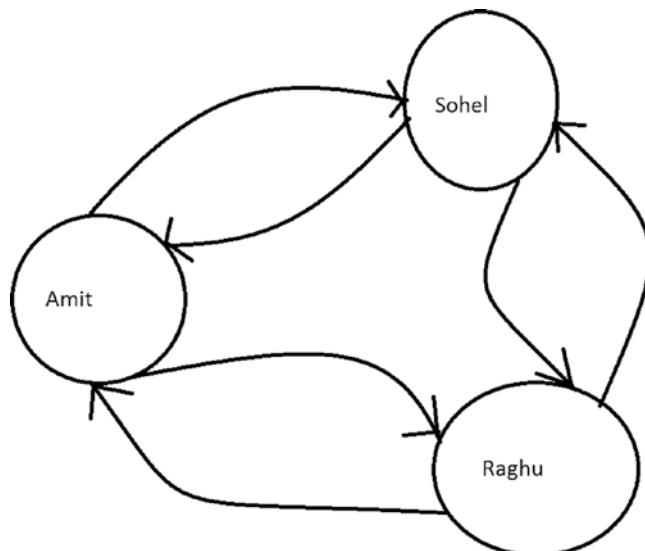


Figure 21-4. Case 1: Without using a mediator

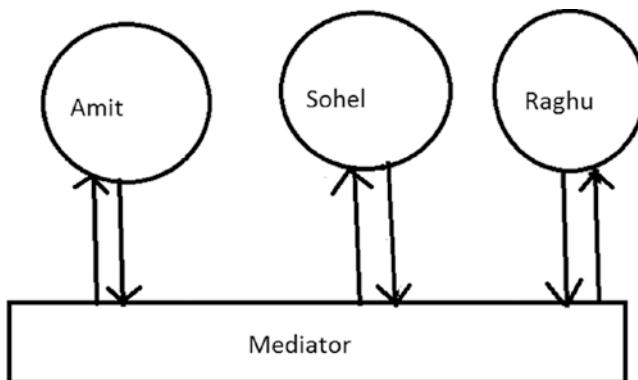


Figure 21-5. Case 2: With a mediator

Modified Illustration

From Figure 21-5, you can understand that the mediator may need to check the status of all the objects, and it may also need to maintain the logic of sending the messages. So, let's modify the program. Notice that I have added a state for each participant and modified the Send method.

Here are the key characteristics of the modified implementation:

- A participant can send messages to another participant if he is online only.
- The mediator takes care of sending the messages to a proper destination, but before it sends a message, the participant's online status is known to him.

Modified Implementation

Here's the modified implementation:

```

using System;

namespace MediatorPatternQAs
{
    interface IMediator
    {
  
```

```
//Our intention is to pass a message from 'fromFriend' to 'toFriend'.
void Send(Friend fromFriend, Friend toFriend, string msg);
}

// ConcreteMediator
class ConcreteMediator : IMediator
{
    private Friend friend1,friend2,boss;

    public Friend Friend1
    {
        set { this.friend1 = value; }
    }

    public Friend Friend2
    {
        set { this.friend2 = value; }
    }

    public Friend Boss
    {
        set { this.boss = value; }
    }

    //Mediator is maintaining the control logic.
    //Message will go from fromFriend to toFriend if toFriend is Online
    //only.

    public void Send(Friend fromFriend,Friend toFriend, string msg)
    {
        if (toFriend.Status == "On")
        {
            Console.WriteLine(String.Format("[{0}->{1}] : {2} Last
message posted {3}", fromFriend.Name,toFriend.Name, msg,
DateTime.Now));
            System.Threading.Thread.Sleep(1000);
        }
    }
}
```

```
        else
        {
            Console.WriteLine(String.Format("[{0}->{1}] : {2}, you
                cannot post messages now. {3} is offline.", fromFriend.
                Name, toFriend.Name, fromFriend.Name, toFriend.Name));

        }
    }
}

// Friend
abstract class Friend
{
    protected IMediator mediator;
    private string name;
    private string status;
    public string Name
    {
        get { return name; }
        set { name = value; }
    }
    public string Status
    {
        get { return status; }
        set { status = value; }
    }

    // Constructor
    public Friend(IMediator mediator)
    {
        this.mediator = mediator;
    }
}

// Friend1-first participant
class Friend1 : Friend
{
    public Friend1(IMediator mediator, string name)
```

```
    : base(mediator)
{
    this.Name = name;
    this.Status = "On";
}
//Message will go to the intended friend.
public void Send(Friend intendedFriend, string msg)
{
    mediator.Send(this, intendedFriend, msg);
}
}
// Friend2-Second participant
class Friend2 : Friend
{
    // Constructor
    public Friend2(IMediator mediator, string name)
        : base(mediator)
    {
        this.Name = name;
        this.Status = "On";
    }
    //Message will go to the intended friend.
    public void Send(Friend intendedFriend, string msg)
    {
        mediator.Send(this, intendedFriend, msg);
    }
}
/* Friend3-Third Participant. He is the boss.*/
class Boss : Friend
{
    // Constructor
    public Boss(IMediator mediator, string name)
        : base(mediator)
    {
        this.Name = name;
```

```
        this.Status = "On";
    }
    //Message will go to the intended friend.
    public void Send(Friend intendedFriend, string msg)
    {
        mediator.Send(this, intendedFriend, msg);
    }
}

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("*** Modified Mediator Pattern Demo***\n");

        ConcreteMediator mediator = new ConcreteMediator();

        Friend1 Amit = new Friend1(mediator, "Amit");
        Friend2 Sohel = new Friend2(mediator, "Sohel");
        Boss Raghu = new Boss(mediator, "Raghu");

        mediator.Friend1 = Amit;
        mediator.Friend2 = Sohel;
        mediator.Boss = Raghu;

        Amit.Send(Sohel, "Hi Sohel, can we discuss the mediator pattern?");
        Sohel.Send(Amit, "Hi Amit, Yup, we can discuss now.");
        Raghu.Send(Amit, "Please get back to work quickly.");
        Raghu.Send(Sohel, "Please get back to work quickly.");

        //Changing the status of Sohel
        Sohel.Status = "Off";
        Amit.Send(Sohel, "I am testing to send a message when Sohel is
in Off state");
        //Sohel is coming online again.
        Sohel.Status = "On";
        Amit.Send(Sohel, "I am testing to send a message when Sohel in
On state again");
```

```

//Amit is going offline.
Amit.Status = "Off";
Raghu.Send(Amit, "Can you please come here?");
Raghu.Send(Sohel, "Can you please come here?");
// Wait for user
Console.Read();
}
}
}

```

Modified Output

Here's the modified output:

*** Modified Mediator Pattern Demo***

At present, registered Participants are:

Amit .Status:On

Sohel .Status:On

Raghu .Status:On

[Amit->Sohel] : Hi Sohel, can we discuss the mediator pattern? Last message posted 4/6/2018 12:23:47 PM

[Sohel->Amit] : Hi Amit, Yup, we can discuss now. Last message posted 4/6/2018 12:23:48 PM

[Raghu->Amit] : Please get back to work quickly. Last message posted 4/6/2018 12:23:49 PM

[Raghu->Sohel] : Please get back to work quickly. Last message posted 4/6/2018 12:23:50 PM

At present, registered Participants are:

Amit .Status:On

Sohel .Status:**Off**

Raghu .Status:On

[Amit->Sohel] : Amit, you cannot post messages now. Sohel is offline.

At present, registered Participants are:

Amit .Status:On

Sohel .Status:On

Raghu .Status:On

[Amit->Sohel] : I am testing to send a message when Sohel in On state again

Last message posted 4/6/2018 12:23:51 PM

At present, registered Participants are:

Amit .Status:Off

Sohel .Status:On

Raghu .Status:On

[Raghu->Amit] : Raghu, you cannot post messages now. Amit is offline.

[Raghu->Sohel] : Can you please come here? Last message posted 4/6/2018

12:23:52 PM

An outsider named Jack trying to send some messages

Note I have made some of the lines bold to draw your attention to those places.

Now you can see that a participant can send messages to another participant if and only if he is online. The mediator takes care of sending the messages to the proper destination, and before it sends a message, it is clear what the participant's online status is.

2. What are advantages of using the Mediator pattern?

Answer:

- You can reduce the complexity of objects communicating in a system.
- The pattern promotes loose coupling. So, objects can be reused.
- The pattern reduces the number of subclasses in the system.
- You replace a many-to-many relationship with a one-to-many relationship, so the code is much easier to read and understand. And as an obvious effect of this, maintenance becomes easier.

- You can provide a centralized control with this pattern.
- In short, it is always a good aim to remove tight coupling from your code, and the Mediator pattern scores high in that context.

3. What are the disadvantages of using the Mediator pattern?

Answer:

- In some cases, implementing the proper encapsulation becomes tricky, and the mediator object's architecture becomes complex.
- Sometimes maintaining a mediator becomes a big concern.

4. If you need to add a new rule or logic, you can directly add it to the mediator. Is this understanding correct?

Answer:

Yes.

5. I am finding some similarities between the Facade pattern and the Mediator pattern. Is this understanding correct?

Answer:

Yes. Steve Holzner in this book *Design Pattern for Dummies* also mentions the similarity by describing the Mediator pattern as a multiplexed Facade pattern. In the case of the Mediator pattern, instead of working with an interface of a single object, you are making a multiplexed interface among multiple objects to do smooth transitions.

6. In this pattern, you are reducing the number of interconnections among various objects. What are the key benefits you have achieved because of this reduction?

Answer:

More interconnections among objects can create a monolithic system that becomes difficult to change (because the behaviors are distributed among many objects). As a side effect, you may need to create many subclasses to bring those changes into the system.

7. **In both implementations, you are using Thread.Sleep(1000). What is the reason for this?**

Answer:

You can ignore that. I have used this to mimic a real-life scenario. I assume that participants are posting the messages after reading them properly, and this activity takes a minimum of 1 second.

CHAPTER 22

Chain of Responsibility Pattern

This chapter covers the Chain of Responsibility pattern.

GoF Definition

Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

Concept

In this pattern, you form a chain of objects where each object in the chain can handle a particular kind of request. If an object cannot handle the request fully, it passes the request to the next object in the chain. This process may continue until the end of the chain. This kind of request-handling mechanism gives you the flexibility to add a new processing object (handler) at the end of the chain. Figure 22-1 shows such a chain with N number of handlers.

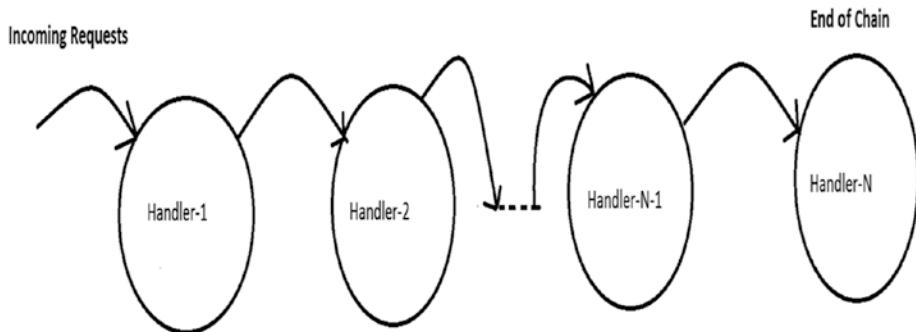


Figure 22-1. *Chain of Responsibility pattern*

Real-Life Example

Most software organizations have some customer care representatives who take feedback from customers and forward any issues to the appropriate departments in the organization. However, not all of these departments will start fixing the issue simultaneously. The department that seems to be responsible will take a look at the issue first, and if those employees believe that the issue should be forwarded to another department, they will forward it.

You may see a similar scenario when a patient visits a hospital. Doctors from one department can refer the patient to a different department (for further diagnosis) if they think it's needed.

Computer World Example

Consider a software application (say, a printer) that can send e-mails and faxes. Obviously, any customer can report either fax issues or e-mail issues, so you need to introduce two different types of error handlers: `EmailErrorHandler` and `FaxErrorHandler`. `EmailErrorHandler` will handle e-mail errors only, and it will not be responsible for fax errors. In the same manner, `FaxErrorHandler` will handle fax errors and will not care about e-mail errors.

You can form a chain like this: whenever your application finds an error, it will just raise a ticket and forward the error with a hope that one of those handlers will handle it. Let's assume that the request first comes to `FaxErrorHandler`. If this handler agrees that it is a fax issue, it will handle it; otherwise, it will forward the issue to `EmailErrorHandler`.

Note that here the chain is ending with `EmailErrorHandler`. But if you need to handle another type of issue, say, an authentication issue because of security vulnerabilities, you can make an `AuthenticationErrorHandler` and put it after `EmailErrorHandler`. Now if an `EmailErrorHandler` also cannot fix the issue completely, it will forward the issue to `AuthenticationErrorHandler`, and the chain will end there. (This is just an example; you are free to place these handlers in any order you'd like.)

The bottom line is that the processing chain may end in either of these two scenarios:

- A handler can process the request completely.
- You have reached the end of the chain.

Illustration

Let's assume that in the following example, you can process both normal and high-priority issues that may come from either e-mail or fax.

Class Diagram

Figure 22-2 shows the class diagram.

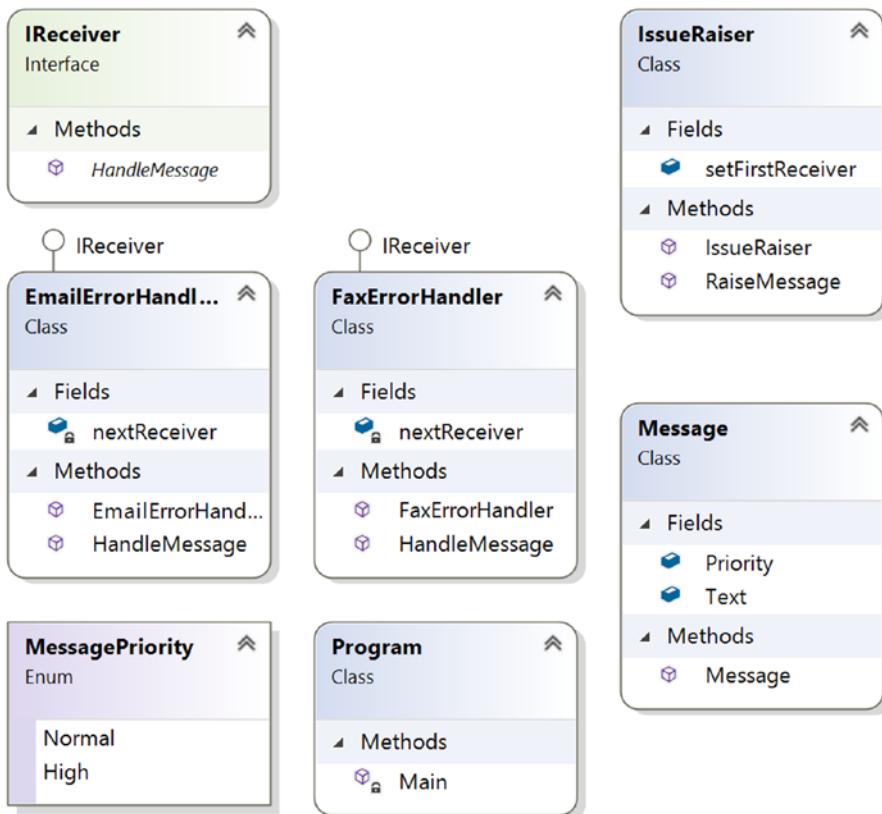


Figure 22-2. Class diagram

Directed Graph Document

Figure 22-3 shows the directed graph document.

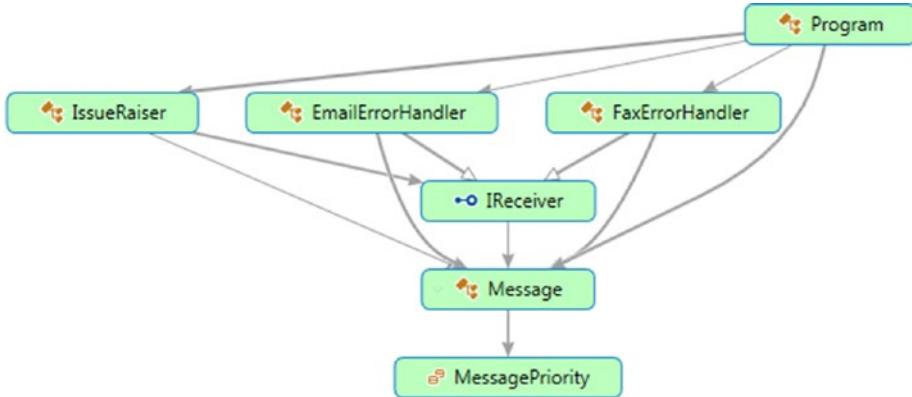


Figure 22-3. Directed Graph Document

Solution Explorer View

Figure 22-4 shows the high-level structure of the parts of the program.

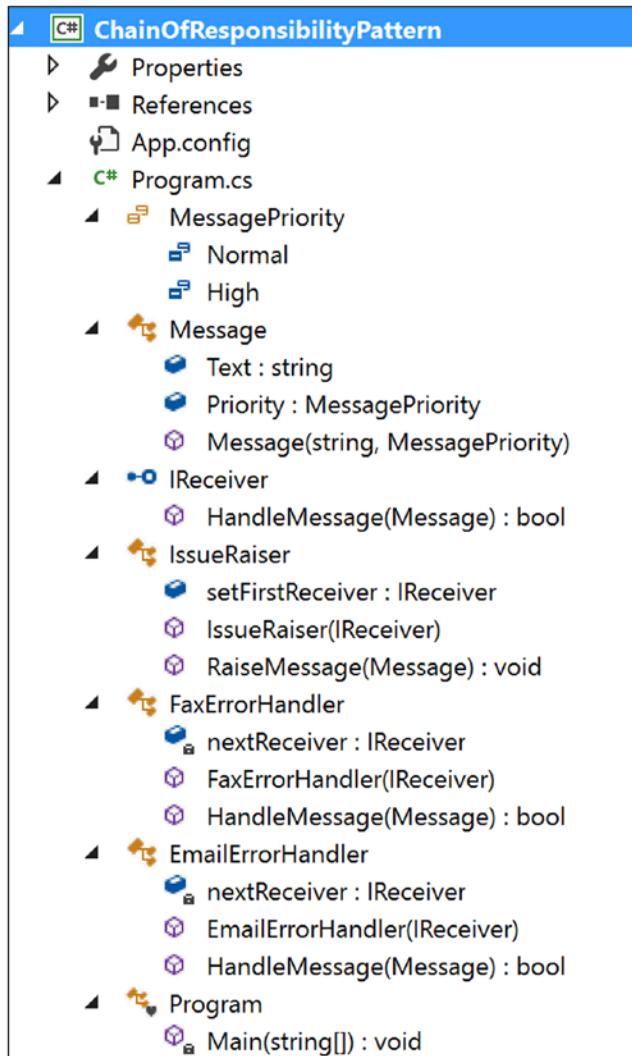


Figure 22-4. Solution Explorer View

Implementation

Here's the implementation:

```
using System;

namespace ChainOfResponsibilityPatternModified
{
    public enum MessagePriority
```

```
{  
    Normal,  
    High  
}  
public class Message  
{  
    public string Text;  
    public MessagePriority Priority;  
    public Message(string msg, MessagePriority p)  
    {  
        Text = msg;  
        this.Priority = p;  
    }  
}  
  
public interface IReceiver  
{  
    bool HandleMessage(Message message);  
}  
public class IssueRaiser  
{  
    public IReceiver setFirstReceiver;  
    public IssueRaiser(IReceiver firstReceiver)  
    {  
        this.setFirstReceiver = firstReceiver;  
    }  
    public void RaiseMessage(Message message)  
    {  
        if (setFirstReceiver != null)  
            setFirstReceiver.HandleMessage(message);  
    }  
}  
public class FaxErrorHandler : IReceiver  
{  
    private IReceiver nextReceiver;  
    public FaxErrorHandler(IReceiver nextReceiver)
```

```
{  
    this.nextReceiver = nextReceiver;  
}  
public bool HandleMessage(Message message)  
{  
    if (message.Text.Contains("Fax"))  
    {  
        Console.WriteLine("FaxErrorHandler processed {0} priority  
issue: {1}", message.Priority, message.Text);  
        return true;  
    }  
    else  
    {  
        if (nextReceiver != null)  
            nextReceiver.HandleMessage(message);  
    }  
    return false;  
}  
}  
public class EmailErrorHandler : IReceiver  
{  
    private IReceiver nextReceiver;  
    public EmailErrorHandler(IReceiver nextReceiver)  
    {  
        this.nextReceiver = nextReceiver;  
    }  
    public bool HandleMessage(Message message)  
    {  
        if (message.Text.Contains("Email"))  
        {  
            Console.WriteLine("EmailErrorHandler processed {0} priority  
issue: {1}", message.Priority, message.Text);  
            return true;  
        }  
        else
```

```

    {
        if (nextReceiver != null)
            nextReceiver.HandleMessage(message);
    }
    return false;
}
}

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("\n ***Chain of Responsibility Pattern
Demo***\n");
/* Making the chain :
    IssueRaiser->FaxErrorHandler->EmailErrorHandler
*/
        IReceiver faxHandler, emailHandler;
        //End of chain
        emailHandler = new EmailErrorHandler(null);
        //fax handler is placed before email handler
        faxHandler = new FaxErrorHandler(emailHandler);

        //Starting point:IssueRaiser will raise issues and set the
        //first //handler
        IssueRaiser raiser = new IssueRaiser(faxHandler);

        Message m1 = new Message("Fax is reaching late to the
        destination.", MessagePriority.Normal);
        Message m2 = new Message("Emails are not raching to
        destinatinations.", MessagePriority.High);
        Message m3 = new Message("In Email, CC field is disabled
        always.", MessagePriority.Normal);
        Message m4 = new Message("Fax is not reaching destination.",
        MessagePriority.High);

        raiser.RaiseMessage(m1);
        raiser.RaiseMessage(m2);
    }
}

```

```

        raiser.RaiseMessage(m3);
        raiser.RaiseMessage(m4);

        Console.ReadKey();
    }
}
}
}

```

Output

Here's the output:

Chain of Responsibility Pattern Demo

FaxErrorHandler processed Normal priority issue: Fax is reaching late to the destination.

EmailErrorHandler processed High priority issue: Emails are not reaching to destinations.

EmailErrorHandler processed Normal priority issue: In Email, CC field is disabled always.

FaxErrorHandler processed High priority issue: Fax is not reaching destination.

Q&A Session

- In the previous example, why do we need the message priorities?**

Answer:

Good catch. Actually, you can ignore the message priorities because, for simplicity, you are just searching for the text *Email* or *Fax* in the handlers. I added these priorities to beautify the code.

Instead of using separate handlers for *Email* and *Fax*, you can make a different kind of chain that can handle the messages based on the priorities.

2. What are the advantages of using the Chain of Responsibility design pattern?

Answer:

- You have more than one object to handle a request. (Notice that if a handler cannot handle the whole request, it may forward the responsibility to the next handler in the chain.)
- The nodes of the chain can be added or removed dynamically. Also, you can shuffle their order. For example, in the previous application, if you see that most issues are e-mail issues, then you may place `EmailErrorHandler` as the first handler to save the average processing time of the application.
- A handler does not need to know how the next handler in the chain will handle the request. It can focus on its own handling mechanism only.
- In this pattern, you are decoupling the senders (of requests) from the receivers.

3. What are the challenges associated with using the Chain of Responsibility design pattern?

Answer:

- There is no guarantee that the request will be handled because you may reach at the end of chain but have not found any explicit receiver to handle the request.
- Debugging becomes tricky with this kind of design.

4. How can you handle the scenario where you have reached the end of the chain but no handler handled the request?

Answer:

One simple solution is through `try/catch` (or `try/finally` or `try/catch/finally`) blocks. You can put all the handlers in the `try` block, and if no one handles the request, you can raise an

exception with the appropriate messages and catch the exception in the catch block to draw your attention to it (or handle it in some different way).

The GoF talked about Smalltalk's automatic forwarding mechanism (`doesNotUnderstand`) in a similar context. If a message cannot find a proper handler, it will be caught in `doesNotUnderstand` implementations that can be overridden to forward the message in the object's successor, log it in some file, and store it in a queue for later processing, or you can simply perform any other intended operations. But you must make a note that by default, this method raises an exception that needs to be handled in a proper way.

5. **In short, we can say that a handler either handles the request fully or passes it to the next handler. Is this understanding correct?**

Answer:

Yes.

6. **It appears to me that there are similarities between the Observer pattern and the Chain of Responsibility pattern. Is this understanding correct?**

Answer:

In an Observer pattern, all registered users get notifications in parallel, but in the case of a Chain of Responsibility pattern, objects in the chain are notified one by one in a sequential manner, and this process will follow until an object handles the notification fully (or you reach at the end of the chain). I have shown the comparisons with diagrams in the “Q&A Session” section of the Observer pattern (Chapter 14).

CHAPTER 23

Interpreter Pattern

This chapter covers the Interpreter pattern.

