Exercise 1:Implementing the Singleton Pattern

using System;

using System.Collections.Generic;

using System.IO;

namespace SingletonPatternExample

{

/// <summary>

/// Logger class implementing the Singleton design pattern

/// Ensures only one instance exists throughout the application lifecycle

/// </summary>

public sealed class Logger

{

// Private static instance variable

private static Logger \_instance = null;

// Lock object for thread safety

private static readonly object \_lock = new object();

// List to store log messages

private readonly List<string> \_logMessages;

// Private constructor prevents direct instantiation

private Logger()

{

\_logMessages = new List<string>();

Console.WriteLine("Logger instance created at: " + DateTime.Now);

}

/// <summary>

/// Public static method to get the single instance of Logger

/// Thread-safe implementation using double-checked locking

/// </summary>

public static Logger Instance

{

get

{

// First check without locking for performance

if (\_instance == null)

{

// Lock to ensure thread safety

lock (\_lock)

{

// Double-check after acquiring lock

if (\_instance == null)

{

\_instance = new Logger();

}

}

}

return \_instance;

}

}

/// <summary>

/// Log an informational message

/// </summary>

/// <param name="message">The message to log</param>

public void LogInfo(string message)

{

string logEntry = $"[INFO] {DateTime.Now:yyyy-MM-dd HH:mm:ss} - {message}";

\_logMessages.Add(logEntry);

Console.WriteLine(logEntry);

}

/// <summary>

/// Log an error message

/// </summary>

/// <param name="message">The error message to log</param>

public void LogError(string message)

{

string logEntry = $"[ERROR] {DateTime.Now:yyyy-MM-dd HH:mm:ss} - {message}";

\_logMessages.Add(logEntry);

Console.WriteLine(logEntry);

}

/// <summary>

/// Log a warning message

/// </summary>

/// <param name="message">The warning message to log</param>

public void LogWarning(string message)

{

string logEntry = $"[WARNING] {DateTime.Now:yyyy-MM-dd HH:mm:ss} - {message}";

\_logMessages.Add(logEntry);

Console.WriteLine(logEntry);

}

/// <summary>

/// Get all logged messages

/// </summary>

/// <returns>List of all log messages</returns>

public List<string> GetAllLogs()

{

return new List<string>(\_logMessages);

}

/// <summary>

/// Clear all log messages

/// </summary>

public void ClearLogs()

{

\_logMessages.Clear();

Console.WriteLine("All logs cleared.");

}

/// <summary>

/// Get the current log count

/// </summary>

/// <returns>Number of log messages</returns>

public int LogCount => \_logMessages.Count;

}

/// <summary>

/// Test class to verify Singleton implementation

/// </summary>

public class SingletonTest

{

public static void RunTests()

{

Console.WriteLine("=== Singleton Pattern Test ===\n");

// Test 1: Verify single instance creation

Console.WriteLine("Test 1: Creating multiple Logger instances");

Logger logger1 = Logger.Instance;

Logger logger2 = Logger.Instance;

Logger logger3 = Logger.Instance;

// Check if all references point to the same instance

bool sameInstance = ReferenceEquals(logger1, logger2) &&

ReferenceEquals(logger2, logger3);

Console.WriteLine($"All instances are the same object: {sameInstance}");

Console.WriteLine($"logger1 hash code: {logger1.GetHashCode()}");

Console.WriteLine($"logger2 hash code: {logger2.GetHashCode()}");

Console.WriteLine($"logger3 hash code: {logger3.GetHashCode()}");

Console.WriteLine();

// Test 2: Test logging functionality

Console.WriteLine("Test 2: Testing logging functionality");

logger1.LogInfo("Application started");

logger2.LogWarning("This is a warning message");

logger3.LogError("This is an error message");

Console.WriteLine($"Total