GoF Definition

Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

Concept

This pattern deals with evaluating sentences in a language. You need to define a grammar to represent the language, and then the interpreter will deal with that grammar. For example, in this example, you will transform a three-digit integer that is input into its equivalent word form (in other words, a string).

Real-Life Example

You can think of a translator who translates a foreign language. You can also consider music notes as a grammar, where musicians play the role of interpreters.

Computer World Example

The Java compiler interprets the Java source code into bytecode that is understandable by the Java Virtual Machine. In C#, the source code is converted to MSIL intermediate code, which is interpreted by the CLR. Upon execution, this MSIL is converted to native code (binary executable code) by the JIT compiler.

Illustration

In general, you represent each of these grammar rules with a class. (To understand some key terms such as *sentences*, *grammar*, *languages*, and so on, you may need to learn about the topic of formal languages in automata theory.) Let's define a simple grammar, as shown here:

$$\begin{aligned} E &::= E_1 E_2 E_3 \\ E_1 &:= \text{"One Hundred"} \mid \text{"Two Hundred"} \mid \dots \mid \text{"Nine Hundred"} \\ E_2 &:= \text{"One Ten and"} \mid \text{"Twenty"} \mid \text{"Thirty"} \mid \dots \mid \text{"Ninety"} \\ E_3 &:= \text{"One"} \mid \text{"Two"} \mid \text{"Three"} \mid \dots \mid \text{"Nine"} \end{aligned}$$

For simplicity and better readability, I am representing this grammar with four classes: `InputExpression` for `E` (an abstract class), `HundredExpression` for `E1`, `TensExpression` for `E2`, and `UnitExpression` for `E3`.

Class Diagram

Figure 23-1 shows the class diagram.

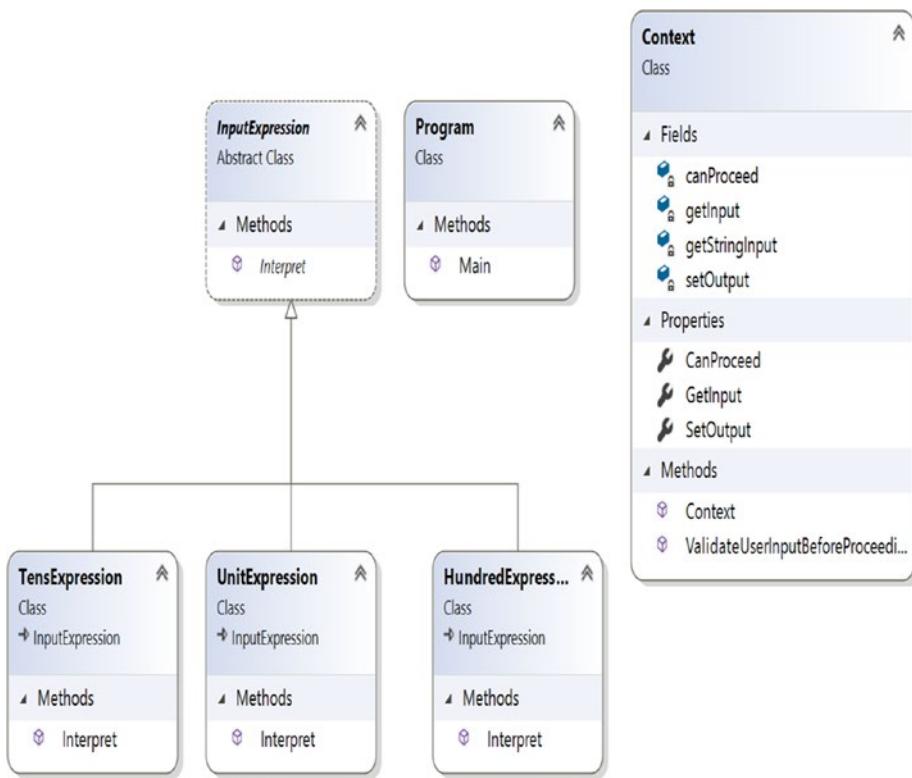


Figure 23-1. Class diagram

Solution Explorer View

Figure 23-2 shows the high-level structure of the parts of the program.

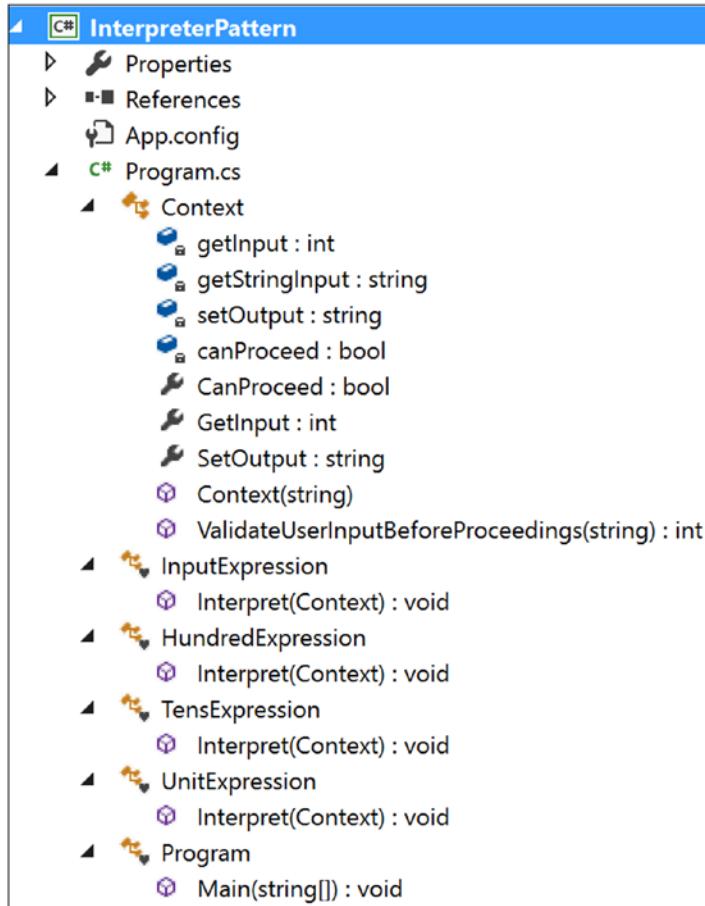


Figure 23-2. Solution Explorer View

Implementation

Here's the implementation:

```
using System;
using System.Collections.Generic;

namespace InterpreterPattern
{
```

```
public class Context
{
    //We will interpret an integer
    private int getInput;
    private string getStringInput;
    //We are printing it in the word form i.e. in String representation
    private string setOutput;
    //Flag-whether it is a valid input or not
    private bool canProceed = false;
    public bool CanProceed
    {
        get {return canProceed;}
    }
    //Using properties to get the input(readonly)
    public int GetInput
    {
        get {return getInput;}
        //set {input = value;}
    }
    //Using properties to get and set output
    public string SetOutput
    {
        get {return setOutput;}
        set {setOutput = value;}
    }
    //Our constructor
    public Context(string input)
    {
        this.getStringInput = input;
    }
    //We'll check whether it is a valid input that lies between 100 and
    //999
    public int ValidateUserInputBeforeProceedings(string inputString)
    {
```

```
if (int.TryParse(inputString, out getInput))
{
    Console.WriteLine("You have entered {0}", getInput);
    //Some basic validations
    if ((getInput < 100) || (getInput > 999))
    {
        Console.WriteLine("Please enter a number between 100
and 999 and try again.");
        //Just returning a 4-digit negative number to indicate
        a wrong input
        return -9999;
    }
}
canProceed = true;
return getInput;
}
}
//abstract class-will hold the common code
abstract class InputExpression
{
    public abstract void Interpret(Context context);
}

class HundredExpression : InputExpression
{
    public override void Interpret(Context context)
    {
        if (context.CanProceed)
        {
            int hundreds = context.GetInput / 100;
            switch (hundreds)
            {
                case 1:
                    context.SetOutput += "One Hundred";
                    break;
            }
        }
    }
}
```

```
        case 2:
            context.SetOutput += "Two Hundred";
            break;
        case 3:
            context.SetOutput += "Three Hundred";
            break;
        case 4:
            context.SetOutput += "Four Hundred";
            break;
        case 5:
            context.SetOutput += "Five Hundred";
            break;
        case 6:
            context.SetOutput += "Six Hundred";
            break;
        case 7:
            context.SetOutput += "Seven Hundred";
            break;
        case 8:
            context.SetOutput += "Eight Hundred";
            break;
        case 9:
            context.SetOutput += "Nine Hundred";
            break;
        default:
            context.SetOutput += "*";
            break;
    }
}
}

class TensExpression : InputExpression
{
public override void Interpret(Context context)
{
```

```
if (context.CanProceed)
{
    int tens = context.GetInput % 100;
    //Process further by dividing it by 10
    tens = tens / 10;
    switch (tens)
    {
        case 1:
            context.SetOutput += "One Ten and";
            break;
        case 2:
            context.SetOutput += "Twenty";
            break;
        case 3:
            context.SetOutput += "Thirty";
            break;
        case 4:
            context.SetOutput += "Forty";
            break;
        case 5:
            context.SetOutput += "Fifty";
            break;
        case 6:
            context.SetOutput += "Sixty";
            break;
        case 7:
            context.SetOutput += "Seventy";
            break;
        case 8:
            context.SetOutput += "Eighty";
            break;
        case 9:
            context.SetOutput += "Ninety";
            break;
    }
}
```

```
        default:
            context.SetOutput += String.Empty;
            break;
        }
    }
}

class UnitExpression : InputExpression
{
    public override void Interpret(Context context)
    {
        if (context.CanProceed)
        {
            int units = context.GetInput % 100;
            //Process further to get the unit digit
            units = units % 10;
            switch (units)
            {
                case 1:
                    context.SetOutput += "One";
                    break;
                case 2:
                    context.SetOutput += "Two";
                    break;
                case 3:
                    context.SetOutput += "Three";
                    break;
                case 4:
                    context.SetOutput += "Four";
                    break;
                case 5:
                    context.SetOutput += "Five";
                    break;
                case 6:
                    context.SetOutput += "Six";
                    break;
            }
        }
    }
}
```

```
        case 7:
            context.SetOutput += "Seven";
            break;
        case 8:
            context.SetOutput += "Eight";
            break;
        case 9:
            context.SetOutput += "Nine";
            break;
        default:
            context.SetOutput += String.Empty;
            break;
    }
}
}

//Client Class
class Program
{
    public static void Main(String[] args)
    {
        Console.WriteLine("****Interpreter Pattern Demo***\n");
        string inputString;
        //int userInput;
        Console.WriteLine("Enter a 3 digit number only (i.e. 100 to
999)");
        inputString = Console.ReadLine();
        //Context context = new Context(userInput);
        Context context = new Context(inputString);
        //Some basic validations before we proceed
        //Checking whether we can parse the string as an integer
        if (context.ValidateUserInputBeforeProceedings(inputString)
!= -9999)
    {
```

```
// Build the 'parse tree'  
List<InputExpression> expTree = new  
List<InputExpression>();  
expTree.Add(new HundredExpression());  
expTree.Add(new TensExpression());  
expTree.Add(new UnitExpression());  
// Interpret the valid input  
foreach (InputExpression inputExp in expTree)  
{  
    inputExp.Interpret(context);  
}  
Console.WriteLine("Original Input {0} is interpreted as  
{1}", context.GetInput, context.SetOutput);  
}  
Console.ReadLine();  
}  
}
```

Output

Here's the output for the positive case:

Interpreter Pattern Demo

Enter a 3 digit number only (i.e. 100 to 999)

375

You have entered 375

Original Input 375 is interpreted as Three Hundred Seventy Five

Here's the output for the negative case:

Interpreter Pattern Demo

Enter a 3 digit number only (i.e. 100 to 999)

- 769

You have entered -769

Please enter a number between 100 and 999 and try again.

Q&A Session

1. In the implementation, why are you using the flag `canProceed`?

It looks like it is redundant. Is this understanding correct?

Answer:

You can execute the program without using this flag. But at the beginning of the interpretation process, each expression wants to confirm from its side that it is going to interpret a valid input. Each expression has the freedom of how to interpret the input.

2. When should you use this pattern?

Answer:

In daily programming life, to be honest, it is not needed much. However, in rare situations you may need to work with your own programming language where it could become handy. But before you proceed, you must ask yourself, what is the return on investment?

3. What are the advantages of using the Interpreter design pattern?

Answer:

- You are very much involved in the process in how to define a grammar for your language and how to represent and interpret those sentences. You can change and extend the grammar also.
- You have full freedom over how to interpret these expressions.

4. What are the challenges associated with using the Interpreter design pattern?

Answer:

I believe that the amount of work is the biggest concern. Also, maintaining a complex grammar becomes tricky because you may need to create (and maintain) separate classes to deal with different rules.

PART II

Additional Design Patterns

CHAPTER 24

Simple Factory Pattern

This chapter covers the Simple Factory pattern.

Definition

Create an object without exposing the instantiation logic to the client.

Concept

In object-oriented programming, a factory is such an object that can create other objects. A factory can be invoked in many different ways but most often, it uses a method that can return objects with varying prototypes. Any subroutine that can help us to create these new objects, can be considered as a factory. Most importantly, it will help you to abstract the process of object creation from the consumers of the application.

Real-Life Example

In a South Indian restaurant, when you place an order for your favorite biryani dish, the waiter may ask whether you like to have your Biryani with more spice or whether it should be prepared with less spice. Based on your choice, the chef will add spices to the core material and make the appropriate dish for you.

Computer World Example

The Simple Factory pattern is common to software applications, but before proceeding, note the following:

- The Simple Factory pattern is not treated as a standard design pattern in the GoF's famous book, but the approach is common to any application you write where you want to separate the code that varies a lot from the part of code that does not vary. It is assumed that you follow this approach in all the applications you write.
- The Simple Factory pattern is considered the simplest form of the Factory Method pattern (and Abstract Factory pattern). So, you can assume that any application that follows either the Factory Method pattern or the Abstract Factory pattern originated from the concept of the Simple Factory pattern's design goals.

In the following implementation, I discuss this pattern with a common use case. Let's go through the implementation.

Illustration

These are the important characteristics of the following implementation:

- In this example, you are dealing with two different types of animals: dogs and tigers. The process of creating an object depends on the user's input.
- You can assume that each type of animal can speak, and they prefer to do some actions.
- In the client code, you will see the following line:
`preferredType = simpleFactory.CreateAnimal();`
- I have put the code for creating objects in a different place (specifically, in a factory class). Using this approach, you are not directly using the new operator in client code.

- You will further notice that I have separated the code that can vary from the code that is least likely to vary. This mechanism can help you to remove tight coupling in the system. (How? Refer to the “Q&A Session” section.)

Note In some places, you may see a variation of this pattern where objects are created through a parameterized constructor such as `preferredType=simpleFactory.CreateAnimal("Tiger")` and where your factory class does not inherit from an abstract class or interface.

Class Diagram

Figure 24-1 shows the class diagram.

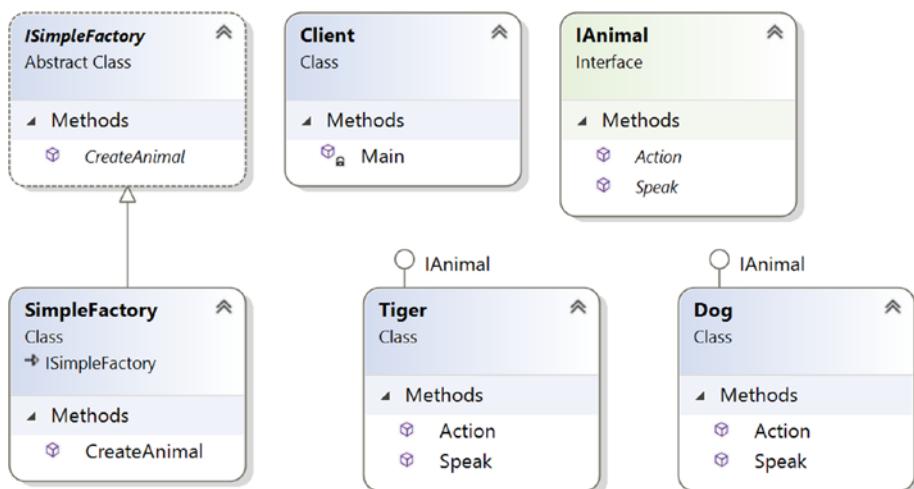


Figure 24-1. Class diagram

Directed Graph Document

Figure 24-2 shows the directed graph document.

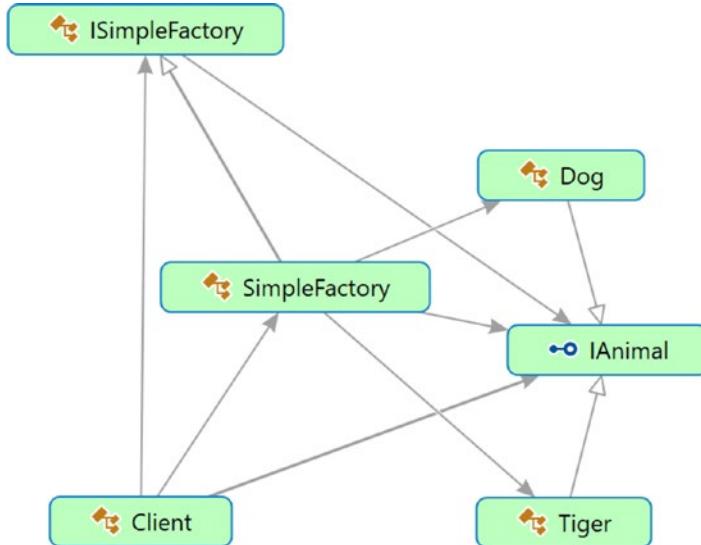


Figure 24-2. Directed Graph Document

Solution Explorer View

Figure 24-3 shows the high-level structure of the parts of the program.

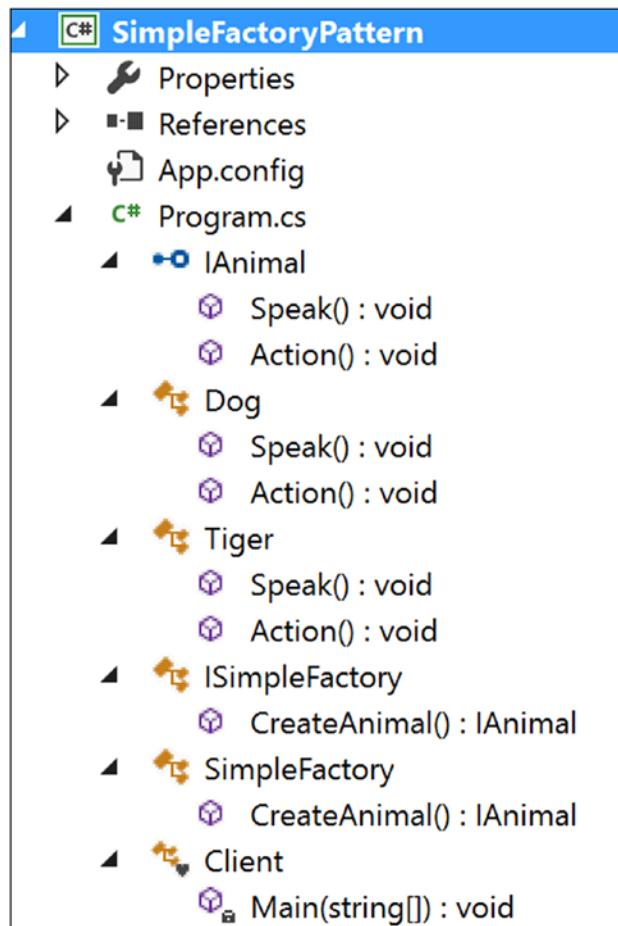


Figure 24-3. Solution Explorer View

Implementation

Here's the implementation:

```
using System;

namespace SimpleFactoryPattern
{
    public interface IAnimal
    {
        void Speak();
        void Action();
    }
}
```

CHAPTER 24 SIMPLE FACTORY PATTERN

```
public class Dog : IAnimal
{
    public void Speak()
    {
        Console.WriteLine("Dog says: Bow-Wow.");
    }
    public void Action()
    {
        Console.WriteLine("Dogs prefer barking...");
    }
}
public class Tiger : IAnimal
{
    public void Speak()
    {
        Console.WriteLine("Tiger says: Halum.");
    }
    public void Action()
    {
        Console.WriteLine("Tigers prefer hunting...");
    }
}
public abstract class ISimpleFactory
{
    public abstract IAnimal CreateAnimal();
}
public class SimpleFactory : ISimpleFactory
{
    public override IAnimal CreateAnimal()
    {
        IAnimal intendedAnimal=null;
        Console.WriteLine("Enter your choice(0 for Dog, 1 for Tiger)");
        string b1 = Console.ReadLine();
        int input;
```

```
if (int.TryParse(b1, out input))
{
    Console.WriteLine("You have entered {0}", input);
    switch (input)
    {
        case 0:
            intendedAnimal = new Dog();
            break;
        case 1:
            intendedAnimal = new Tiger();
            break;
        default:
            Console.WriteLine("You must enter either 0 or 1");
            //We'll throw a runtime exception for any other
            choices.
            throw new ApplicationException(String.Format
                (" Unknown Animal cannot be instantiated"));
    }
}
return intendedAnimal;
}

//A client is interested to get an animal who can speak and perform an
//action.
class Client
{
    static void Main(string[] args)
    {
        Console.WriteLine("*** Simple Factory Pattern Demo***\n");
        IAnimal preferredType=null;
        ISimpleFactory simpleFactory = new SimpleFactory();
        #region The code region that will vary based on users
        preference
        preferredType = simpleFactory.CreateAnimal();
        #endregion
    }
}
```

```
#region The codes that do not change frequently
preferredType.Speak();
preferredType.Action();
#endregion

Console.ReadKey();
}

}
```

Output

The following is case 1, with user input 0:

Simple Factory Pattern Demo

```
Enter your choice( 0 for Dog, 1 for Tiger)
0
You have entered 0
Dog says: Bow-Wow.
Dogs prefer barking...
```

The following is case 2, with user input 1:

Simple Factory Pattern Demo

```
Enter your choice(0 for Dog, 1 for Tiger)
1
You have entered 1
Tiger says: Halum.
Tigers prefer hunting...
```

The following is case 3, with user input 3:

Simple Factory Pattern Demo

```
Enter your choice(0 for Dog, 1 for Tiger)
3
```

You have entered 3

You must enter either 0 or 1

And you will receive the following Exception: "Unknown Animal cannot be instantiated"

Figure 24-4 shows an exception.

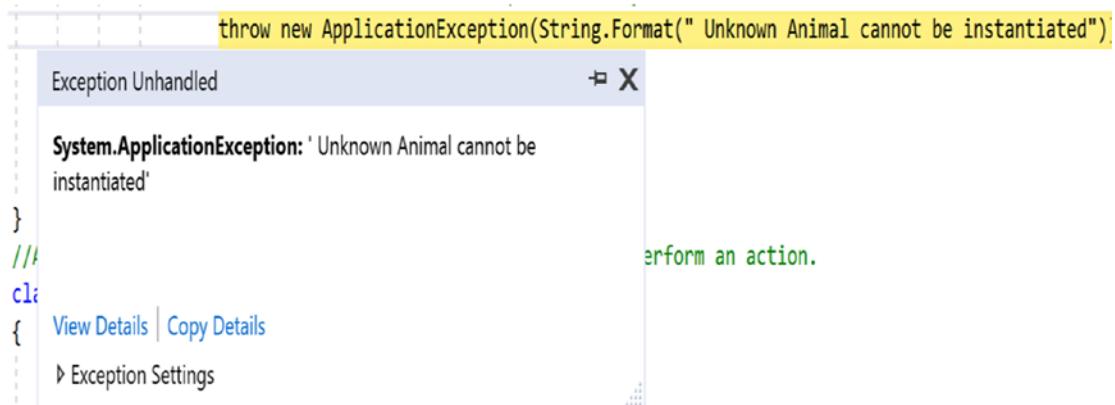


Figure 24-4. Exception encountered due to an invalid input

Q&A Session

1. In this example, I am seeing that the clients are delegating the object's creation through the Simple Factory pattern. But instead of this, they could directly create the objects with the new operator. Is this understanding correct?

Answer:

No. These are key reasons behind the previous design:

- One of the key object-oriented design principles is to separate the parts of your code that are most likely to change from the rest.

In this case, only the creation process for objects changes.

You can assume that these animals must speak and perform some actions and that part of code does not need to vary inside the client code. So, in the future, if there is any change

required in the creation process, you need to change only the `CreateAnimal()` method of the `SimpleFactory` class. The client code will be unaffected because of those changes.

- You do not want to put lots of `if-else` blocks (or `switch` statements) inside the client body. That makes your code clumsy.
- How you are creating the objects is hidden from the client code. This kind of abstraction promotes security.

2. What are the challenges associated with this pattern?

Answer:

If you want to add a new animal or delete an existing animal, you need to modify the `CreateAnimal()` method. This process will violate the open/closed principle (which basically says that your stuff should be open for extension but closed for modification) of the SOLID principles.

Note The SOLID principles were promoted by Robert C. Martin. You can learn about them here: [https://en.wikipedia.org/wiki/SOLID_\(object-oriented_design\)](https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)).

3. Can you ignore the use of `ISimpleFactory` in the previous example?

Answer:

Programming with an abstract class or an interface is always a good practice. This approach can prevent you from lots of changes in the future because any new class can simply implement the interface and settle down in the architecture through polymorphism. But if you solely depend on concrete classes, you need to change your code when you want to integrate a new class in the architecture, and in that case, you will violate the rule that says that your code should be closed for modification.

But yes, in this example, you could put all the stuff directly into the concrete class `SimpleFactory`. You do not need to defer the instantiation process to the subclass. (However, for a factory method pattern, it is mandatory. In this context, remember the GoF definition, which says a factory method lets a class defer instantiation to subclasses.)

4. Can you make the factory class static?

Answer:

You can, but you have to remember the restrictions associated with a static class. For example, you cannot inherit them, and so on.

CHAPTER 25

Null Object Pattern

This chapter covers the Null Object pattern.

Definition

There is no universally accepted definition. So, let's choose the definition from Wikipedia which says the following:

*"In object-oriented computer programming, a null object is an object with no referenced value or with defined neutral ('null') behavior. The null object design pattern describes the uses of such objects and their behavior (or lack thereof). It was first published in the *Pattern Languages of Program Design* book series."*

Concept

The pattern can implement a “do-nothing” relationship, or it can provide a default behavior when an application encounters a null object instead of a real object. In simple words, the core aim is to make a better solution by avoiding a “null objects check” or “null collaborations check” through if blocks.

To explain the concept better, I will explain the problems associated with the following program (which is basically a faulty program), analyze the probable solutions, and, ultimately, implement the concept of this design pattern.

A Faulty Program

Here's a faulty program:

```
using System;

namespace NullObjectPattern
{
    interface IVehicle
    {
        void Travel();
    }

    class Bus : IVehicle
    {
        public static int busCount = 0;
        public Bus()
        {
            busCount++;
        }
        public void Travel()
        {
            Console.WriteLine("Let us travel with Bus");
        }
    }

    class Train : IVehicle
    {
        public static int trainCount = 0;
        public Train()
        {
            trainCount++;
        }
        public void Travel()
        {
            Console.WriteLine("Let us travel with Train");
        }
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("/**Null Object Pattern Demo**\n");
        string input = String.Empty;
        int totalObjects = 0;

        while (true)
        {
            Console.WriteLine("Enter your choice(Type 'a' for Bus, 'b'
for Train) ");
            input = Console.ReadLine();
            IVehicle vehicle = null;
            switch (input)
            {
                case "a":
                    vehicle = new Bus();
                    break;
                case "b":
                    vehicle = new Train();
                    break;
            }
            totalObjects = Bus.busCount + Train.trainCount;
            vehicle.Travel();
            Console.WriteLine("Total objects created in the
system ={0}", totalObjects);
        }
    }
}
```

Output with Valid Inputs

Here is some output with valid inputs:

Enter your choice(Type 'a' for Bus, 'b' for Train)

a

Let us travel with Bus

Total objects created in the system =1

Enter your choice(Type 'a' for Bus, 'b' for Train)

b

Let us travel with Train

Total objects created in the system =2

Enter your choice(Type 'a' for Bus, 'b' for Train)

a

Let us travel with Bus

Total objects created in the system =3

Enter your choice(Type 'a' for Bus, 'b' for Train)

Analysis with Unwanted Input

Let's assume that by mistake the user has supplied a different character, say *e*, as shown here:

Enter your choice(Type 'a' for Bus, 'b' for Train)

a

Let us travel with Bus

Total objects created in the system =1

Enter your choice(Type 'a' for Bus, 'b' for Train)

b

Let us travel with Train

Total objects created in the system =2

Enter your choice(Type 'a' for Bus, 'b' for Train)

a

Let us travel with Bus

Total objects created in the system =3

Enter your choice(Type 'a' for Bus, 'b' for Train)

e

Encountered Exception

This time, you will receive the runtime exception `System.NullReferenceException`, as shown in Figure 25-1.

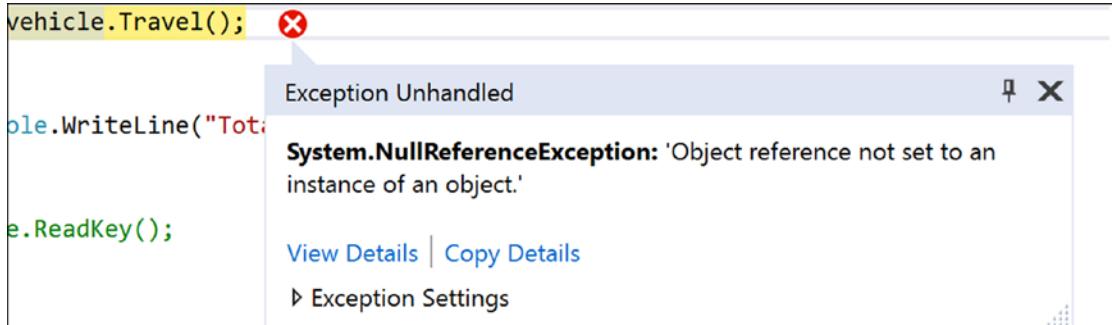


Figure 25-1. A runtime exception occurred

Immediate Remedy

The immediate remedy is to do a null check before invoking the operation, as shown here:

```
if (vehicle != null)
{
    vehicle.Travel();
}
```

Analysis

This solution will work in this case. But think of an enterprise application. If you need to do null checks for every possible scenario like this, you have n number of `if` conditions to evaluate. That will make your code dirty, and as a side effect, the maintenance becomes tough. Instead, you can use the Null Object pattern in this kind of scenario.

Real-Life Example

A washing machine can wash properly if the door is closed and there is a smooth water supply without any internal leakage. But suppose, on one occasion, you forget to close the door or the water supply stops while it's running. The washing machine should not damage itself in those situations, and it should make a beeping sound or create some alarm to draw your attention to the issue.

Computer World Example

As mentioned earlier, in an enterprise application, you can avoid a large number of null checks and *if/else* blocks using this design pattern. The following implementation gives a nice overview of this pattern.

Illustration

In the code implementation, assume that you have two types of vehicles: Bus and Train. A client can opt for a bus or a train through different inputs: *a* or *b*. If by mistake the user supplies any invalid data (in other words, any input other than *a* or *b* in this case), the user cannot travel at all. The application will ignore those invalid inputs by doing nothing through a NullVehicle object. In the following example, you will not create NullVehicle objects repeatedly; once a NullVehicle object is created, you will simply reuse that object.

Class Diagram

Figure 25-2 shows the class diagram.

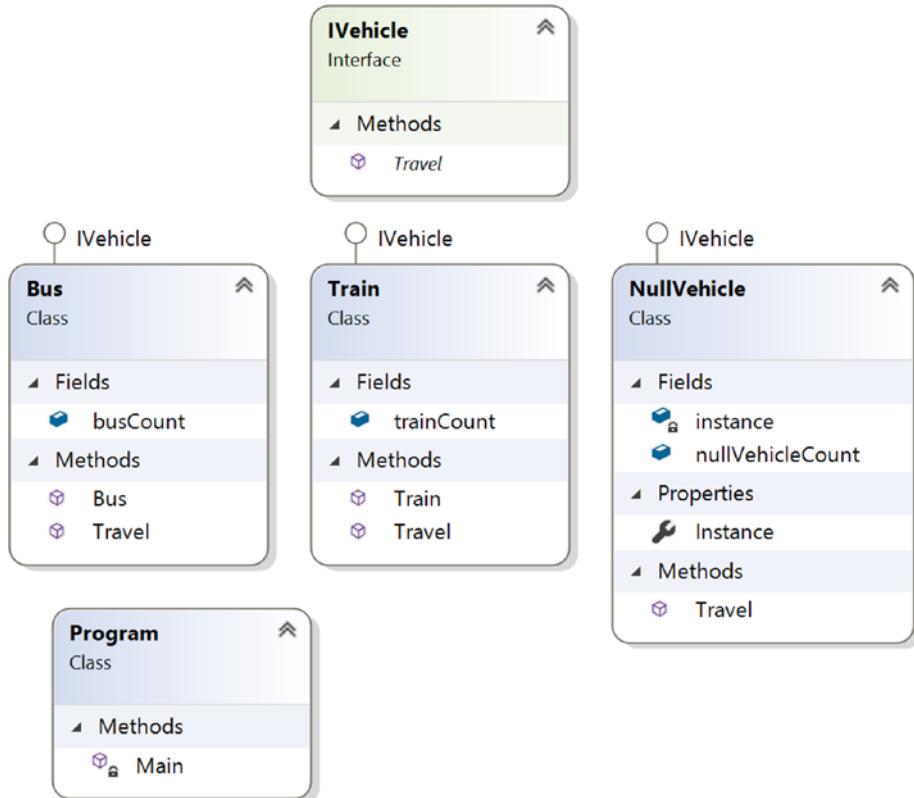


Figure 25-2. Class diagram

Solution Explorer View

Figure 25-3 shows the high-level structure of the parts of the program.

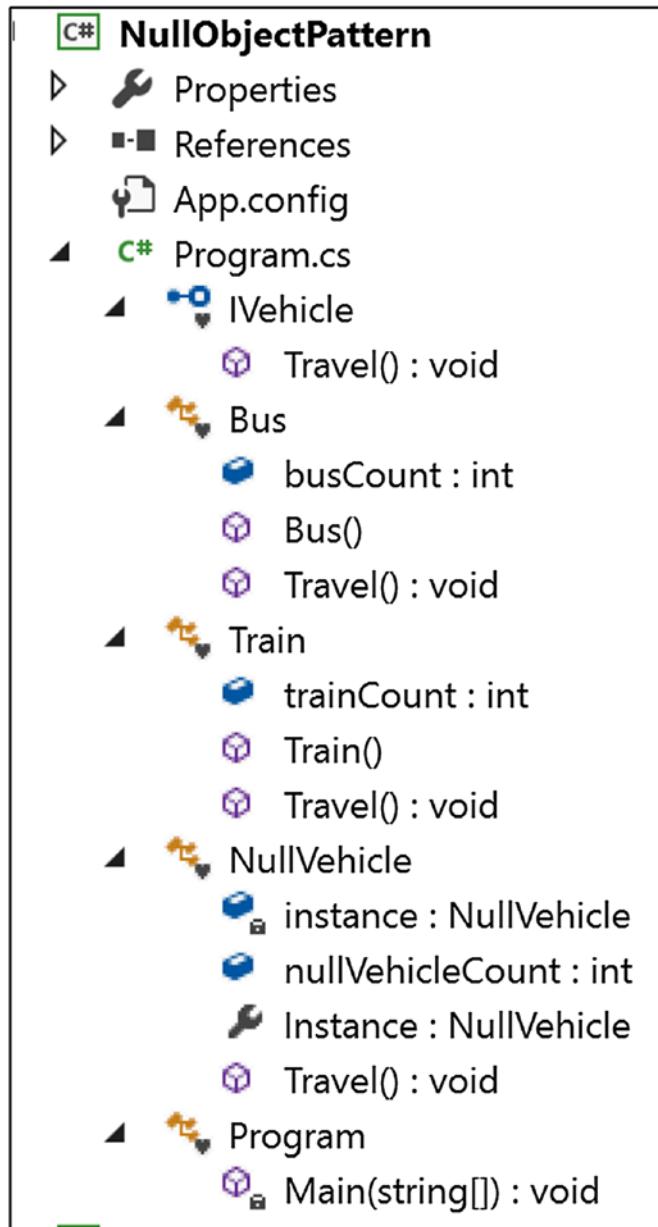


Figure 25-3. Solution Explorer View

Implementation

Here's the implementation:

```
using System;
namespace NullObjectPattern
{
    interface IVehicle
    {
        void Travel();
    }
    class Bus : IVehicle
    {
        public static int busCount = 0;
        public Bus()
        {
            busCount++;
        }
        public void Travel()
        {
            Console.WriteLine("Let us travel with Bus");
        }
    }
    class Train : IVehicle
    {
        public static int trainCount = 0;
        public Train()
        {
            trainCount++;
        }
        public void Travel()
        {
            Console.WriteLine("Let us travel with Train");
        }
    }
}
```

CHAPTER 25 NULL OBJECT PATTERN

```
class NullVehicle : IVehicle
{
    private static readonly NullVehicle instance = new NullVehicle();
    public static int nullVehicleCount = 0;
    public static NullVehicle Instance
    {
        get
        {
            //Console.WriteLine("We already have an instance now.
            //Use it.");
            return instance;
        }
    }
    public void Travel()
    {
        //Do Nothing
    }
}
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***Null Object Pattern Demo***\n");
        string input = String.Empty;
        int totalObjects = 0;

        while (input != "exit")
        {
            Console.WriteLine("Enter your choice( Type 'a' for Bus, 'b'
                for Train.Type 'exit' to quit) ");
            input = Console.ReadLine();
            IVehicle vehicle = null;
            switch (input)
```

```
{  
    case "a":  
        vehicle = new Bus();  
        break;  
    case "b":  
        vehicle = new Train();  
        break;  
    case "exit":  
        Console.WriteLine("Closing the application");  
        vehicle = NullVehicle.Instance;  
        break;  
    default:  
        vehicle = NullVehicle.Instance;  
        if(input=="exit")  
        {  
            Console.WriteLine("Closing the application.  
            Press Enter at end.");  
        }  
        break;  
    }  
    totalObjects = Bus.busCount + Train.trainCount+  
    NullVehicle.nullVehicleCount;  
    //ride the vehicle  
    //if (vehicle != null)  
    //    vehicle.Travel();  
    //}  
  
    Console.WriteLine("Total objects created in the system ={0}",  
    totalObjects);  
  
}  
Console.ReadKey();  
}  
}  
}
```

Output

Here's the output:

```
***Null Object Pattern Demo***

Enter your choice(Type 'a' for Bus, 'b' for Train.Type 'exit' to quit)
a
Let us travel with Bus
Total objects created in the system =1
Enter your choice(Type 'a' for Bus, 'b' for Train.Type 'exit' to quit)
b
Let us travel with Train
Total objects created in the system =2
Enter your choice(Type 'a' for Bus, 'b' for Train.Type 'exit' to quit)
c
Total objects created in the system =2
Enter your choice(Type 'a' for Bus, 'b' for Train.Type 'exit' to quit)
d
Total objects created in the system =2
Enter your choice(Type 'a' for Bus, 'b' for Train.Type 'exit' to quit)
e
Total objects created in the system =2
Enter your choice(Type 'a' for Bus, 'b' for Train.Type 'exit' to quit)
b
Let us travel with Train
Total objects created in the system =3
Enter your choice(Type 'a' for Bus, 'b' for Train.Type 'exit' to quit)
exit
Closing the application.Press Enter at end.
Total objects created in the system =3
```

Consider the following:

- Invalid inputs and their effects are shown in bold.
- Notice that objects counts are not increasing because of null vehicle objects/invalid inputs.
- You did not perform any null check. Still the program execution is not interrupted because of invalid user inputs.

Q&A Session

1. At the beginning of the implementation, I see an additional object is created. Is this intentional?

Answer:

To save some computer memory/storage, I have followed a Singleton design pattern mechanism in the architecture of the NullVehicle class. You do not want to create a NullVehicle object for each invalid input. Over a period of time, it is likely that the application may need to deal with a large number of invalid inputs. If you do not guard against this situation, a huge number of NullVehicle objects will reside in the system (which is basically useless), and they will occupy big amount of memory and you may notice the bad side-effects. (For example, the system may become slow, applications response time may increase etc.)

2. When should you use this pattern?

Answer:

- The pattern is useful if you do not want to encounter a NullReferenceException (for example, if by mistake you try to invoke a method of a null object).
- You can ignore lots of null checks in your code.
- Without these null checks, you make your code cleaner and easily maintainable.

3. What are the challenges associated with the Null Object pattern?

Answer:

- Most often, you may want to find and fix the root cause of failure. So, if you throw a `NullReferenceException`, that can work better for you. You can always handle those exceptions in a `try/catch` block or in a `try/catch/finally` block and update the log information accordingly.
- The Null Object pattern basically helps you to implement a default behavior when you unconsciously want to deal with an object that is not present at all. But trying to supply such a default behavior for all kinds of objects is not appropriate always.

CHAPTER 26

MVC Pattern

This chapter covers the MVC pattern.

Definition

There is no universally accepted definition. So, let's consider the definition from two commonly used resources. Wikipedia (<https://en.wikipedia.org/wiki/Model-view-controller>) says the following:

“Model-view-controller (MVC) is an architectural pattern commonly used for developing user interfaces that divides an application into three interconnected parts. This is done to separate internal representations of information from the ways information is presented to and accepted from the user. The MVC design pattern decouples these major components allowing for efficient code reuse and parallel development.”

wiki.c2.com (<http://wiki.c2.com/?ModelViewController>) says this: “We need smart models, thin controllers, and dumb views.”

Concept

From the definition, it is apparent that the pattern consists of these major components: model, view, and controller. The controller is placed between the view and model in such a way that they can communicate to each other only through the controller. This model separates the mechanism of how the data is displayed from the mechanism of how the data will be manipulated. Figure 26-1 shows the MVC pattern.

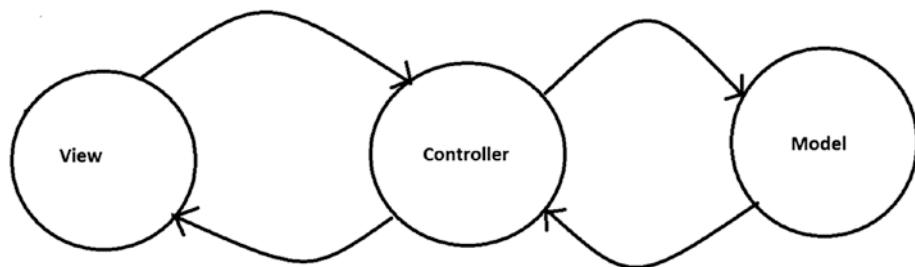


Figure 26-1. A typical MVC architecture

Key Points to Remember

These are brief descriptions of the key components in this pattern:

- The view is used to represent the output. You can think it as a user interface/GUI. It can be designed with HTML, CSS, WPF (for .NET), AWT/Swing (in Java), and so on.
- The model manages the data and business logic. It knows how to store, manage/manipulate the data, and handle the requests that come from controller. But this component is separated from the view component. A typical example is a database, a file system, or similar kind of storage. It can be designed with Oracle, SQL Server, DB2, Hadoop, MySQL, and so on.
- The controller is the intermediary. It accepts a user's input from the view component and passes the request to the model. When it gets a response from the model, it will pass the data to a view. It can be designed with C# .NET, ASP.NET, VB.NET, Core Java, JSP, Servlets, PHP, Ruby, Python, and so on.

You may notice that the concept is implemented with some variations in different application architectures. Here are some examples:

- You can have multiple views.
- Views can pass runtime values (for example, using JavaScript) to controllers.
- Your controller can validate the user's inputs.

- Your controller can receive input in various ways. For example, it can get input from a web request via a URL, or input can be passed by clicking a Submit button on a form.
- In some frameworks, you may see that the model can update the view component also.

There is no universally agreed upon implementation for this pattern. The following are some variations of an MVC architecture.

Variation 1

Figure 26-2 shows variation 1.

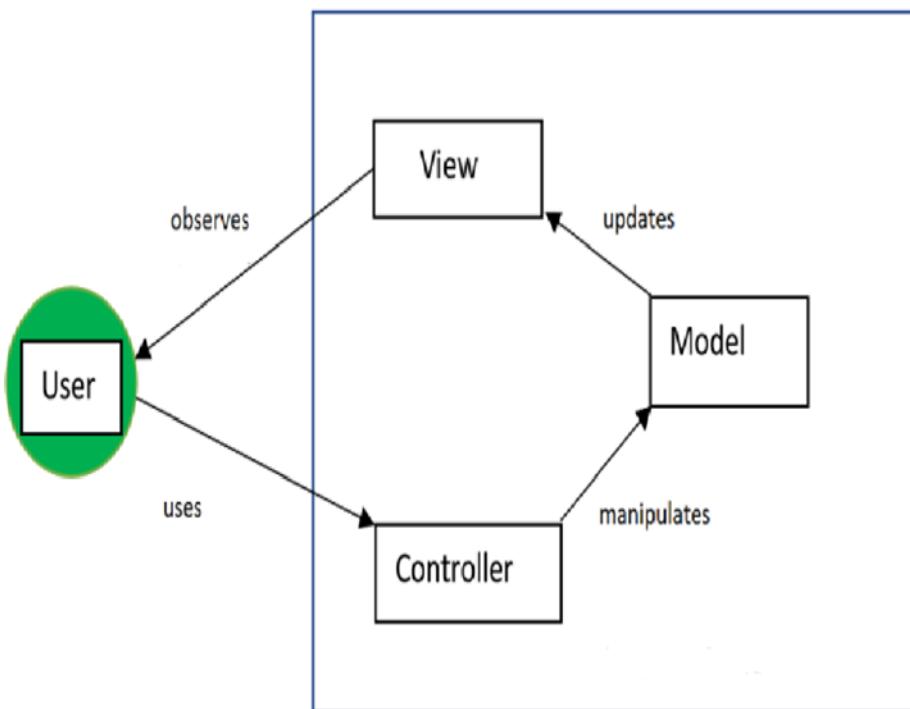


Figure 26-2. A typical MVC framework

Variation 2

Figure 26-3 shows variation 2.

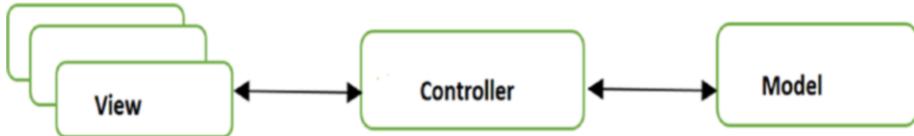


Figure 26-3. An MVC framework with multiple views

Variation 3

Figure 26-4 shows variation 3.

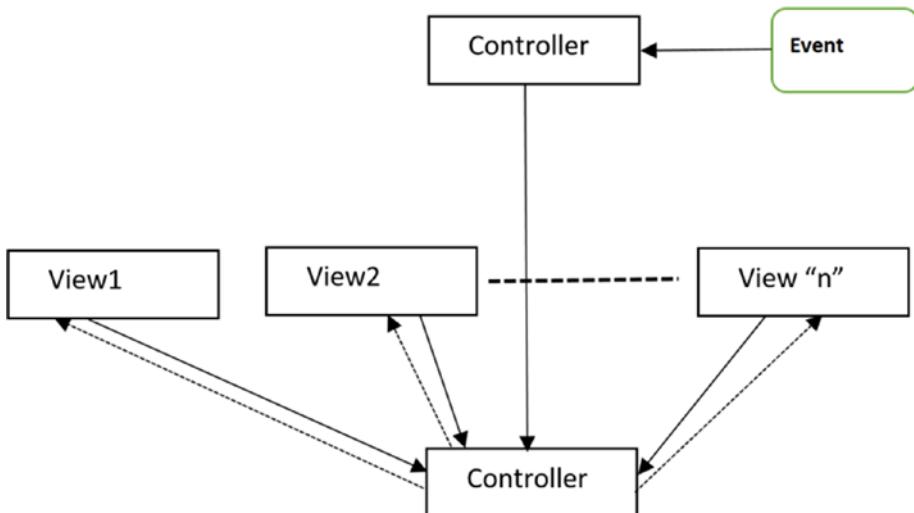


Figure 26-4. An MVC pattern implemented with an Observer pattern/event-based mechanism

My favorite description of MVC comes from Connelly Barnes. He says this:

An easy way to understand MVC: the model is the data, the view is the window on the screen, and the controller is the glue between the two.

—Connelly Barnes (<http://wiki.c2.com/?ModelViewController>)

Real-Life Example

Let's revisit the Factory Method pattern's real-life example but this time interpret it differently. I said that in a restaurant, based on customer inputs, a chef will vary the taste and make the final products. But customers do not place their orders directly with the chef. The customers see the menu (the view), may consult with the waiter, and finally place the order. The waiter passes the order slip to the chef, who gathers the required materials from the restaurant's kitchen (similar to a computer database). Once the meal is prepared, the waiter carries the plate to the customer's table. So, you can consider the waiter as the controller and the chef and kitchen as the model.

Computer World Example

Many web programming frameworks use the concept of the MVC framework. Typical examples include Django, Ruby on Rails, ASP.NET, and so on. A typical ASP.NET MVC project can have the view shown in Figure 26-5.

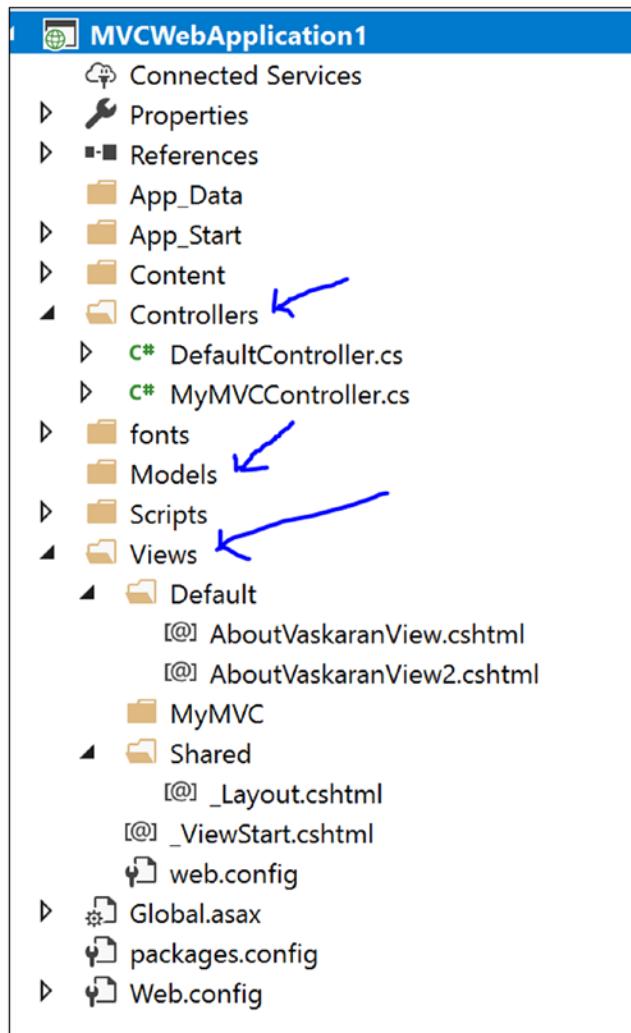


Figure 26-5. Solution Explorer View of a typical ASP.NET MVC Project

Illustration

From a C# perspective, most of the time you will want to use ASP.NET (or a similar technology) to implement the MVC pattern because you will have a lot of built-in support. At the same time, you may need to learn some new terminology. For simplicity, let's assume you want to test the MVC pattern with a simple Windows Forms application, as shown in Figure 26-6.

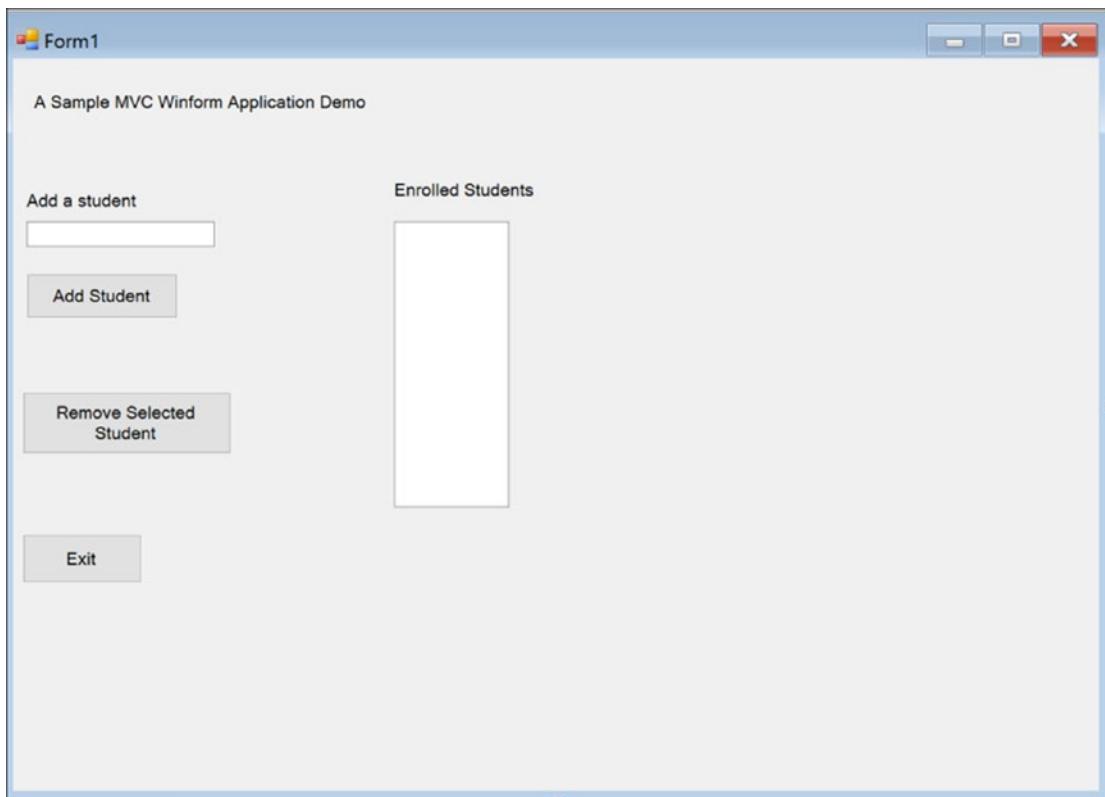


Figure 26-6. Windows Forms application

The requirements are simple:

- At any point in time, you should be able to see the enrolled students in the corresponding list view.
- You should be able to add a new student name (for simplicity, you are considering only the name attribute) to this list view and can also delete a student from the list view.

In this type of application, you can put lots of verifications/constraints. But here my focus is on the MVC pattern, not on the WinForms application or those kinds of verifications/constraints. *In the following implementation, I have taken care of the fact that you cannot add an empty student name. Also, to delete a student name, you need to select it first. Multiple selection is also disallowed in this application.*

If you follow the code, you will see that when you want to add a student name, the control flows to the `AddStudent()` method of `StudentController`, which in turn calls the method `AddStudentToModel` in `StudentModel`. When the processing is completed in `StudentModel`, control comes back following the reverse path. When you want to remove a student, control follows the same paths.

Class Diagram

Figure 26-7 shows the class diagram.

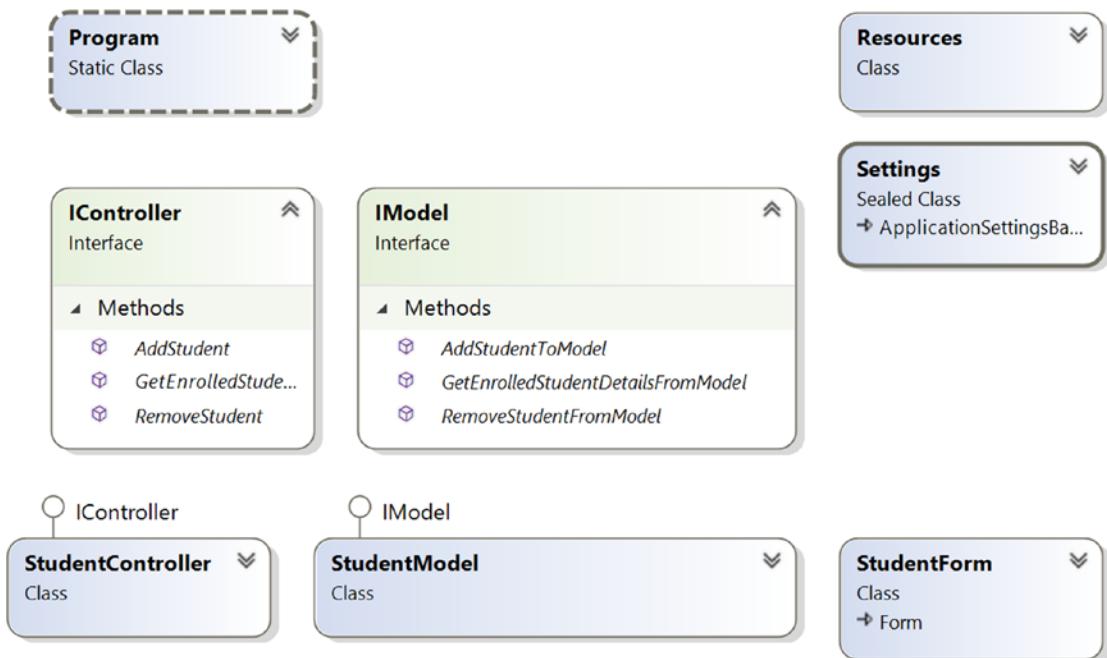


Figure 26-7. Class diagram

Solution Explorer View

Figure 26-8 shows the high-level structure of the parts of the program.

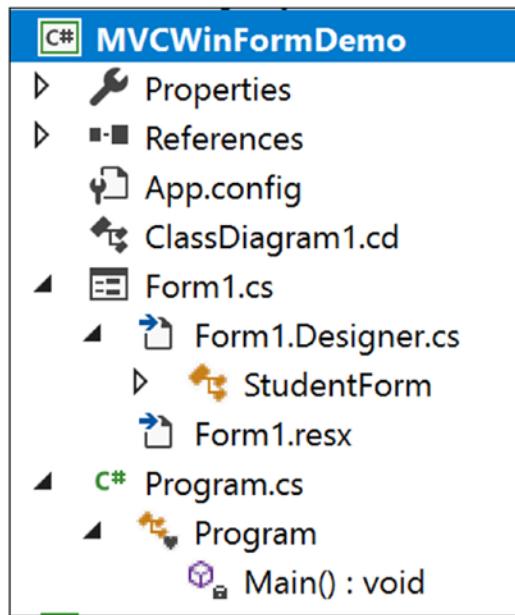


Figure 26-8. Solution Explorer View

Implementation

Form1.cs and Program.cs are the main parts of the following implementation. Form1.Designer.cs is included for your reference only. You are free to choose your preferred variable names and positions in that file.

```
//Form1.cs
using System;
using System.Collections.Generic;
using System.Windows.Forms;

namespace MVCWinFormDemo
{
    //View
    public partial class StudentForm : Form
    {
        StudentController studentController;
        public StudentForm()
        {
```

```
        InitializeComponent();
    }

private void Form1_Load(object sender, EventArgs e)
{
    studentController = new StudentController(new StudentModel(),
this);
    this.showStudentsListView.MultiSelect = false;
    this.showStudentsListView.HideSelection = false;
    //Show enrolled student at the beginning
    studentController.GetEnrolledStudents();
}

public void ShowEnrolledStudents(List<string> studentList)
{
    //clear the listview first
    showStudentsListView.Items.Clear();
    System.Text.StringBuilder sb = new System.Text.StringBuilder();
    foreach (string student in studentList)
    {
        showStudentsListView.Items.Add(student);
    }
}

private void addStudentButton_Click(object sender, EventArgs e)
{
    //We will not add an empty student name
    if (addStudentTextBox.Text != String.Empty)
    {
        //Get the new name from GUI
        string newName = addStudentTextBox.Text;
        List<string> updatedStudentList = new List<string>();
        //Add a student name and get the new student list through
        //controller
        updatedStudentList = studentController.AddStudent(newName);
        //Now Display back to GUI.This value is coming through the
        //controller.
    }
}
```

```
showStudentsListView.Items.Clear();
studentController.GetEnrolledStudents();
//clearing the addStudentTextBox content
addStudentTextBox.Clear();
}
}

private void removeStudentButton_Click(object sender, EventArgs e)
{
    //We can select only one item at a time
    if(showStudentsListView.SelectedItems.Count==1)
    {
        //Get the name from GUI
        string studentName = showStudentsListView.SelectedItems[0].
        Text;
        List<string> updatedStudentList = new List<string>();
        //Remove the student name and get the new student list
        //through controller
        updatedStudentList = studentController.
        RemoveStudent(studentName);
        //Now Display back to GUI.This value is coming through the
        //controller.
        showStudentsListView.Items.Clear();
        studentController.GetEnrolledStudents();
    }
}

private void exitButton_Click(object sender, EventArgs e)
{
    this.Close();
}
}

//Controller
public interface IController
{
    void GetEnrolledStudents();
```

```
    List<string> AddStudent(String studentName);
    List<string> RemoveStudent(String studentName);
}
public class StudentController : IController
{
    private IModel model;
    private StudentForm view;

    public StudentController(IModel model, StudentForm view)
    {
        this.model = model;
        this.view = view;
    }
    public void GetEnrolledStudents()
    {
        List<string> enrolledStudents = model.
        GetEnrolledStudentDetailsFromModel();
        view.ShowEnrolledStudents(enrolledStudents);
    }
    //Sending a request to model to add a student to the student list
    public List<string> AddStudent(String studentName)
    {
        List<string> postAdditionStudentList = new List<string>();
        postAdditionStudentList = model.AddStudentToModel(studentName);
        return postAdditionStudentList;
    }
    //Sending a request to model to remove a student from the student
    //list
    public List<string> RemoveStudent(String studentName)
    {
        List<string> postRemovalStudentList = new List<string>();
        postRemovalStudentList = model.RemoveStudentFromModel(studentName);
        return postRemovalStudentList;
    }
}
```

```
//Model
public interface IModel
{
    List<string> GetEnrolledStudentDetailsFromModel();
    List<string> AddStudentToModel(string studentName);
    List<string> RemoveStudentFromModel(string studentName);
}

public class StudentModel:IModel
{
    private List<string> enrolledStudents = new List<string>{"Amit",
    "John", "Sam"};
    public List<string> GetEnrolledStudentDetailsFromModel()
    {
        return enrolledStudents;
    }
    //Adding a student to the model(student list)
    public List<string> AddStudentToModel(string studentName)
    {
        enrolledStudents.Add(studentName);
        return enrolledStudents;
    }
    //Removing a student from model(student list)
    public List<string> RemoveStudentFromModel(string studentName)
    {
        enrolledStudents.Remove(studentName);
        //After the removal of a student name, send the updated list to
        //controller
        return enrolledStudents;
    }
}

//Form1.Designer.cs

namespace MVCHinFormDemo
{
    partial class StudentForm
```

```
{  
    /// <summary>  
    /// Required designer variable.  
    /// </summary>  
    private System.ComponentModel.IContainer components = null;  
  
    /// <summary>  
    /// Clean up any resources being used.  
    /// </summary>  
    /// <param name="disposing">true if managed resources should be  
    /// disposed; otherwise, false.</param>  
    protected override void Dispose(bool disposing)  
{  
        if (disposing && (components != null))  
        {  
            components.Dispose();  
        }  
        base.Dispose(disposing);  
    }  
  
    #region Windows Form Designer generated code  
  
    /// <summary>  
    /// Required method for Designer support-do not modify  
    /// the contents of this method with the code editor.  
    /// </summary>  
    private void InitializeComponent()  
{  
        this.showStudentsListView = new System.Windows.Forms.  
        ListView();  
        this.label2 = new System.Windows.Forms.Label();  
        this.addStudentLabel = new System.Windows.Forms.Label();  
        this.addStudentButton = new System.Windows.Forms.Button();  
        this.addStudentTextBox = new System.Windows.Forms.TextBox();  
        this.removeStudentButton = new System.Windows.Forms.Button();  
        this.label1 = new System.Windows.Forms.Label();  
        this.exitButton = new System.Windows.Forms.Button();  
    }  
}
```

```
this.SuspendLayout();
//
// showStudentsListView
//
this.showStudentsListView.Location = new System.Drawing.
Point(465, 198);
this.showStudentsListView.Name = "showStudentsListView";
this.showStudentsListView.Size = new System.Drawing.Size
(140, 348);
this.showStudentsListView.TabIndex = 6;
this.showStudentsListView.UseCompatibleStateImageBehavior = false;
//
// label2
//
this.label2.AutoSize = true;
this.label2.Location = new System.Drawing.Point(460, 147);
this.label2.Name = "label2";
this.label2.Size = new System.Drawing.Size(182, 25);
this.label2.TabIndex = 7;
this.label2.Text = "Enrolled Students";
//
// addStudentLabel
//
this.addStudentLabel.AutoSize = true;
this.addStudentLabel.Location = new System.Drawing.Point(12, 160);
this.addStudentLabel.Name = "addStudentLabel";
this.addStudentLabel.Size = new System.Drawing.Size(145, 25);
this.addStudentLabel.TabIndex = 9;
this.addStudentLabel.Text = "Add a student";
//
// addStudentButton
//
this.addStudentButton.Location = new System.Drawing.Point
(17, 262);
this.addStudentButton.Name = "addStudentButton";
this.addStudentButton.Size = new System.Drawing.Size(184, 54);
```

```
    this.addStudentButton.TabIndex = 10;
    this.addStudentButton.Text = "Add Student";
    this.addStudentButton.UseVisualStyleBackColor = true;
    this.addStudentButton.Click += new System.EventHandler(this.
addStudentButton_Click);
    //
    // addStudentTextBox
    //
    this.addStudentTextBox.Location = new System.Drawing
.Point(17, 198);
    this.addStudentTextBox.Name = "addStudentTextBox";
    this.addStudentTextBox.Size = new System.Drawing.Size(229, 31);
    this.addStudentTextBox.TabIndex = 11;
    //
    // removeStudentButton
    //
    this.removeStudentButton.Location = new System.Drawing.
Point(12, 406);
    this.removeStudentButton.Name = "removeStudentButton";
    this.removeStudentButton.Size = new System.Drawing.Size(254, 75);
    this.removeStudentButton.TabIndex = 12;
    this.removeStudentButton.Text = "Remove Selected Student";
    this.removeStudentButton.UseVisualStyleBackColor = true;
    this.removeStudentButton.Click += new System.EventHandler(this.
removeStudentButton_Click);
    //
    // label1
    //
    this.label1.AutoSize = true;
    this.label1.Location = new System.Drawing.Point(21, 40);
    this.label1.Name = "label1";
    this.label1.Size = new System.Drawing.Size(416, 25);
    this.label1.TabIndex = 13;
    this.label1.Text = "A Sample MVC Winform Application Demo";
    //
```

```
// exitButton
//
this.exitButton.Location = new System.Drawing.Point(12, 579);
this.exitButton.Name = "exitButton";
this.exitButton.Size = new System.Drawing.Size(145, 59);
this.exitButton.TabIndex = 14;
this.exitButton.Text = "Exit";
this.exitButton.UseVisualStyleBackColor = true;
this.exitButton.Click += new System.EventHandler(this.
exitButton_Click);
//
// StudentForm
//
this.AutoScaleDimensions = new System.Drawing.SizeF(12F, 25F);
this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
this.ClientSize = new System.Drawing.Size(1324, 891);
this.Controls.Add(this.exitButton);
this.Controls.Add(this.label1);
this.Controls.Add(this.removeStudentButton);
this.Controls.Add(this.addStudentTextBox);
this.Controls.Add(this.addStudentButton);
this.Controls.Add(this.addStudentLabel);
this.Controls.Add(this.label2);
this.Controls.Add(this.showStudentsListView);
this.Name = "StudentForm";
this.Text = "Form1";
this.Load += new System.EventHandler(this.Form1_Load);
this.ResumeLayout(false);
this.PerformLayout();
this.PerformLayout();

}

#endregion
private System.Windows.Forms.ListView showStudentsListView;
private System.Windows.Forms.Label label2;
private System.Windows.Forms.Label addStudentLabel;
```

CHAPTER 26 MVC PATTERN

```
    private System.Windows.Forms.Button addStudentButton;
    private System.Windows.Forms.TextBox addStudentTextBox;
    private System.Windows.Forms.Button removeStudentButton;
    private System.Windows.Forms.Label label1;
    private System.Windows.Forms.Button exitButton;
}
}

//Program.cs
using System;
using System.Windows.Forms;

namespace MVCWinFormDemo
{
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            StudentForm studentView = new StudentForm();
            IModel studentModel = new StudentModel();
            IController cnt = new StudentController(studentModel,studentView);
            Application.Run(new StudentForm());
        }
    }
}
```

The Exit button simply closes the form. I have not put any MVC framework- specific logic into its click event.

Output

Figure 26-9 shows the output when the application is loaded with default values.

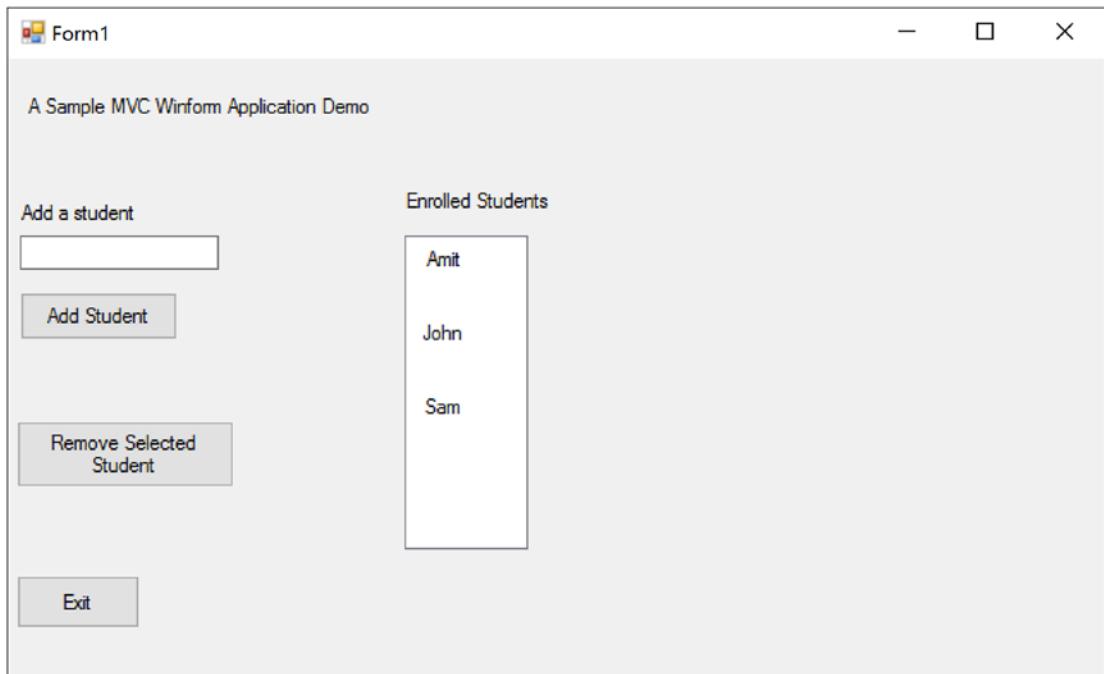


Figure 26-9. Application loaded with default values

CHAPTER 26 MVC PATTERN

Figure 26-10 shows the output when adding a student named Bob.

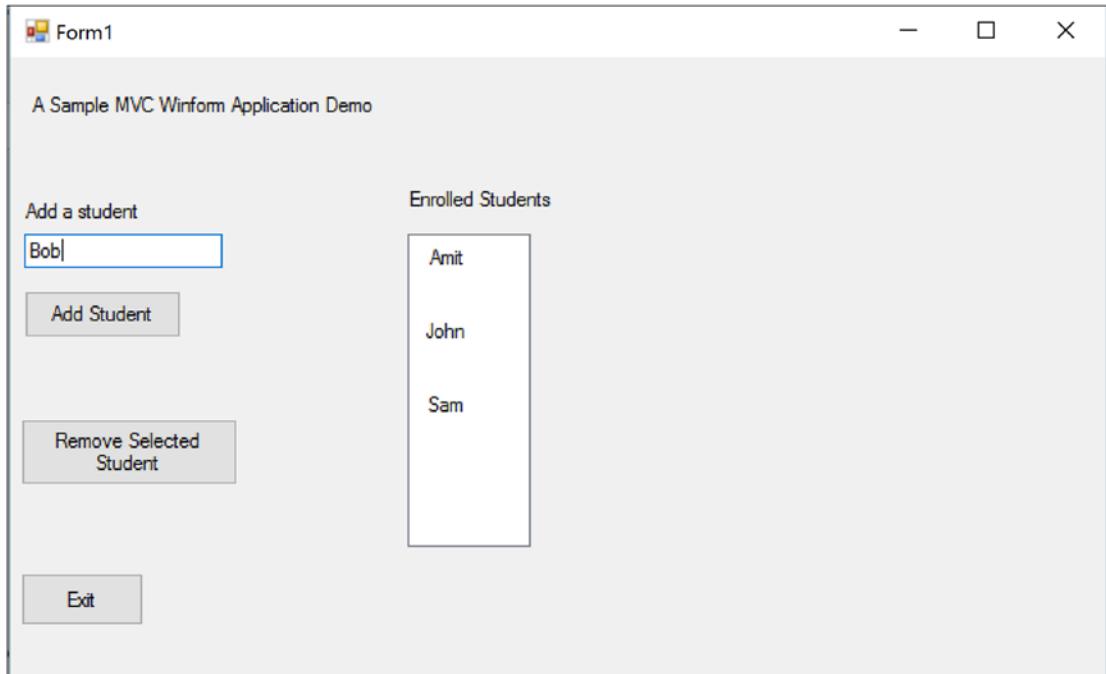


Figure 26-10. Prior to add Bob as a student

Figure 26-11 shows the output when you click the Add Student button now.

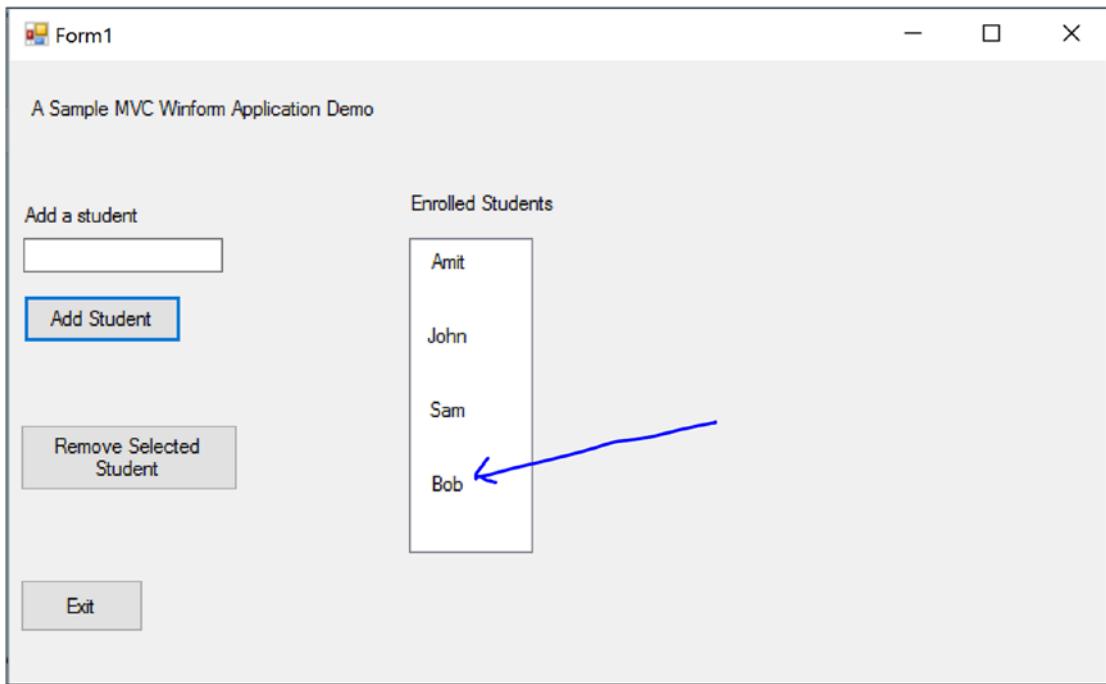


Figure 26-11. Immediate output after clicking the “Add Student” button

CHAPTER 26 MVC PATTERN

Figure 26-12 shows the output when selecting a student named John to remove.

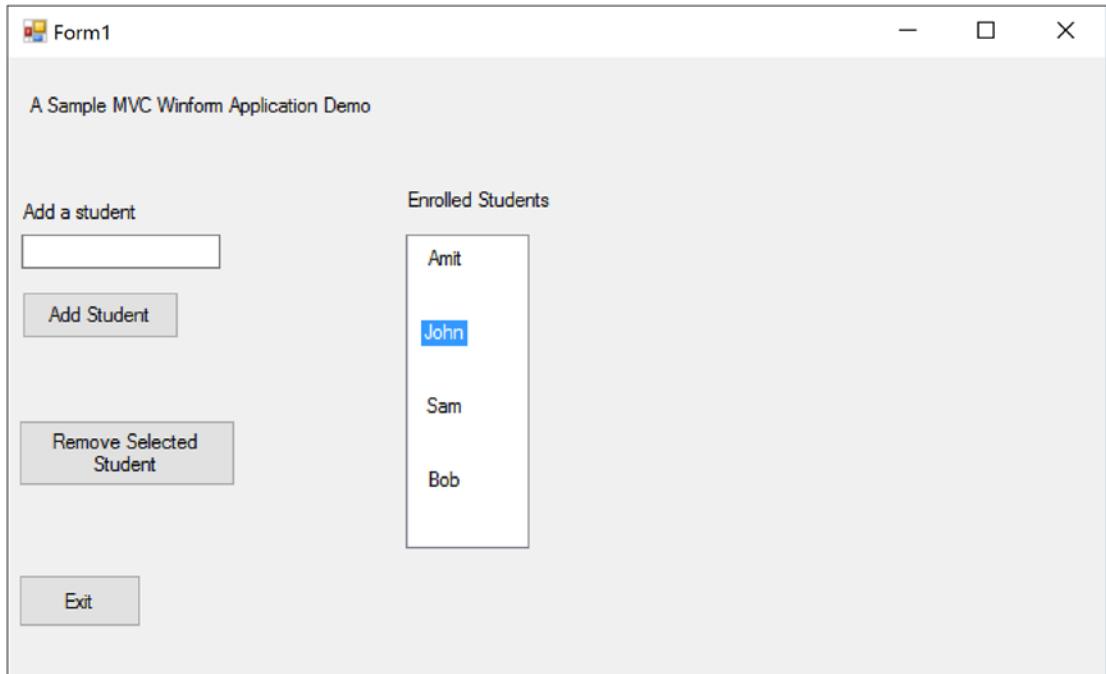


Figure 26-12. Prior to remove the student “John” from the enrolled students list

Figure 26-13 shows the output when clicking the Remove Selected Student button.

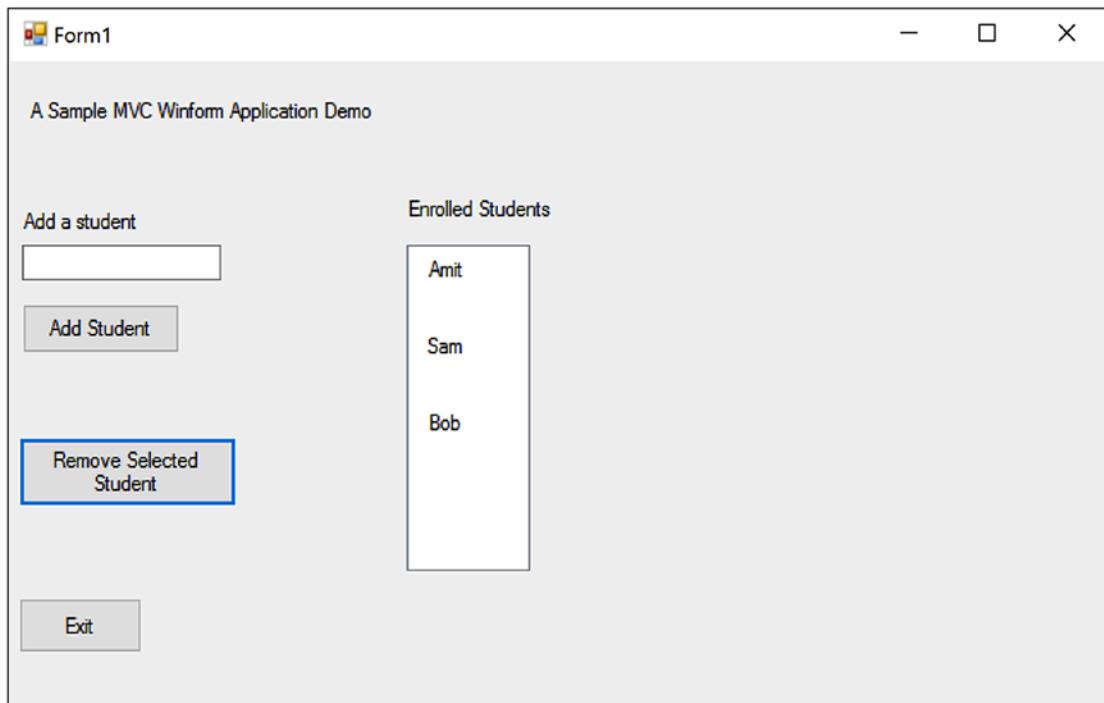


Figure 26-13. Immediate output after removing the student “John” from students list

Q&A Session

1. Suppose you have a programmer, a DBA, and a graphic designer. What are their roles in an MVC architecture?

Answer:

The graphic designer will design the view layer, the DBA will make the model, and the programmer will work to make an intelligent controller.

2. What are the key advantages of using the MVC design pattern?

Answer:

- High cohesion and low coupling are the benefits of MVC. You have probably noticed that tight coupling between the model and the view is easily removed in this pattern. So, the application can be easily extendable and reusable.
- The pattern supports parallel development.
- You can also accommodate multiple runtime views.

3. What are the challenges associated with the MVC pattern?

Answer:

- It requires highly skilled personnel.
- For a tiny application, it may not be suitable.
- Developers need to be familiar with multiple languages, platforms, and technologies.
- Multi-artifact consistency is a big concern because you are separating the overall project into three different parts.

4. It would be helpful if you showed an example where you make a change in one component but the impact is not seen in other parts.

Answer:

Sure. Suppose, you want to change the display order of students in the view component when you add a student to the current list. (For simplicity, I am not making any change in the “remove student” logic). So, try this code:

```
//Original Structure-Showing newly added student at the bottom of list
//foreach (string student in studentList)
//{
//showStudentsListView.Items.Add(student);
//}
```

```
//New Structure-Showing newly added student at the top of list  
studentList.Reverse();  
foreach (string student in studentList)  
{  
    showStudentsListView.Items.Add(student);  
}
```

Now when you run the application, you will see that in your form the newly added student's name appears at the top of the list.

Figure 26-14 shows the output when the application is loaded with default values.

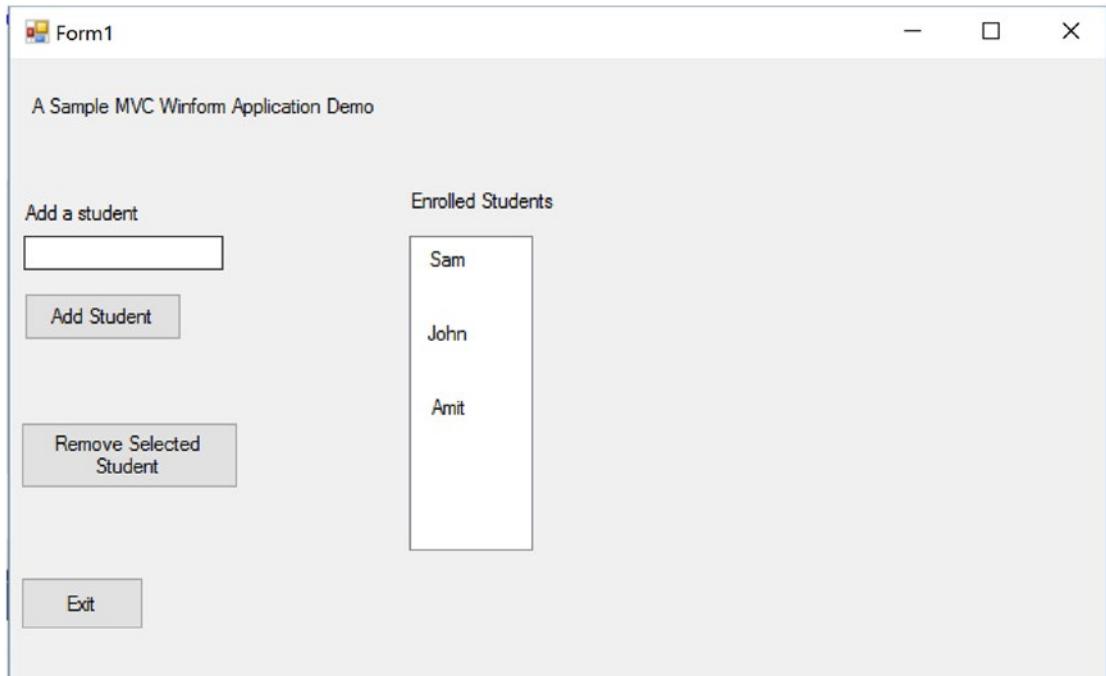


Figure 26-14. Application loaded with default values

CHAPTER 26 MVC PATTERN

Figure 26-15 shows the output when adding a student named Bob.

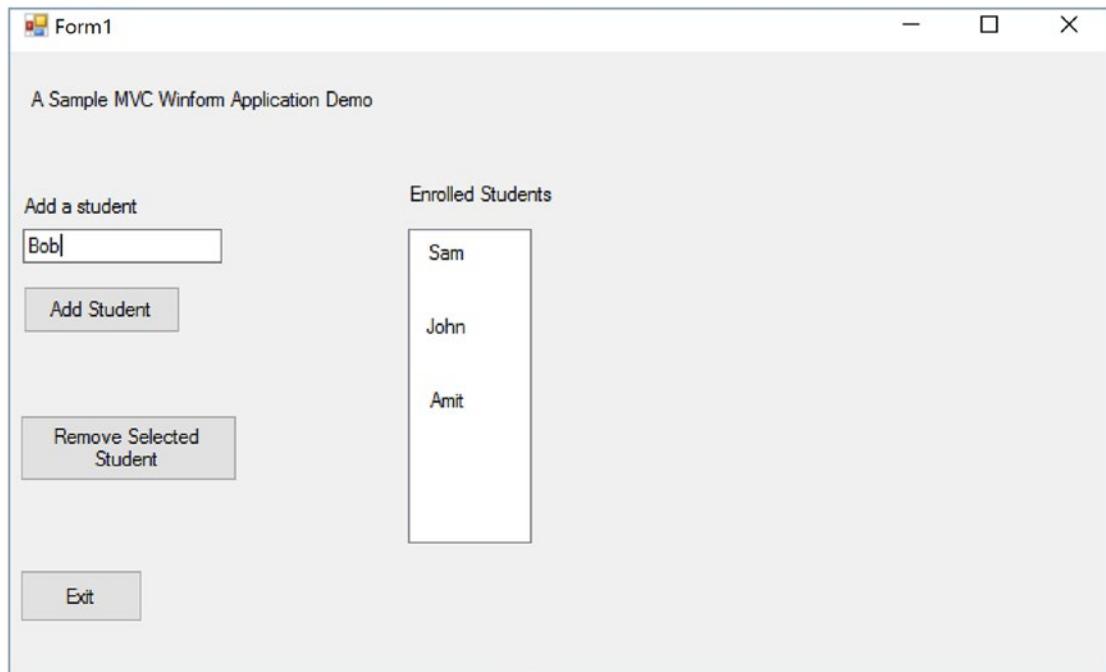


Figure 26-15. Prior to add a student “Bob” to enrolled student list

Figure 26-16 shows the output when you click the Add Student button.

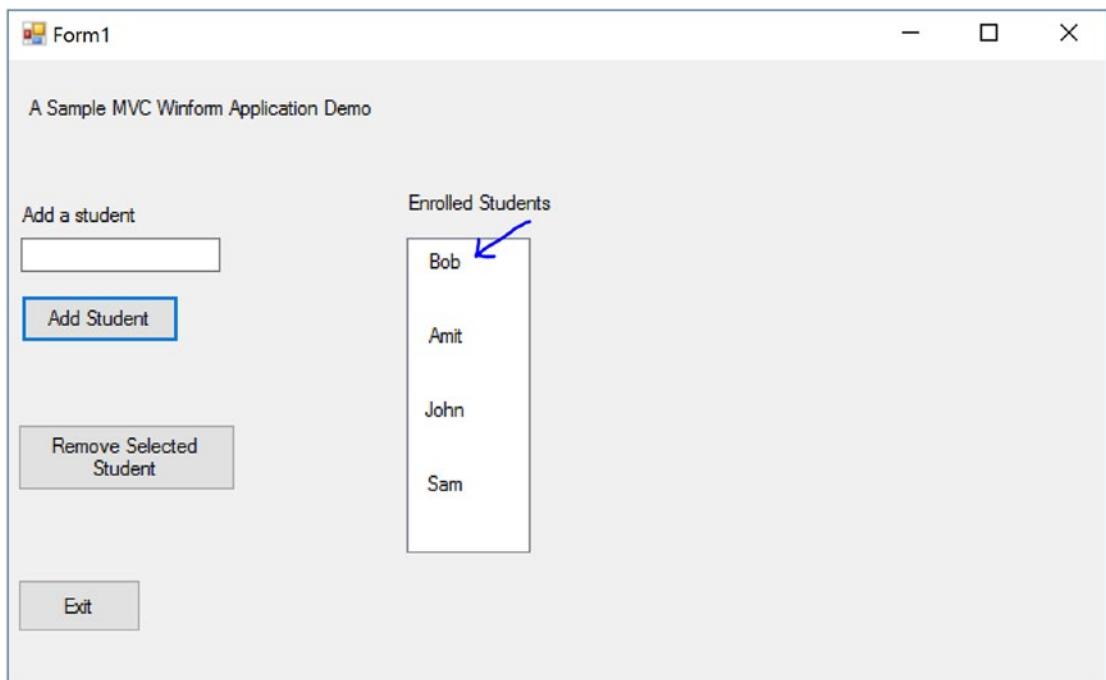


Figure 26-16. Immediate output after adding a student “Bob” to the enrolled student list

PART III

Final Thoughts on Design Patterns

CHAPTER 27

Criticisms of Design Patterns

In this chapter, I present some of the criticisms of design patterns. Reading about the criticisms can offer you some real value. If you think critically about patterns before you design your software, you can predict your “Return on Investment” to some degree.

Design patterns basically help you benefit from other people’s experience. I term this as *experience reuse*. You learn how they solved challenges, how they tried to adapt to new behaviors in their systems, and so on. A pattern may not perfectly fit into your work, but if you concentrate on the best practices as well as the problems of a pattern at the beginning, you are more likely to make a better invention.

So, here are some of the criticisms of patterns:

- Many of these patterns are closely related to each other. So, these closely related pattern may create confusions in developers mind. Also, there are always pros and cons associated with each pattern. (I discuss some of them in the “Q&A Sessions” section at the end of each chapter.) In some cases, once a pattern is applied, maintaining the pattern may become more difficult.
- The pattern that is giving you satisfactory results today can be a big burden to you in the future because the software industry is continuously changing.
- It is unlikely that the infinite number of requirements can be well designed with these finite number of design patterns.
- Designing a software is basically an art, and there is no definition or criteria for the best art.

- Design patterns give you the ideas but not the implementations (like libraries or frameworks). We all know human minds are unique. Each engineer may have their own preference for implementing a similar concept, and that can create chaos in a team.
- Consider a simple example. Patterns encourage people to code to a super type (abstract class/ interface). But for a simple application where you know that there are no upcoming changes or the application is created for demo purposes only, this rule may not make much sense.
- In a similar way, in some small applications, you may find that enforcing the rules of design patterns is increasing your code size and maintenance costs.
- Erasing the old and adapting the new is not always easy. For example, when you first learned about inheritance, you were probably excited. You probably wanted to use it in many ways and were seeing only the benefits from the concept. But later when you started experimenting with design patterns, you started learning that in many cases, compositions are preferred over inheritance. This shifting of gears is not easy.
- Design patterns are based on some key principles, and one of them is to identify the code that may vary and then separate it from the rest of the code. This sounds good from a theoretical perspective. But in real-world implementations, who guarantees that your judgment is perfect? The software industry always changes, and it needs to adapt to new requirements/demands.
- Some of the patterns are already integrated into modern-day languages. Instead of implementing the pattern from scratch, you can use the built-in support of the language constructs (for example, consider the Iterator pattern in C#).
- The inappropriate use of patterns can lead to anti-patterns. For example, an inappropriate use of the Mediator pattern can lead to the God Class anti-pattern. I will briefly cover anti-patterns in Chapter 28.

- Many people believe that the concepts of design patterns simply indicate that a programming language may need some additional features. So, patterns will have less significance with the increasing capabilities of modern-day programming languages. For example, Wikipedia says that famous computer scientist Peter Norvig believes that 16 out of the 23 patterns in the GoF design patterns are simplified or eliminated via direct language support in Lisp or Dylan. You can see some similar thoughts here: https://en.wikipedia.org/wiki/Software_design_pattern

Q&A Session

1. Is there any catalog for these patterns?

Answers:

I started with the GoF's 23 design patterns and then discussed 3 other patterns in this book. The GoF's famous catalog is considered the most fundamental pattern catalog.

But definitely there are many other catalogs that focus on particular domains.

2. Why are you not covering other patterns?

Answers:

These are my beliefs:

- Computer science will keep growing, and we will keep getting new patterns.
- If you are not familiar with the fundamental patterns, you cannot evaluate the true needs of the remaining or upcoming patterns. For example, if you know MVC well, you can see how it is different than Model-View-Presenter (MVP) and understand why MVP is needed.

- This book is already fat. A detailed discussion of each pattern would need many more pages, which would make the size of the book too big to digest.

So, in this book, I focused on the fundamental patterns that are still relevant in today's programming world.

3. I often see the term *force* in the description of design patterns. What does it mean?

Answers:

Force is a criterion on which developers justify their developments. Broadly, your target and current constraints are two important parts of your force. Therefore, when you develop your application, you can justify the development with these parts.

4. In various forums, I have seen that people are fighting about a pattern definition and say something like this: "A pattern is a proven solution to a problem in a context." What does this mean?

Answers:

This is a simple and easy-to-remember definition of what a pattern is. But simply breaking it down into three parts (problem, context, and solution) is not enough.

As an example, suppose you are going to the airport and you are in a hurry. Suddenly, you discover that you have left your boarding pass at home. Let's analyze the situation:

Problem: You need to reach the airport on time.

Context: You left your boarding pass at home.

Solution: Turn back, go at a high speed, and rush toward home to get the boarding pass.

This solution may work one time, but can you apply the same procedure repeatedly? You know the answer. It is not an intelligent solution because it depends on how much time you have to collect the pass from home and go back to the airport. It also depends on the current traffic on the road and many other factors. So, even if you can get the success for one time, you may want to prepare yourself for a better solution for a similar situation in future.

So, try to understand the meaning, intent, context, and so on, to understand a pattern clearly.

- 5. Sometimes I am confused by seeing similar UML diagrams for two different patterns. Also, I am further confused by the classification of the patterns in many cases. How can I overcome this?**

Answers:

This is perfectly natural. The more you read and the more you try to understand the intent behind the designs, the more the distinctions among them will make sense to you.

CHAPTER 28

Anti-patterns

The discussion of design patterns is not complete without talking about anti-patterns. So, this chapter gives a brief overview of them.

What Is an Anti-pattern?

In real-world application development, sometimes you will implement an approach that seems attractive at the beginning but in the long run causes problems. For example, you may choose to implement a certain technology to get a quick fix to meet a delivery deadline. But if you aren't aware of potential pitfalls of the technology, you will pay a price for your choice.

Anti-patterns alert you to the common mistakes you can encounter when developing so that you can take precautionary measures. The proverb "Prevention is better than a cure" very much suits in this context.

POINTS TO REMEMBER

Anti-patterns alert you to common mistakes by describing how attractive approaches can make your life difficult in the long run. At the same time, they suggest alternate solutions that may seem tough or ugly at the beginning but ultimately help you build a better solution. In short, anti-patterns identify problems with established practices.

Q&A Session

1. How are anti-patterns related to design patterns?

Answer:

With design patterns, you reuse the experience of others who came before you. But when you start blindly using those concepts too much without thinking critically about what you're doing, you can create problems in your programs. As a result, over a period of time, you may find that your return on investment decreases while your maintenance costs increase.

2. A design pattern may turn into an anti-pattern. Is this understanding correct?

Answer:

Yes. If you apply a design pattern in the wrong context, it can cause more trouble than the problem it solves, and eventually it will turn into an anti-pattern. So, before you start, understanding the nature and context of the problem you're solving is important.

3. Software developers alone should study these anti-patterns. Is this understanding correct?

Answer:

No. The usefulness of an anti-pattern is not limited to developers; it is also applicable to managers and technical architects.

4. Even if you do not get much benefit from anti-patterns now, they can help you to adapt new features easily with less maintenance costs in future. Is this understanding correct?

Answer:

Yes.

5. What are the probable causes of anti-patterns?

Answer:

The causes of anti-patterns can arise from various sources and mind-sets. These are a few examples of what someone might say or be thinking:

- “We need to deliver the product as soon as possible.”
- “Currently we do not need to analyze the impact.”
- “I am an expert of reuse. I know design patterns very well.”
- “We will use the latest technologies and features to impress our customers. We do not need to care about legacy systems.”
- “More complicated code will reflect my expertise in the subject.”

6. What are some symptoms of anti-patterns?

Answer:

In object-oriented programming, the most common symptom is that your system cannot integrate a new feature easily. Another example is that maintenance costs are increasing. Or you might notice that you have lost the power of key object-oriented features such as inheritance, polymorphism, and so on.

In addition, you may notice the following symptoms:

- Use of global variables
- Code duplication
- Limited/no reuse of code
- One big class (a God class)
- The presence of a large number of parameterless methods

7. Can you give some common anti-pattern examples?

Answer:

The anti-pattern catalog (<http://wiki.c2.com/?AntiPatternsCatalog>) maintains a list of anti-patterns. The following are the concepts behind some of them:

- *Overuse of patterns:* Developers may try to use a pattern at any cost, regardless of whether it is appropriate.
- *God class:* A big object tries to control almost everything with many unrelated methods. An inappropriate use of the Mediator pattern may end up with this anti-pattern.
- *Not invented here:* Here's the thinking on this one: "I am a big company, and I want to build everything from scratch. Though there is a library available that was developed by some small company, I won't use it. I will make everything on my own, and once something is developed, I'll use my brand value to announce, 'Hey Guys, the ultimate library has just launched.'"
- *Zero means null:* It may appear when developers ignores an optional field due to laziness, optimism or some other forces. As a result, when your application really needs a zero, it may not get it (or, use it). Another common variation can be seen when a programmer uses -1, 999, or anything like that to represent an inappropriate integer value. Another erroneous use case is when a user treats 09/09/9999 as a null date in an application. In these cases, if the user needs to have the numbers -1, 999, or the date 09/09/9999, he will not be able to use those.
- *Golden hammer:* In this example, Mr. X believes that technology T is always the best. If Mr. X needs to develop a new system (that demands new learning), he still prefers T even if it is inappropriate. He thinks, "I do not need to learn anything new if I can somehow manage it with T."

Note Again, you can refer to the anti-pattern catalog at <http://wiki.c2.com/?AntiPatternsCatalog> to learn more.

8. What is the remedy if you detect an anti-pattern?

Answer:

You may need to follow a refactored and better solution. For example, in the case of the previous golden hammer example, you may try to educate Mr. X through some proper training. Or, in the case of zero means null, you can use an additional Boolean variable to indicate the null value properly.

9. What do you mean by *refactoring*?

Answer:

In the coding world, the term *refactoring* means improving the design of existing code without changing the external behavior of the system/application. This process helps you to create more readable code. At the same time, these newly designed code should be more adaptable to new requirements (or, change requests), and they should be more maintainable.

CHAPTER 29

Sealing the Leaks in Your Applications

If a computer application allocates resources continuously but does not deallocate them when they are not needed, you will see an impact eventually. For example, machines will become slower, or in the worst case, they can crash, and you will need to reboot the system. This scenario is described as a *memory leak*. How quickly you notice it depends on the *leaking rate* of your application.

Even if you implement a design pattern properly in your application, if you do not guard against memory leakage, you may end up with low-quality software.

Memory leakage is a large topic. In this chapter, you will experiment with some of the important aspects of it. Once you are aware of them, you can avoid some of the common mistakes that result in memory leaks.

To understand the concept, I will start by explaining how garbage collection (GC) works. Managing memory is an important concern for programmers, and .NET tries to make their life easier by taking on some of the responsibilities. Specifically, it can clear objects that have no use after a particular point. In programming terms, you can call these objects *dirty* objects or *unreferenced* objects.

A garbage collector program runs in the background as a low-priority thread and keeps track of these dirty objects. Also, the .NET runtime can invoke this program on a regular interval to remove these unreferenced or dirty objects from memory.

However, there is a catch. Some objects require special teardown code to release resources. The following are some common examples:

- You open a file but forget to close it.
- You register some events but forget to unregister them.
- You are dealing with unmanaged objects, with locking mechanisms, or with operating system handles in your program but you forgot to release those resources properly.

In C#, when programmers put effort into cleaning up (or *releasing*) memory, they are trying to *dispose* of the objects. But if the Common Language Runtime (CLR) takes on the responsibility of releasing resources, then the garbage collector is doing its job.

POINTS TO REMEMBER

Programmers can release resources by explicitly disposing of objects, or the CLR can release them automatically through the GC mechanism.

How Garbage Collection Works

A generational garbage collector is used to collect short-lived objects more frequently than longer-lived objects. Here we have three generations: 0, 1, and 2. Short-lived objects are stored in generation 0. Longer-lived objects are pushed into the higher generations, either 1 or 2. The garbage collector works more frequently in the lower generations than in the higher ones.

Once you create an object, it resides in generation 0. When generation 0 is filled up, the garbage collector is invoked. The objects that survive the garbage collection in the first generation are promoted into the next-higher generation, generation 1. The objects that survive garbage collection in generation 1 move to the next-highest generation, generation 2.

Note Garbage collection works in three different phases, and in general, a garbage collector can be invoked in three different cases.

The following are the three different phases of garbage collection:

1. *Marking phase*: In this phase, the live objects are marked or identified.
2. *Relocating phase*: In this phase, the garbage collector updates the references of those objects that will be compacted in phase 3.
3. *Compacting phase*: Here you reclaim memory from the dead/unreferenced objects, and the compaction operation is performed on the live objects. The garbage collector moves the live objects (the ones that have survived so far) to the older end of the segment.

Basically, the garbage collector starts its job from the root object reference and traces an object graph to identify the reachable objects. Once the process is finished, the unmarked objects are subject to garbage collection. Then the memory of unmarked objects with no destructor is immediately reclaimed by the garbage collector. If you are familiar with destructors (or, `Finalize()` methods), you know that objects with destructors are placed in a finalization queue.

After the marking phase when the garbage collector identifies the dead objects, the garbage collector checks whether they have destructors associated with them (in other words, the garbage collector checks the finalization queue). If it sees that there is a finalization method associated with an object, it flushes the object into the freachable (pronounced as “F-reachable”) queue.

When the freachable queue is not empty, a separate thread wakes up and keeps cleaning up the entries by executing the corresponding finalization methods.

On the next pass, the garbage collector sees that these objects are ready to be garbage collected (their finalization methods were executed already, so the freachable queue is empty), and it reclaims the memory.

These are three different times when the garbage collector is invoked.

- *Case 1:* The memory consumption rate of the program is much higher than the releasing rate. As an obvious impact, at some point, the available memory will be low. It can get a notification from the host or the operating system.
- *Case 2:* The allocated objects (in the managed heap) surpass a defined threshold limit.
- *Case 3:* The `System.GC.Collect()` method is invoked.

The `GC.Collect()` method can be used to force the garbage collection mechanism. There are many overloaded versions of this method. In the following example, I have used `GC.Collect(Int32)`, which can force an immediate garbage collection from generation 0 through the specified generation. I bring the garbage collector into action by invoking `System.GC.Collect()` in Case 3.

Demonstration 1

Here's the implementation for the first demonstration:

```
using System;

namespace GarbageCollectionEx1
{
    class MyClass
    {
        private int myInt;
        //private int myInt2;
        private double myDouble;

        public MyClass()
        {
            myInt = 25;
            //myInt2 = 100;
            myDouble = 100.5;
        }
    }
}
```

```
public void ShowMe()
{
    Console.WriteLine("MyClass.ShowMe()");
}
public void Dispose()
{
    GC.SuppressFinalize(this);
    Console.WriteLine("Dispose() is called");
    Console.WriteLine("Total Memory:" + GC.GetTotalMemory(false));
}
~MyClass()
{
    Console.WriteLine("Destructor is Called..");
    Console.WriteLine("After this destruction total Memory:" +
        GC.GetTotalMemory(false));
    //To catch the output at end, we are putting some sleep
    //Or, you can run it from command prompt to avoid the following
    //sleep
    System.Threading.Thread.Sleep(60000);
}
class Program
{
    public static void Main(string[] args)
    {
        Console.WriteLine("/**Exploring Garbage Collections.**");
        try
        {
            Console.WriteLine("Maximum Generations of GC:" +
                GC.MaxGeneration);
            Console.WriteLine("Total Memory:" +
                GC.GetTotalMemory(false));
            MyClass myOb = new MyClass();
            Console.WriteLine("myOb is in Generation : {0}",
                GC.GetGeneration(myOb));
        }
    }
}
```

```
Console.WriteLine("Now Total Memory is:{0}",  
    GC.GetTotalMemory(false));  
Console.WriteLine("Collection occurred in 0th  
Generation:{0}", GC.CollectionCount(0));  
Console.WriteLine("Collection occured in 1th  
Generation:{0}", GC.CollectionCount(1));  
Console.WriteLine("Collection occured in 2th  
Generation:{0}", GC.CollectionCount(2));  
  
//myOb.Dispose();  
  
GC.Collect(0);//will call generation 0  
Console.WriteLine("\n After GC.Collect(0)");  
  
Console.WriteLine("Collection occured in 0th  
Generation:{0}", GC.CollectionCount(0));//1  
Console.WriteLine("Collection occured in 1th  
Generation:{0}", GC.CollectionCount(1));//0  
Console.WriteLine("Collection occured in 2th  
Generation:{0}", GC.CollectionCount(2));//0  
Console.WriteLine("myOb is in Generation : {0}",  
    GC.GetGeneration(myOb));  
Console.WriteLine("Total Memory:" +  
    GC.GetTotalMemory(false));  
  
GC.Collect(1);//will call generation 1 with 0  
Console.WriteLine("\n After GC.Collect(1)");  
  
Console.WriteLine("Collection occured in 0th  
Generation:{0}", GC.CollectionCount(0));//2  
Console.WriteLine("Collection occured in 1th  
Generation:{0}", GC.CollectionCount(1));//1  
Console.WriteLine("Collection ccured in 2th  
Generation:{0}", GC.CollectionCount(2));//0  
Console.WriteLine("myOb is in Generation : {0}",  
    GC.GetGeneration(myOb));  
Console.WriteLine("Total Memory:" +  
    GC.GetTotalMemory(false));
```

```

        GC.Collect(2); //will call generation 2 with 1 and 0
        Console.WriteLine("\n After GC.Collect(2)");

        Console.WriteLine("Collection occurred in 0th
        Generation:{0}", GC.CollectionCount(0)); //3
        Console.WriteLine("Collection occurred in 1th
        Generation:{0}", GC.CollectionCount(1)); //2
        Console.WriteLine("Collection occurred in 2th
        Generation:{0}", GC.CollectionCount(2)); //1
        Console.WriteLine("myOb is in Generation : {0}",
        GC.GetGeneration(myOb));
        Console.WriteLine("Total Memory:" + GC.GetTotalMemory(false));

    }
    catch (Exception ex)
    {
        Console.WriteLine("Error:" + ex.Message);
    }

    Console.ReadKey();
}
}
}

```

Output

Here's the output:

```

***Exploring Garbage Collections.***
Maximum Generations of GC:2
Total Memory:38092
myOb is in Generation : 0
Now Total Memory is:38092
Collection occurred in 0th Generation:0
Collection occurred in 1th Generation:0
Collection occurred in 2th Generation:0

After GC.Collect(0)
Collection occurred in 0th Generation:1

```

CHAPTER 29 SEALING THE LEAKS IN YOUR APPLICATIONS

Collection occurred in 1th Generation:0

Collection occurred in 2th Generation:0

myOb is in Generation : 1

Total Memory:40264

After GC.Collect(1)

Collection occurred in 0th Generation:2

Collection occurred in 1th Generation:1

Collection occurred in 2th Generation:0

myOb is in Generation : 2

Total Memory:40368

After GC.Collect(2)

Collection occurred in 0th Generation:3

Collection occurred in 1th Generation:2

Collection occurred in 2th Generation:1

myOb is in Generation : 2

Total Memory:40420

Destructor is Called..

After this destruction total Memory:48612

Note the following:

- I have made some of the lines bold to draw your attention to those places.
- The total memory count will differ depending on which system the code is running on.

Analysis

If needed, review the theory of garbage collection described earlier in the chapter to understand how the garbage collection happened. You will notice the following important characteristics:

- In this example, I have not called the Dispose() method. I wanted the memory to be collected by the finalizer method (the destructor).
- When you call generation 2, the other generations (-0 and 1) are also called.
- The object created was originally placed in generation 0.

Q&A Session 1

1. How can you call the destructor?

Answer:

You cannot call the destructor. The garbage collector takes care of that responsibility.

2. What is a managed heap?

Answer:

When the CLR initializes the garbage collector, it allocates a segment of memory to store and manage the objects. This memory is termed the *managed heap*.

In general, `Finalize()` (or the destructor of the object) is invoked to clean up the memory. Hence, you can provide a destructor to free up some unreferenced resources that are held by your objects. But in that case, you need to override the `Object` class's `Finalize()` method.

3. At what point does the garbage collector call the `Finalize()` method?

Answer:

You never know. It may call it instantly when an object is found with no references or later when the CLR needs to reclaim some memory.

You can force the garbage collector to run at a given point by calling `System.GC.Collect()`, which has many overloaded versions. (I have shown one usage by invoking `GC.Collect(Int32)`).

4. Why is compaction necessary?

When the garbage collection removes all the intended objects (in other words, those objects with no references) from the heap, the heap will contain scattered objects. For simplicity, you can assume Figure 29-1 is your heap, and after the garbage collector's cleanup operation it looks like Figure 29-2 (the white blocks represent free/available blocks).

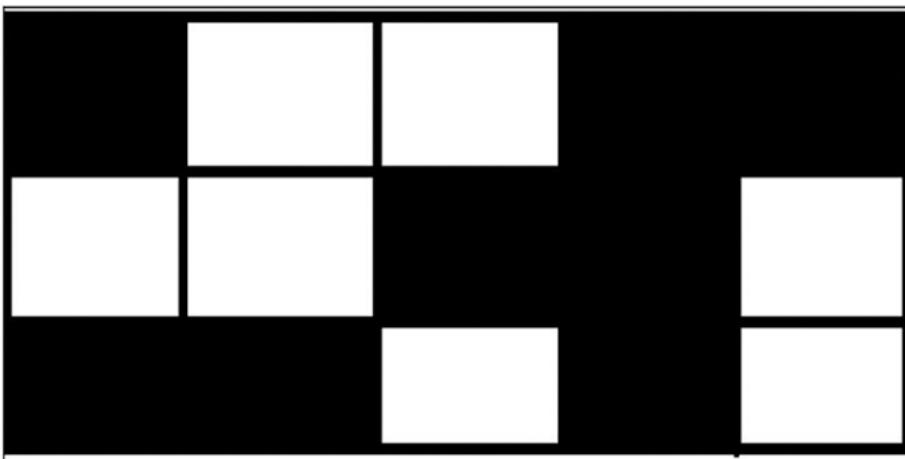


Figure 29-1. Your original heap

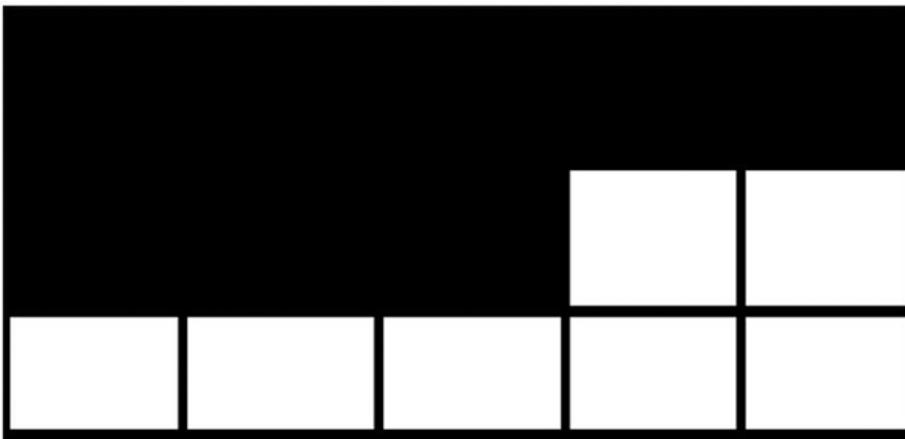


Figure 29-2. The heap after compaction

You can see, in this situation, that if you need to allocate five contiguous memory blocks in your heap, you cannot allocate them even though collectively you have enough spaces to hold them. To deal with this kind of situation, the garbage collector needs to apply the compaction technique, which moves all the remaining objects (live objects) to one end to form one continuous block of memory. After this compaction, your heap looks like Figure 29-2.

Now you can easily allocate five contiguous blocks of memory in the heap. So, the compaction technique can help you avoid memory fragmentation. Also, when you allocate memory for new objects at the heap end, there is no need to maintain a list to identify the free memory segments.

Note In this way, a managed heap is different from an old unmanaged heap. Here you do not need to iterate through a linked list of addresses to find spaces for new data. Rather, you can simply use the heap pointer, so instantiating an object under .NET is faster. Once the compaction is done, objects generally stay in the same area, so accessing them also becomes easier and faster (because page swapping occurs less). This is why Microsoft believes that although a compaction operation is costly, it is worthwhile.

5. When should you invoke `GC.Collect()`?

Answer:

I already mentioned that invoking the garbage collector is generally treated as a costly operation. But in some special scenarios, when you invoke the garbage collector, you will gain some significant benefits. Such an example may arise after you dereference a large number of objects in your code.

Another common scenario is observed when you try to find memory leaks through some automated tests. For example, you may execute a test repeatedly, and after each execution of these tests, you may try to gather different counters to analyze the memory growth. To get the correct counters, you may need to call `GC.Collect()` at the beginning of each operation.

(I will discuss the memory leak analysis shortly.)

6. Can you reclaim memory at some specified point when your application is running?

Answer:

The .NET Framework provides a special interface called `IDisposable` for this. Let's see what the Visual Studio IDE tells you:

```
using System.Runtime.InteropServices;

namespace System
{
    //
    // Summary:
    // Provides a mechanism for releasing unmanaged resources. To
    // browse the .NET Framework
    // source code for this type, see the Reference Source.
    [ComVisible(true)]
public interface IDisposable
{
    //
    // Summary:
    // Performs application-defined tasks associated with
    // freeing, releasing, or resetting
    // unmanaged resources.
    void Dispose();
}
}
```

So, you need to implement this `IDisposable` interface, and as an obvious action, you need to override its `Dispose()` method. Programmatically, when a developer wants to free up resources, this is the best practice. Another key advantage with this kind of approach is that you know when your programs are going to free up unreferenced resources.

POINTS TO REMEMBER

When you implement the `IDisposable` interface, you assume that the programmer will call the `Dispose()` method correctly. Some experts still suggest that as a precautionary measure, you should implement a destructor also. This approach is useful if somehow `Dispose()` is not called (for example, an exception occurred in between and the expected control flow breaks). I agree that this kind of dual implementation can make sense in real-world programming.

C# provides some special support in this context. You can use the `using` statement to reduce your code size and make it more readable. It is used as a syntactic shortcut for the `try/finally` block.

7. **If you use a destructor, it will be called for sure. But you do not have control over that invocation. On the other hand, if you implement the `IDisposable` interface, you know when the call to the `Dispose()` method will happen, but somehow this call can be skipped. So, I am confused. Which of these two approaches should be used?**

Answer:

I already mentioned that programmers can ensure that `Dispose()` calls the `using` statements. You can also place the cleanup code in a `finally` block if there is a possibility of encountering an exception. But in general, I try to follow the experts' advice

who vote for the dual implementation where you try to call the `Dispose()` method from a destructor like the following:

```
~MyClass()
{
    Dispose();
}
...
```

This approach guarantees that the `Dispose()` method will be called in all scenarios (even if the program terminates unexpectedly).

Microsoft suggests a basic dispose pattern as follows:

```
//Microsoft's Basic Dispose Pattern-
public class MyDisposableClass : IDisposable
{
    //private "ResourceType" resource; // handle to a resource

    public MyDisposableClass()
    {
        // allocates the resource
        //this.resource = ...
    }

    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    protected virtual void Dispose(bool disposing)
    {
        if (disposing)
        {
            if (resource != null) resource.Dispose();
        }
    }
}
```

Let's analyze this.

The Boolean parameter `disposing` checks whether the method was called from the finalizer. The `Dispose(bool)` implementation should check this parameter before accessing other reference objects. Such objects should be accessed only when the method is called from the `IDisposable.Dispose` implementation (in other words, when `disposing=true`). If the method is invoked from the finalizer (in other words, when `disposing=false`), you should prevent the access to other objects. This is because the objects are finalized in an unpredictable order, so these objects or their dependents might already have been finalized.

You can refer to the following articles for a deeper understanding:

<https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/dispose-pattern>

<https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/implementing-dispose>

8. What is purpose of the statement `GC.SuppressFinalize(this);` in this context?

Answer:

It says, “Hi, CLR, you do not need to call the finalizer for this object because I have done the job for you already.” This statement should run only after `Dispose(true)` executes successfully. The key aim is to recollect the memory as early as possible.

9. What is purpose of the statement `protected virtual void Dispose(bool disposing);` in this context?

Answer:

To make a centralized location to release the resources (basically unmanaged resources), you keep this method. The subclasses may need to override this method in the future (instead of overriding `Finalize()` again).

- 10. The CLR supports automatic memory management. Still, there are so many concerns about releasing memory. Can you please elaborate?**

Answer:

This statement is applicable in the managed context such as when you allocate memory with the new operator. But there are a variety of system resources. Managed memory is just a subset of that variation. Other parts are generally referred to as unmanaged memory, and the garbage collector cannot help you directly there. So, it is the programmer's job to release those resources.

Still, the CLR can help you with its `System.Object.Finalize()` method. This is a virtual method, and a type can override this method. (The types that override this method are called *finalizable types*.) The garbage collector calls this method prior to reclaiming the memory.

- 11. You can simply override this finalizer to reclaim the memory. Is this understanding correct?**

Answer:

Depending solely on finalizers has some drawbacks, as listed here:

- You do not know exactly when the memory will be released. In some situations, you may need to reclaim memory as early as possible.
- When the CLR calls the finalizer, it suspends the collection of the object's memory (and the memory of all objects that are referred by it) until the next round of garbage collection. In other words, the garbage collector runs its own thread and executes at some specific time. But when it runs, other threads that are running in parallel will halt. This is needed because the garbage collector may need to move the objects and update those references. So, this waiting time is not always acceptable.

Quiz

- Will the following code compile?

```
class MyClass
{
    public MyClass()
    {
        Console.WriteLine("Allocating some resources");
    }
    public void SomeMethod()
    {
        Console.WriteLine("MyClass.SomeMethod()");
    }
    protected override void Finalize()
    {
        Console.WriteLine("Try to release resources");
    }
}
```

Answer:

No. The output shown in Figure 29-3 is self-explanatory.

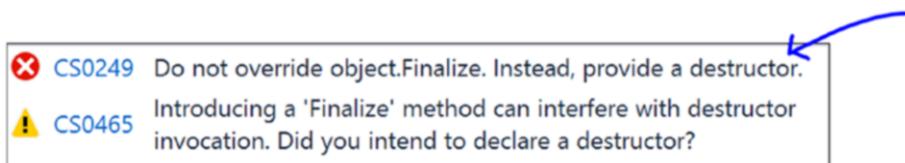


Figure 29-3. Quiz output

Now replace the erroneous method with the destructor as follows:

```
class MyClass
{
    public MyClass()
    {
        Console.WriteLine("Allocating some resources");
    }
}
```

```

public void SomeMethod()
{
    Console.WriteLine("MyClass.SomeMethod()");
}
//protected override void Finalize()
//{
//    Console.WriteLine("Try to release resources");
//}
~MyClass()
{
    Console.WriteLine("In Destructor: Try to release resources");
}
}

```

Compile your code. Then notice the IL code, as shown in Figure 29-4.

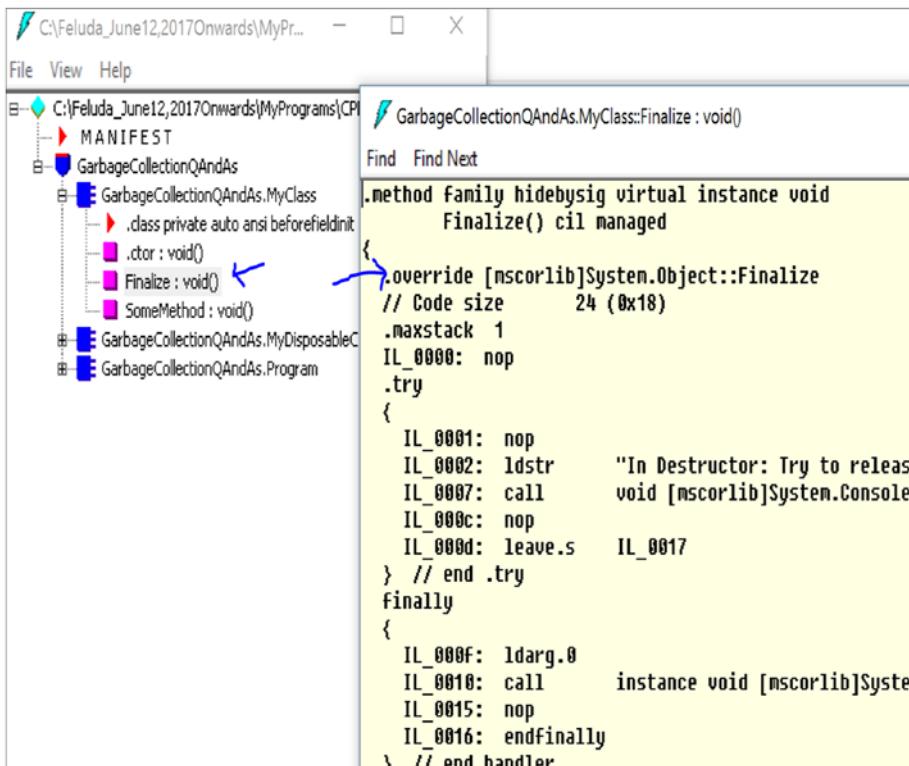


Figure 29-4. IL code

The C# compiler converted the destructor to the `Finalize()` method. In other words, the destructor is equivalent to `Finalize()` in C#. So, you can put your intended code in the destructor.

Consider the following three programs with their outputs to understand the key distinctions between the use of `Dispose()` and the destructor and to understand how you can reclaim the memory in C#.

Quiz

1. Can you predict the output of the following code?

```
using System;
namespace GarbageCollectionQuiz1
{
    class MyClass : IDisposable
    {
        public void DoSomething()
        {
            Console.WriteLine("MyClass.DoSomething");
        }
        public void Dispose()
        {
            GC.SuppressFinalize(this);
            Console.WriteLine("MyClass.Dispose() is called");
        }
        ~MyClass()
        {
            Console.WriteLine("MyClass.Destructor is Called..");
            System.Threading.Thread.Sleep(5000);
        }
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine(" ***Quiz on Garbage
Collections***");
        MyClass myOb = new MyClass();
        myOb.DoSomething();
        myOb.Dispose();
        Console.ReadKey();
    }
}
```

Answer:

Quiz on Garbage Collections
MyClass.DoSomething
MyClass.Dispose() is called

Notice that the Dispose() method is called, but the destructor of the object is not called.

Quiz

Now comment out the two lines shown in Figure 29-5.

```

public void Dispose()
{
    //GC.SuppressFinalize(this); ←
    Console.WriteLine("MyClass.Dispose() is called");
}
~MyClass()
{
    Console.WriteLine("MyClass.Destructor is Called..");
    System.Threading.Thread.Sleep(5000);
}
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("*** Quiz on Garbage Collections***");
        MyClass myOb = new MyClass();
        myOb.DoSomething();
        //myOb.Dispose(); ←
        Console.ReadKey();
    }
}

```

Figure 29-5. Commenting out two lines

2. Can you predict the output of the following code?

Answer:

The destructor method is invoked in this case.

Quiz on Garbage Collections

MyClass.DoSomething

MyClass.Destructor is Called..

Note that when the destructor is in action, if you want to hold the output for a long time, you can simply use the developer command prompt, as shown in Figure 29-6.

```
C:\Program Files (x86)\Microsoft Visual Studio\2017\Community>"C:\Feluda_June12, 2017Onwards\MyPrograms\CPPprogs\ConsoleApplication1\GarbageCollectionQuiz1\obj\Debug\GarbageCollectionQuiz1.exe"
*** Quiz on Garbage Collections***
 MyClass.DoSomething
 MyClass.Destructor is Called..
```

Figure 29-6. Developer command prompt to hold the output

Quiz

Keep commenting out the line `//GC.SuppressFinalize(this)`, but invoke the `Dispose()` method, as shown in Figure 29-7.

```
public void Dispose()
{
    //GC.SuppressFinalize(this); ←
    Console.WriteLine("MyClass.Dispose() is called");
}
~MyClass()
{
    Console.WriteLine("MyClass.Destructor is Called..");
    System.Threading.Thread.Sleep(15000);
}
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine(" *** Quiz on Garbage Collections***");
        MyClass myOb = new MyClass();
        myOb.DoSomething();
        myOb.Dispose();
        Console.ReadKey();
    }
}
```

Figure 29-7. Invoking `Dispose()`

3. Can you predict the output?

Answer:

Quiz on Garbage Collections

MyClass.DoSomething

MyClass.**Dispose()** is called

MyClass.**Destructor** is Called..

Notice that both the `Dispose()` method and the destructor of the object are called because you are not restricting the garbage collector to reclaiming the memory occupied by the object with `SuppressFinalize(...)`.

Understanding Memory Leaks

So far, you know that managing memory is an important concern when you build the architecture of an application. You may implement design patterns effectively, but if you do not care about managing memory, you will not end up with high-quality software.

At the beginning of the chapter, I said that how quickly you notice the impact of memory leaks depends on the *leaking rate* of your application. I'll now elaborate on that with a simple example.

Suppose you have an online application where users need to fill in some data and then click a Submit button. Now assume that the developers did not implement a proper dispose pattern for the application. Because of this poor design, when a user clicks the Submit button, the application starts leaking 512 bytes per click. You probably will not notice any performance degradation after just a few clicks. But what happens if thousands of online users are using the application simultaneously? If 100,000 users click the Submit button, you will eventually lose 48.8MB of memory, 10 million clicks will lead to a loss of 4.76 GB, and so on.

In short, even if your application or program is leaking a small amount of data per execution, it is quite obvious that you will see some kind of malfunctioning over a period of time. For example, you may notice that your device is crashing with a `System.OutOfMemoryException` or operations in the device become so slow that you need to restart your application often.

In an unmanaged language like C++, you need to deallocate the memory when the intended job is done. Otherwise, over a period of time, the impact of memory leaks will be huge. As discussed, in a managed environment, the CLR's garbage collector provides a tremendous help to programmers. Still, there are cases that you need to handle with care. Otherwise, you may notice the impact of memory leaks.

How can you detect leaks? Many tools are available to do this. For example, Windbg.exe is a common tool to find memory leaks in large applications. In addition, you can use some other graphical tools such as Microsoft's CLR profiler, SciTech's Memory Profiler, Red Gate's ANTS Memory Profiler, and so on, to find leaks in your system. Many organizations even have their own memory leak tools to detect and analyze leaks.

In the latest editions of Visual Studio (I am using Visual Studio Community 2017), you can use the Diagnostic tool to detect and analyze memory leaks. It is easy to use, and you can take different memory snapshots at different times. Markers in the tool can indicate the garbage collector activities. The real power of this tool is that you can analyze the data in real time while the debugging session is active. The spikes in the graph can help draw the programmer's attention immediately.

Note the following:

- You can learn more about the Diagnostic tool at <https://blogs.msdn.microsoft.com/devops/2015/01/16/diagnostic-tools-debugger-window-in-visual-studio-2015/>.
- In Visual Studio 2017, you can open the Diagnostic tool by selecting Debug ► Windows ► Show Diagnostic Tools (or by pressing Ctrl+Alt+F2).

Demonstration 2

Here's the demonstration code:

```
using System;
using System.Collections.Generic;

namespace MemoryLeakWithSimpleEventDemo
{
    public delegate string MyDelegate(string str);

    class SimpleEventClass
```

```
{  
    public int Id {get; set;}  
  
    public event MyDelegate SimpleEvent;  
  
    public SimpleEventClass()  
    {  
        SimpleEvent += new MyDelegate(PrintText);  
    }  
    public string PrintText(string text)  
    {  
        return text;  
    }  
  
    static void Main(string[] args)  
    {  
        IDictionary<int, SimpleEventClass> col = new Dictionary<int,  
        SimpleEventClass>();  
        for (int currentObjectNo = 0; currentObjectNo < 500000;  
        currentObjectNo++)  
        {  
            col[currentObjectNo] = new SimpleEventClass  
            {Id = currentObjectNo};  
            string result = col[currentObjectNo].SimpleEvent("Raising  
            an event");  
            Console.WriteLine(currentObjectNo);  
        }  
        Console.ReadKey();  
    }  
}
```

Snapshots

Figure 29-8 shows the Diagnostic Tools window with seven different snapshots.

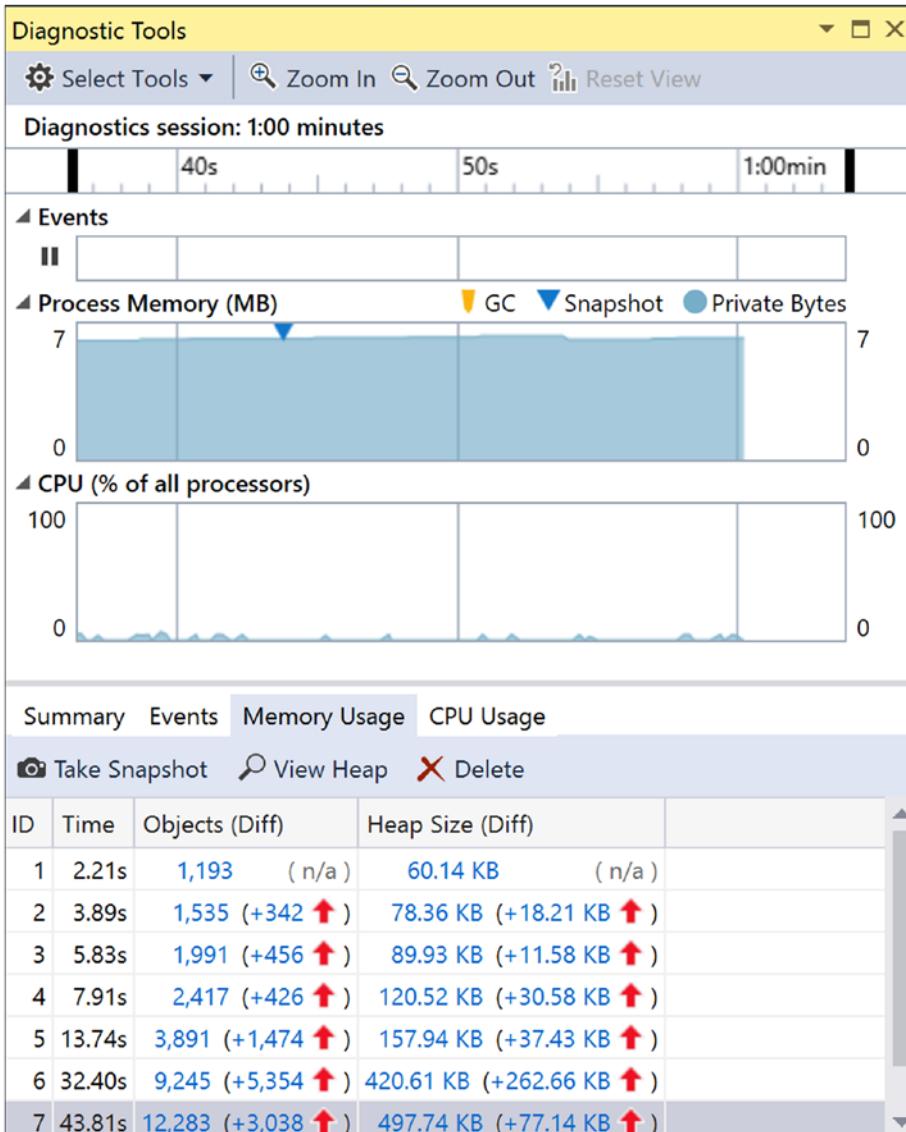


Figure 29-8. The Diagnostic Tools window with seven different snapshots to analyze memory usage at given points of time

Analysis

Notice how the heap size is growing over time? Open the heap view for any of these snapshots. Figure 29-9 shows snapshot 7.

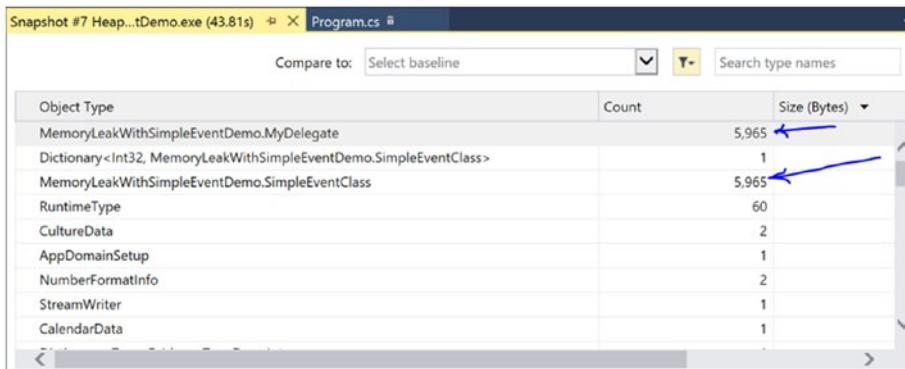


Figure 29-9. Heap view from snapshot 7

If you want to see the Objects (Diff) numbers, click the plus sign. You will notice that two things are showing up in Figure 29-10.

- `MemoryLeakWithSimpleEventDemo.SimpleEventClass`
- `MemoryLeakWithSimpleEventDemo.MyDelegate`

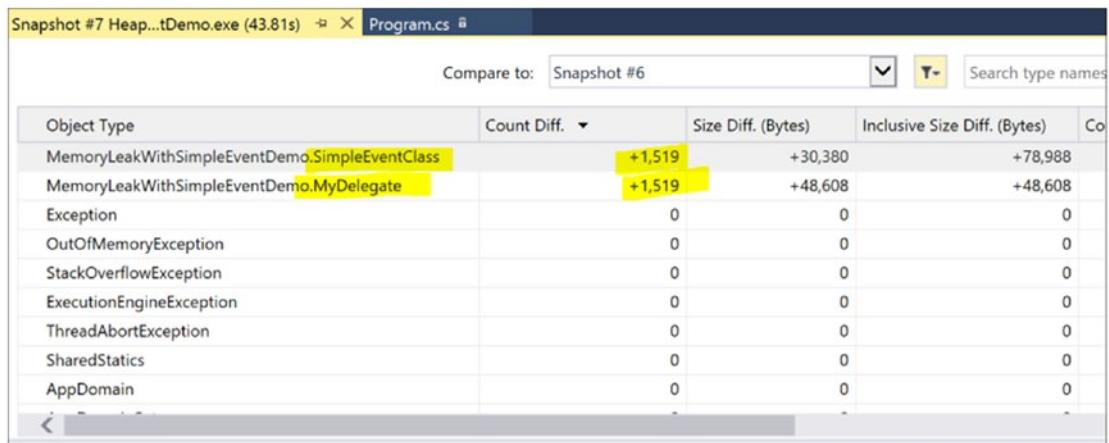


Figure 29-10. Memory leaks in snapshot 7

These results give you the following information:

- You registered an event in the code, but you never unregistered it.

```
SimpleEvent += new MyDelegate(PrintText);
```

- You are not making `SimpleEventClass` objects eligible for garbage collection.

So, following Microsoft's basic dispose pattern, let's modify the code.

Modified Code

Here's the modified code (the changes are shown in bold):

```
using System;
using System.Collections.Generic;

namespace MemoryLeakWithSimpleEventDemo
{
    public delegate string MyDelegate(string str);

    //class SimpleEventClass
    class SimpleEventClass:IDisposable
    {
        public int Id {get; set;}

        public event MyDelegate SimpleEvent;
        public bool disposed = false;

        public SimpleEventClass()
        {
            SimpleEvent += new MyDelegate(PrintText);
        }

        public string PrintText(string text)
        {
            return text;
        }

        public void Dispose()
    }
}
```

```
{  
    Dispose(true);  
    GC.SuppressFinalize(this);  
}  
  
protected virtual void Dispose(bool disposing)  
{  
    if (disposing)  
    {  
        //Use this section to cleanup managed objects  
        //In our case, we are just clearing the event subscription  
        if (this.SimpleEvent != null)  
        {  
            this.SimpleEvent -= new MyDelegate(PrintText);  
            Console.WriteLine("Unsubscribed");  
        }  
        //Use this section to cleanup unmanaged objects/resources  
        (if any)  
        //....  
        disposed = true;  
    }  
}  
  
~SimpleEventClass()  
{  
    Dispose(false);  
}  
  
static void Main(string[] args)  
{  
    IDictionary<int, SimpleEventClass> col = new Dictionary<int,  
    SimpleEventClass>();  
    for (int currentObjectNo = 0; currentObjectNo < 500000;  
    currentObjectNo++)  
    {  
        //using (col[currentObjectNo] = new SimpleEventClass  
        {Id = currentObjectNo})  
    }  
}
```

```
//{
    col[currentObjectNo] = new SimpleEventClass
    {Id = currentObjectNo};

    string result = col[currentObjectNo].
        SimpleEvent("Raising an event ");
    Console.WriteLine(currentObjectNo);
    col[currentObjectNo].Dispose();
//We are indicating that the object is now ready for GC
    col[currentObjectNo] = null;

}
Console.ReadKey();
}
}
```

Snapshots

Figure 29-11 shows the results after the modifications.

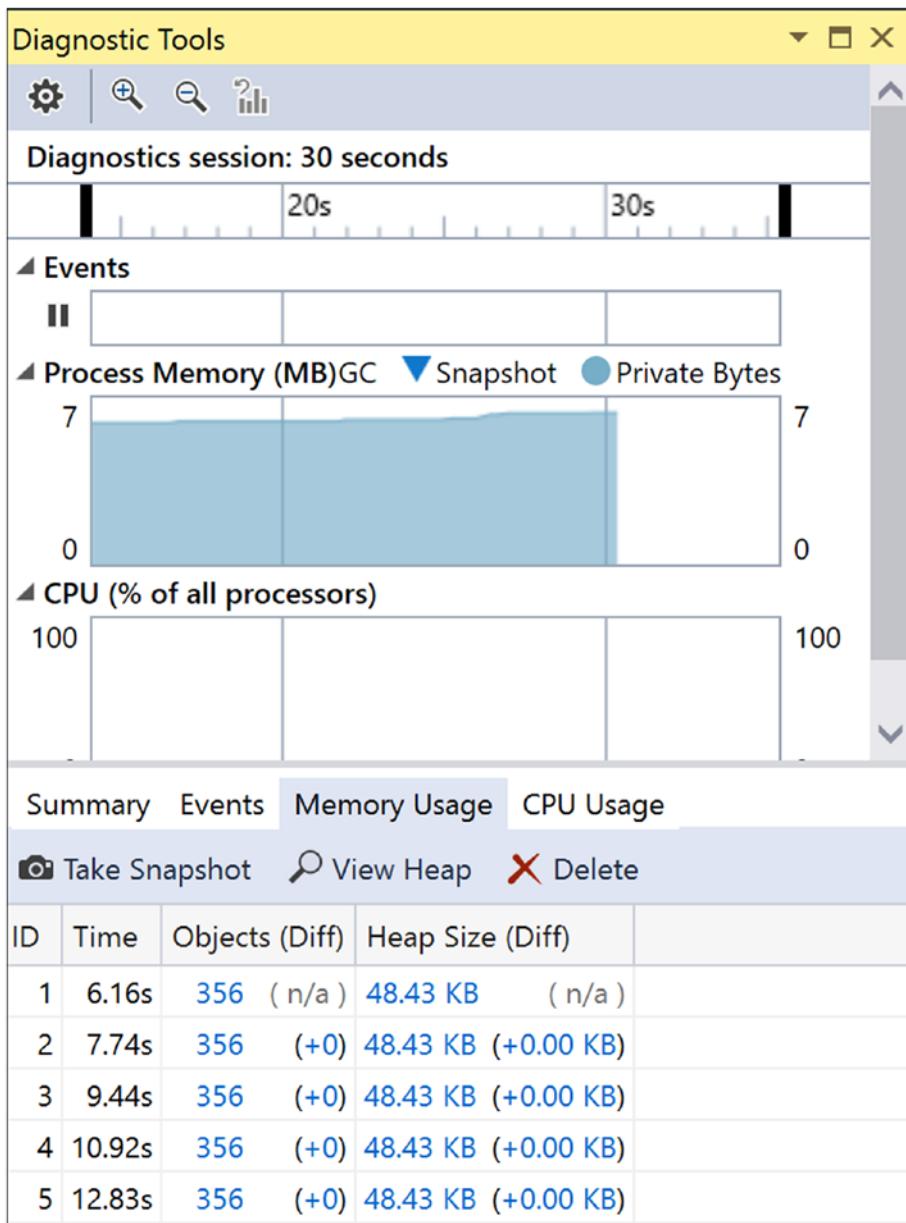


Figure 29-11. Modified results

Figure 29-12 shows the associated heap view. You will see clickable links for each of the components under Objects (Diff) and Heap Size (Diff) so you can analyze the growth (if any). I clicked the item 48.43 KB in the fifth row of Figure 29-11.

Object Type	Count	Size (Bytes)	Inclusive Size (Bytes)
Dictionary<Int32, MemoryLeakWithSimpleEventDemo.SimpleEventClass>	1	18,452	18,452
RuntimeType	60	1,680	1,680
CultureData	2	1,628	1,628
AppDomainSetup	1	848	848
NumberFormatInfo	2	692	692
StreamWriter	1	600	672
CalendarData	1	512	512
Dictionary<Type, EvidenceTypeDescriptor>	1	412	412
Dictionary<RuntimeType, RuntimeType>	1	292	304
CultureInfo	2	192	1,240

Figure 29-12. Modified heap view

Analysis

Here is what you can learn from this example:

- There is no incremental growth. In other words, everything is under control. This is a simple case study of detecting the leaks in an application and then sealing them.
- You can use the `using` statement (which is always better because even if the `Dispose()` call is bypassed, you will still be able to guard against the situation), as shown in Figure 29-13.

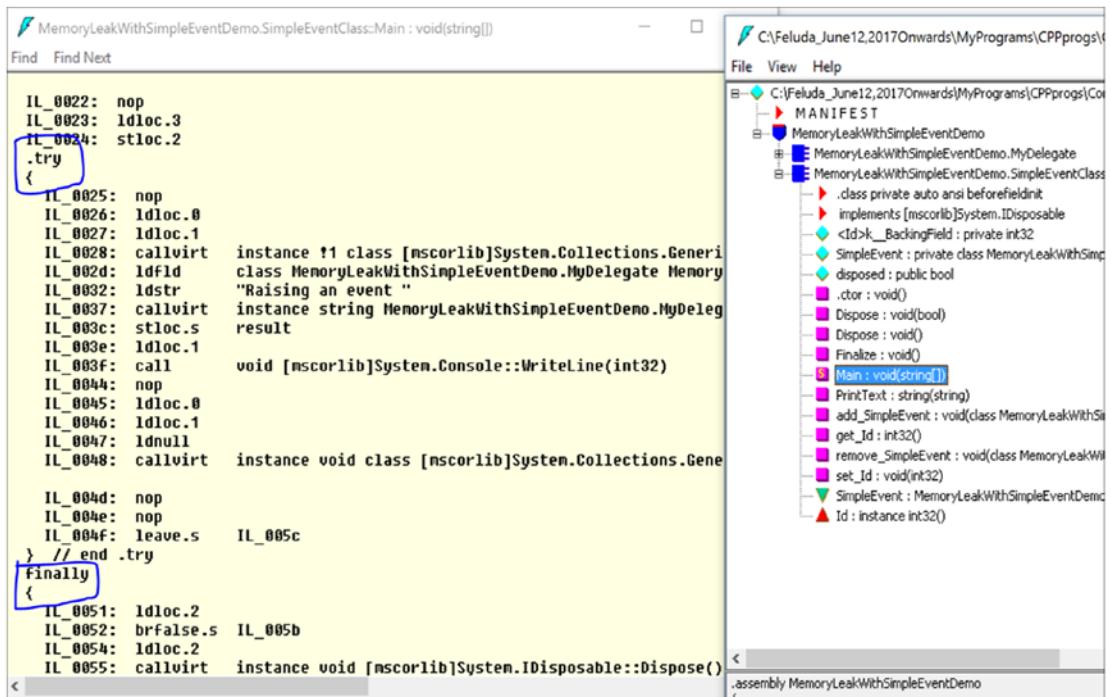
```

using (col[currentObjectNo] = new SimpleEventClass { Id = currentObjectNo })
{
    //col[currentObjectNo] = new SimpleEventClass { Id = currentObjectNo };
    string result = col[currentObjectNo].SimpleEvent("Raising an event ");
    Console.WriteLine(currentObjectNo);
    //col[currentObjectNo].Dispose();
    //We are indicating the the object is now ready for GC
    col[currentObjectNo] = null;
}

```

Figure 29-13. Use of using statement

- In addition, you must note another important characteristic. If you analyze the IL code of the Main() method, you will see a try/finally block, as shown in Figure 29-14.

**Figure 29-14.** IL code of the Main() method

In this case, the compiler has created this try/finally block for you since you are experimenting with the using statement. I already mentioned earlier that using statements act as syntactic shortcuts for try/finally blocks.

In my previous book, *Interactive C#*, I discussed a case study using Microsoft's CLR profiler. This tool is free and easy to use. You can download the CLR profiler (for .NET Framework 4) from <https://www.microsoft.com/en-in/download/confirmation.aspx?id=16273>.

If you are interested in viewing results from the CLR profiler, please refer to that book.

POINTS TO REMEMBER

As per Microsoft, the using statement ensures that Dispose is called even if an exception occurs while you are calling methods on the object. You can achieve the same result by putting the object inside a try block and then calling Dispose in a finally block; in fact, this is how the using statement is translated by the compiler. You can refer to the following documentation for more information: <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/using-statement>

Q&A Session 2

1. What are the common causes of memory leaks?

Answer:

Memory leaks can come from a variety of sources.

- You registered an event but forgot to unregister that event.
- Files/connections were opened but not closed properly.
- The Dispose() call was not implemented properly.
- The Dispose() call was bypassed somehow; for example, an exception was encountered in between.
- You encountered a deadlock (which may result in not releasing root objects).
- The finalizer thread is blocked; for example, the STA thread for the COM object is not available.

- There are leaks from unmanaged code.
- An object's lifetime is too high.
- There is a circular reference.
- Leaks can come from the .NET runtime environment (on rare occasions).
- Leaks may also appear from your testing framework. (In that case, it is a test leak, not a development environment leak, and test engineers are responsible for resolving it.)

2. What is a false leak?

Answer:

This means you do not need to worry about it. A common example is when you deal with a singleton class or a static class and there is some growth initially but you are certain that this kind of growth will not be continuous.

3. Many C# applications are integrated with C++ DLLs. So, leaks can come from that C++ code. Is this understanding correct?

Answer:

Yes.

4. Can you give an example of C++ leaks?

Answer:

If you allocate memory and do not free it after use, you will encounter leaks. So, in C++, the improper use of any of the functions such as `malloc()` or `calloc()` can result in leaks.

If you use following line of code:

```
int *ptr = (int *) malloc(sizeof(int));
```

Do not forget to use the following code to avoid leaks:

```
free(ptr);
```

5. Can you name some common symptoms of memory leaks?

- Devices hang frequently.
- The overall performance of the device/application keeps degrading.
- Repeated restarts are required.

6. It appears to me that if you restart the device, you can get back the memory. So, these leaks are not permanent. Is this understanding correct?

Answer:

Yes. With restart operations, you are clearing everything. But a restart operation is treated as a costly operation. For example, suppose you are sending a big file and midway your devices are restarted. Can you imagine the pain this would cause? So, in any application, if you need to perform undesirable restart operations often, you cannot treat this as a solution to the actual problem.

CHAPTER 30

FAQ

This chapter is a subset of the “Q&A Session” sections of all the chapters in this book. Many of these questions were not discussed in certain chapters because the related patterns were not covered yet. So, it is highly recommended that in addition to the following Q&As, you go through all the “Q&A Session” sections in the book for a better understanding of all the patterns.

1. Which design pattern do you like the most?

It depends on many factors such as the context, situation, demand, constraints, and so on. If you know about all of the patterns, you will have more options to choose from.

2. Why should developers use design patterns?

They are reusable solutions for software design problems that appear repeatedly in real-world software development.

3. What is the difference between the Command and Memento patterns?

All actions are stored for the Command pattern, but the Memento pattern saves the state only on request. Additionally, the Command pattern has undo and redo operations for every action, but the Memento pattern does not need that.

4. What is the difference between the Facade pattern and the Builder pattern?

The aim of the Facade pattern is to make a specific portion of code easier to use. It abstracts details away from the developer.

The Builder pattern separates the construction of an object from its representation. In Chapter 3, the director is calling the same `Construct()` method to create different types of vehicles. In other words, you can use the same construction process to create multiple types.

5. What is the difference between the Builder pattern and the Strategy pattern? They have similar UML representations.

First, you must take care of the intent first. The Builder pattern falls into the category of creational patterns, and the Strategy pattern falls into the category of behavioral patterns. Their areas of focus are different. With the Builder pattern, you can use the same construction process to create multiple types, and with the Strategy pattern, you have the freedom to select an algorithm at runtime.

6. What is the difference between the Command pattern and the Interpreter pattern?

In the Command pattern, commands are basically objects. In the Interpreter pattern, the commands are sentences. Sometimes the Interpreter pattern looks convenient, but you must keep in mind the cost of building an interpreter.

7. What is the difference between the Chain of Responsibility pattern and the Observer pattern?

For the Observer pattern, all registered users will be notified or get requests (for the change in subject) in parallel, but for the Chain of Responsibility pattern, you may not reach the end of the chain, so all users do not need to handle the same scenario. The request can be processed much earlier by some user who is placed at the beginning of the chain.

8. What is the difference between the Chain of Responsibility pattern and the Decorator pattern?

They are not the same at all, but you may feel that they are similar in their structures. Similar to the previous difference, at a given point of time, in the Chain of Responsibility pattern, only one class

handles the request, but in the case of the Decorator pattern, all classes handle the request. You must remember that decorators are effective in the context of adding and removing responsibilities only, and if you can combine the Decorator pattern with the single responsibility principle, you can add/remove a single responsibility at runtime.

9. What is the difference between the Mediator pattern and the Observer pattern?

The GoF says this: “These are competing patterns. The difference between them is that Observer distributes communication by introducing observer and subject objects, whereas a mediator object encapsulates the communication between other objects.”

Here I suggest you consider the example of the Mediator pattern in Chapter 21. In this example, two workers are always getting messages from their boss. It doesn’t matter whether they like those messages. But if they are simple observers, then they should have the option to unregister their boss’s control on themselves, effectively saying “I do not want to see messages from the boss.”

The GoF also found that you may face fewer challenges when making reusable observers and subjects than when making reusable mediators, but regarding the flow of communication, Mediator scores higher than Observer.

10. Which one do you prefer, a singleton class or a static class?

It depends. First, you can create objects of a singleton class, which is not possible with a static class. So, the concepts of inheritance and polymorphism can be implemented with a singleton class. In addition, some developers believe that mocking a static class (e.g., consider unit testing scenarios) in a real-world application is challenging.

11. How can you distinguish between proxies and adapters?

Answer:

Proxies work on similar interfaces as their subjects. Adapters work on different interfaces (to the objects they adapt).

12. How are proxies different from decorators?

Answer:

There are different types of proxies, and they vary as per their implementations. So, it may appear that some of these implementations are close to decorators. For example, a protection proxy might be implemented like a decorator. But you must remember that decorators focus on adding responsibilities, while proxies focus on controlling the access to an object.

13. How are mediators different from facades?

Answer:

In general, both simplify a complex system. In the case of a Mediator pattern, a two-way connection exists between a mediator and the internal subsystems, whereas in the case of a Facade pattern, in general, you provide a one-way connection (the subsystems do not care about facades).

14. Is there any relation between the Flyweight pattern and the State pattern?

Answer:

The GoF says that the Flyweight pattern can help you to decide when and how to share the state objects.

15. What are the similarities among the Simple Factory, Factory Method, and Abstract Factory design patterns?

Answer:

All of them encapsulate object creation. They suggest that you code to the abstraction (interface) but not to the concrete classes. In simple words, each of these factories promotes loose coupling by reducing the dependencies on concrete classes.

16. What are the differences among the Simple Factory, Factory Method, and Abstract Factory design patterns?

Answer:

This is an important question that you may face in various job interviews. So, refer to the discussion of question 3 in the “Q&A Session” section of Chapter 5.

17. How can you distinguish the Singleton pattern from the Factory Method pattern?

Answer:

The Singleton pattern ensures you get a unique instance each time. It also restricts the creation of additional instances.

But the Factory Method pattern does not say that you will get a unique instance only. Most often this pattern is used to create as many instances as you want, and the instances are not necessarily unique. These newly typed instances may implement a common base class. (Just remember that the Factory method lets a class defer instantiation to subclasses, per the GoF definition.)

APPENDIX A

Brief Overview of GoF Design Patterns

We all have unique thought processes. So, in the early days of software development, engineers faced a common problem—there was no standard to instruct them how to design their applications. Each team followed their own style, and when a new member (experienced or unexperienced) joined an existing team, understanding the architecture was a gigantic task. Senior or experienced members of the team would need to explain the advantages of the existing architecture and why alternative designs were not considered. The experienced developer also would know how to reduce future efforts by simply reusing the concepts already in place. *Design patterns* address this kind of issue and provide a common platform for all developers. You can think of them as the recorded experience of experts in the field. Patterns were intended to be applied in object-oriented designs with the intention of reuse.

In 1994, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides published the book *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley, 1994). In this book, they introduced the concept of design patterns in software development. These authors became known as the Gang of Four. We will refer them as the GoF throughout this book. The GoF described 23 patterns that were developed by the common experiences of software developers over a period of time. Nowadays, when a new member joins a development team, the developer first learns about the design patterns of the existing system. Then the developer learns about the existing architecture. This allows the developer to actively participate in the development process within a short period of time.

APPENDIX A BRIEF OVERVIEW OF GOF DESIGN PATTERNS

The first concept of a real-life design pattern came from the building architect Christopher Alexander. During his lifetime, he discovered that many of the problems he faced were similar in nature. So, he tried to address those issues with similar types of solutions.

Each pattern describes a problem, which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.

—Christopher Alexander

The GoF assured us that though these patterns were described for buildings and towns, the same concepts can be applied to patterns in object-oriented design. We can substitute the original concepts of walls and doors with objects and interfaces. The common thing in both fields is that, at their cores, patterns are solutions to common problems.

The GoF discussed the original concepts in the context of C++. C# 1.0 was released in 2002, and at the time of this writing, C# 7.0 is available with Visual Studio 2017. In this book, I'll examine the GoF's original design patterns but with C#. The book is written in C#, but if you are familiar with any other popular programming languages such as Java, C++, and so on, you will be able to relate because I focus on the design patterns and not on the latest features of C#. In fact, I have purposely chosen simple examples to help you to understand these concepts easily.

Key Points

The following are key points about design patterns:

- A design pattern is a general, reusable solution for commonly occurring problems. The basic idea is that you can solve similar kinds of problems with similar kinds of solutions. In addition, these solutions have been tested over a long period of time.
- Patterns generally provide a template of how to solve a problem and can be used in many different situations. At the same time, they help you to achieve the best possible design much faster.

- Patterns are descriptions of how to create objects and classes and then customize them to solve a general design problem in a particular context.
- The GoF discussed 23 design patterns. Each of these patterns focuses on a particular object-oriented design. Each pattern can also describe the consequences and trade-offs of use. The GoF categorized these 23 patterns based on their purposes, as shown here:

A. *Creational patterns*

These patterns abstract the instantiation process. You make the systems independent from how their objects are composed, created, and represented. The following five patterns are this category:

Singleton pattern
Prototype pattern
Factory Method pattern
Builder pattern
Abstract Factory pattern

B. *Structural patterns*

This category focuses on how classes and objects can be composed to form relatively large structures. They generally use inheritance to compose interfaces or implementations. The following seven patterns fall into this category:

Proxy pattern
Flyweight pattern
Composite pattern
Bridge pattern
Facade pattern
Decorator pattern
Adapter pattern

C. *Behavioral patterns*

Here the concentration is on algorithms and the assignment of responsibilities among objects. You also need to focus on the communication between them and how the objects are interconnected. The following 11 patterns fall into this category:

Observer pattern

Strategy pattern

Template Method pattern

Command pattern

Iterator pattern

Memento pattern

State pattern

Mediator pattern

Chain of Responsibility pattern

Visitor pattern

Interpreter pattern

The GoF made another classification based on scope, namely, whether the pattern primary focuses on the classes or its objects. Class patterns deal with classes and subclasses. They use inheritance mechanisms, which are static and fixed at compile time. Object patterns deal with objects that can change at run time. So, object patterns are dynamic.

For a quick reference, you can refer to the following table that was introduced by the GoF:

			Purpose	
		Creational	Structural	Behavioral
Scope	Class	1. Factory Method	1.1. Adapter(class)	1. Interpreter 2. Template Method
	Object	2. Singleton 3. Prototype 4. Builder 5. Abstract Factory	1.2. Adapter(object) 2. Proxy 3. Flyweight 4. Composite 5. Bridge 6. Facade 7. Decorator	3. Observer 4. Strategy 5. Command 6. Iterator 7. Memento 8. State 9. Mediator 10. Visitor 11. Chain of Responsibility

- In this book, you can start with any pattern you like. I have chosen simple examples so that you can pick up the basic ideas quickly. Read about each pattern, practice, try to solve other problems with these patterns, and then just keep coding. This process will help you to master the subject quickly.

Q&A Session

1. What are differences between class patterns and object patterns?

In general, class patterns focus on static relationship, and object patterns can focus on dynamic relationships. As the names suggest, class patterns focus on classes and their subclasses, and object patterns focus on the object relationships.

The GoF further differentiated them as follows:

	Class Patterns	Object Patterns
Creational	Can defer object creation to subclasses	Can defer object creation to another object
Structural	Focuses on the composition of classes (primarily uses the concept of inheritance)	Focuses on the different ways of composition of objects
Behavioral	Describes the algorithms and execution flows	Describes how different objects can work together and complete a task

2. Can two or more patterns be combined in an application?

Yes. In real-world scenarios, this type of activity is common.

3. Do these patterns depend on a particular programming language?

Programming languages can play an important role in pattern use. But the basic ideas are the same; patterns are just like templates, and they will give you some idea in advance of how you can solve a particular problem. In this book, I primarily focus on object-oriented programming with the concept of reuse. But instead of an object-oriented programming language, suppose you have chosen some other language like C. In that case, you may need to think about the core object-oriented principles such as inheritance, polymorphism, encapsulation, abstraction, and so on, and how to implement them. So, the choice of a particular language is always important because it may have some specialized features that can make your life easier.

4. Should we consider common data structures such as arrays and linked lists as design patterns?

The GoF clearly excludes these concepts, saying that they are not complex, domain-specific designs for an entire application or subsystem. They can be encoded in classes and reused as is. So, they are not your concern in this book.

APPENDIX B

Some Useful Resources

This appendix lists some useful resources. The following are helpful books:

- *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma et al. (Addison-Wesley, 1995)
- *Head First Design Patterns* by Eric Freeman and Elisabeth Robson (O'Reilly, 2004)
- *Java Design Patterns* by Vaskaran Sarcar (Apress, 2015)
- *Design Patterns For Dummies* by Steve Holzner (Wiley Publishing, Inc, 2006)
- *Design Patterns in C#* by Jean Paul (Kindle edition, 2012)

The following are helpful online resources/websites:

- www.dofactory.com
- www.c-sharpcorner.com
- www.dotnet-tricks.com/
- www.codeproject.com/
- http://sourcemaking.com/design_patterns
- <https://en.wikipedia.org>
- <https://www.youtube.com/watch?v=ffQZIGTTM48&list=PL8C53D99ABAD3F4C8>
- www.tutorialspoint.com/design_pattern
- www.dotnetexamples.com/
- [https://java.dzone.com/](http://java.dzone.com/)
- [https://msdn.microsoft.com/](http://msdn.microsoft.com/)

APPENDIX C

The Road Ahead

Congratulations! You have reached the end of our journey together. I believe you have enjoyed your learning experience, and this experience can help you to learn and experiment further in this category. I said earlier that if you repeatedly think about the discussions, examples, implementations, and questions and answers discussed in the book, you will have more clarity about them, you will feel more confident about them, and you will remake yourself in the programming world.

A detailed discussion of any particular design pattern would need many more pages, and the size of the book would be too gigantic to digest. So, what is next? You should not forget the basic principle that learning is a continuous process. This book was an attempt to encourage you to learn the core concepts in depth so that you can continue learning.

Still, learning by yourself will not be enough. I suggest you participate in open forums and discussions to get more clarity on the subjects in this book. This process will not only help you, it will help others also.

Lastly, I have a request: criticism is allowed, but please also let me know what you liked about this book. In general, an artistic view and open mind are required to discover the true efforts that are associated with any kind of work. Thank you and happy coding!

Index

A

Abstract Factory pattern

- Class Diagram, 58
- concept, 55
- GoF definition, 55
- illustration, 57
- implementation, 60–63
- Output, 63
- PetAnimalFactory, 57
- real-life example, 56
- Solution Explorer View, 58–59
- structure, 56, 57
- WildAnimalFactory, 57

Adapter pattern

- class, 110
- Class Diagram, 99
- concept, 97
- Directed Graph
 - Document, 100
 - drawbacks, 111
 - electrical outlet, 97–98
 - GetArea() method, 99
 - GoF definition, 97
 - implementation, 102–103
 - modified illustration
 - characteristics, 104
 - implementation, 106–108
 - Output, 109
 - Solution Explorer View, 105

object, 109

Output, 103

Solution Explorer

View, 100–101

user interface and
database, 98

Anti-patterns

- catalog, 394
- causes, 393
- concepts, 394
- definition, 391
- refactoring, 395
- software developers, 392
- symptoms, 393

B

Bridge pattern

- advantages, 164
- challenges, 164
- Class Diagram, 157–158
- concept, 155
- electronic items, 156–157
- GoF definition, 155
- GUI frameworks, 156
- implementation, 159–162
- Output, 163
- simple subclassing, 164
- Solution Explorer View, 159
- State pattern, 163

INDEX

Builder pattern

 Class Diagram, 33

 concept, 31

 ConcreteBuilder objects, 32

 converting RTF to ASCII, 32

 GoF definition, 31

 IBuilder interface, 32

 implementation, 34–38

 Output, 39

 real-life example, 32

 Solution Explorer View, 33–34

vs. Strategy pattern, 434

C

Chain of Responsibility pattern

 advantages, 313

 AuthenticationErrorHandler, 305

 challenges, 313

 Class Diagram, 306

 concept, 303

 customer care representatives, 304

vs. Decorator pattern, 434

 Directed Graph Document, 307

 EmailErrorHandler and

 FaxErrorHandler, 304–305

 GoF definition, 303

 implementation, 308, 310–311

 message priorities, 312

vs. Observer pattern, 314, 434

 Output, 312

 request-handling mechanism, 303

 software application, 304

 Solution Explorer View, 307–308

Class adapters, 110

Command pattern

 advantages, 242

 challenges, 242

Class Diagram, 225

concept, 223

Directed Graph Document, 226

GoF definition, 223

implementation, 228–230

vs. Interpreter pattern, 434

invocation process, 223

invoker, 241

vs. Memento pattern, 433

modified Class

 Diagram, 232

modified

 implementation, 234–239

Modified Output, 240

Output, 231

painting, 223

receivers, 231

 Solution Explorer
 View, 227, 233

WPF, 224

Common Language

 Runtime (CLR), 398

Composite pattern

 abstract class, 154

 advantages, 152

 challenges, 153

 Class Diagram, 146

 college organization, 144–145

 concept, 143

 data structure, 153

 GoF definition, 143

 implementation, 148–150

 iterator design pattern, 153

 object-oriented

 programming, 143

 organization and

 employees, 144

Output, 151

Solution Explorer View, 147
tree data structure, 144
Criticisms, 385–387

D, E

Decorator pattern
abstract method, 94
advantages, 91
Class Diagram, 86
component, 85
concept, 83
disadvantages, 94
dynamic behavior, 91
dynamic binding, 96
functionality, 83
GoF definition, 83
GUI-based toolkit, 85
implementation, 88–89
inheritance mechanism, 92–93
original house, 83–84
Output, 90
polymorphism, 95
single responsibility, 93
Solution Explorer View, 87

F

Facade pattern
vs. Adapter pattern, 122
advantages, 121
birthday party, 113
vs. Builder pattern, 433
challenges, 122
Class Diagram, 114
concept, 113
ConstructMilanoRobot and
 DestroyMilanoRobot, 114
Directed Graph Document, 115

GoF definition, 113
implementation, 116–119
loose coupling, 113
Output, 120–121
Solution Explorer View, 115
subsystems, 121–122
Factory Method pattern
analysis, 51
Class Diagram, 44
concept, 43
database users, 44
Directed Graph Document, 45
GoF definition, 43
IAnimalFactory class, 49
implementation, 46, 48
Output, 49, 51
Solution Explorer
 View, 45–46
Flyweight pattern
advantages, 140
business cards, 124
challenges, 141
Class Diagram, 125
computer game, 124
concept, 123
GoF definition, 123
implementation, 127–130
improvements, 131
modified Class Diagram, 132
modified implementation, 133–137
Modified Output, 137–138
multithreading, 140
nonshareable, 141
Output, 130
vs. Singleton pattern, 138, 140
small and large robots, 124
Solution Explorer View, 127, 133
vs. State pattern, 436

G, H

Garbage collection (GC)
 allocated objects, 400
 automatic memory management, 412
 characteristics, 404
 coding
 comment lines, 417
 compile, 413, 415
 destructor method, 417
 developer command
 prompt, 417–418
 invoking Dispose(), 418
 prediction, output, 415–416
 Common Language Runtime (CLR), 398
 compaction, 406
 destructors, 399, 405
 dirty objects, 397
 Dispose() method, 409–411
 Finalize() method, 405
 finalizers, drawbacks, 412
 GC.Collect(), 407
 GC.SuppressFinalize(this), 411
 IDisposable interface, 408–409
 implementation, 400–403
 longer-lived objects, 398
 low-priority thread, 397
 managed heap, 405
 memory consumption rate, 400
 Output, 403
 short-lived objects, 398
 System.GC.Collect() method, 400

I, J, K, L

Interpreter pattern
 advantages, 326
 challenges, 326
 Class Diagram, 316–317

concept, 315
 GoF definition, 315
 grammar rules, 316
 implementation, 318–325
 Java compiler, 315
 Output, 325
 Solution Explorer View, 318
 translator, foreign language, 315
 Iterator pattern
 challenges, 254
 Class Diagram, 245
 collection objects, 243, 254
 concept, 243
 data structure, student records, 244
 Directed Graph Document, 246
 First(), Next(), IsDone(), and
 CurrentItem() methods, 244
 GoF definition, 243
 implementation, 248–255
 Output, 253
 Solution Explorer View, 247

M

Mediator pattern
 advantages, 300–301
 airplanes, 283
 analysis, 292
 Class Diagram, 286
 concept, 283
 disadvantages, 301
vs. Facade pattern, 301, 436
 GoF definition, 283
 implementation, 288–292
 interconnections, objects, 301
 modified implementation, 294–299
 Modified Output, 299–300
vs. Observer pattern, 435

- Output, 292
 participants, 285
 registered participants, 293–294
 Solution Explorer View, 286–287
- Memento pattern
 advantages, 267
 ArrayList, 266
 challenges, 267
 Class Diagram, 258
 concept, 257
 Directed Graph Document, 259
 drawing application, 257
 GoF definition, 257
 implementation, 260–263
 mathematical model, 257
 memento, originator, and caretaker, 258
 modified implementation, 264–265
 Modified Output, 266
 operations, 268
 Output, 264
 Solution Explorer View, 260
- Memory leaks, 397
 causes, 430–431
 CLR profiler, 430
 demonstration code, 420–421
 Diagnostic Tools, 420, 422
 heap view, snapshots, 423, 428
 leaking rate, 397, 419
 IL code of Main() method, 429
 modified code, 424–425
 modified results, 427
 objects, 423
 online application, 419
 symptoms, 432
 System.OutOfMemoryException, 419
 tools, 420
 try/finally blocks, 429–430
 using statement, 428–430
- Model-view-controller (MVC) pattern
 advantages, 378
 architecture, variations, 356–358
 challenges, 378
 Class Diagram, 362
 components, 355–356
 definition, 355
 implementation, 363–372
 Output, 373–377, 380–381
 restaurant, 359
 Solution Explorer View, 360, 362–363
 web programming frameworks, 359
 Windows Forms application, 360–361
- ## N
- Null Object pattern
 analysis, unwanted input, 344
 challenges, 354
 Class Diagram, 347
 concept, 341
 definition, 341
 faulty program, 342–343
 if/else blocks, 346
 immediate remedy, 345
 implementation, 349–351
 Output, 352–353
 Output with valid inputs, 344
 runtime exception, 345
 Solution Explorer View, 348
 vehicles, 346
 washing machine, 346
- ## O
- Object adapters, 109
 Observer pattern
 benefits, 198
 Chain of Responsibility, 196

INDEX

- Observer pattern (*cont.*)
challenges, 198
Class Diagram, 189
concept, 185
GoF definition, 185
implementation, 191–194
notifications, 189, 196
notifications from subject, 185–188
one-to-many relationships, 197
Output, 194
Publisher-Subscriber model, 185
Solution Explorer View, 190
UI-based example, 188
- ## P, Q, R
- Prototype pattern
BasicCar object, 18
Class Diagram, 18–19
computer world example, 17
concept, 17
demonstration code, 27, 28
Directed Graph Document, 20
GoF definition, 17
implementation, 22–24
Output, 24, 29
real-life example, 17
Solution Explorer View, 20–21
- Proxy pattern
and adapters, 435
Class Diagram, 72
computer world example, 71
concept, 71
and decorators, 436
Directed Graph Document, 73
GoF definition, 71
implementation, 74–76
modified implementation, 79–81
- Modified Output, 82
Output, 76
real-life example, 71
Solution Explorer View, 73–74
- ## S
- Simple Factory pattern
challenges, 338
characteristics, 330
Class Diagram, 331
concept, 329
definition, 329
Directed Graph Document, 332
exception, 337
Factory Method and Abstract Factory patterns, 436–437
implementation, 333–336
ISimpleFactory, 338
Output, 336
Solution Explorer View, 332–333
static class, 339
Wikipedia, 329
- Singleton pattern
challenges, 10–11
characteristics, 6
Class Diagram, 6
computer world example, 5
concept, 5
vs. Factory Method pattern, 437
GoF definition, 5
implementation, 8–9
Output, 9–10
real-life example, 5
Solution Explorer View, 7
static initialization, 7
- State pattern
characteristics, 282

Class Diagram, 271
 concept, 269
 connection issues, 280
 contexts, method parameters, 282
 Directed Graph Document, 272
 GoF definition, 269
 implementation, 274–278
 job-processing system, 270
 Output, 279
 pros and cons, 282
 real-world scenario, 279
 Solution Explorer View, 272–273

S
 Strategy pattern
 abstract class, 210
 advantages, 210
 challenges, 210
 Class Diagram, 200
 concept, 199
 default behavior, 208, 210
 Directed Graph Document, 201
 GoF definition, 199
 implementation, 203–205
 inheritance mechanism, 206
 Output, 205
 Solution Explorer View, 202

Static initialization, 7

T, U

Template Method pattern
 advantages, 222
 BasicEngineering class, 217–218

challenges, 222
 Class Diagram, 213
 concept, 211
 GoF definition, 211
 implementation, 214, 216
 Modified Output, 221
 Modified Program, 218–220
 Output, 217
 Solution Explorer View, 214

V

Visitor pattern
 Class Diagram, 169
 Composite pattern
 implementation, 177–183
 Output, 183
 Solution Explorer View, 176
 tree structure, 173–176

concept, 167
 drawbacks, 184
 encapsulation, 184
 GoF definition, 167
 implementation, 170, 172
 Output, 172
 public APIs, 168
 Solution Explorer View, 170

W, X, Y, Z

Windows Presentation Foundation
 (WPF), 224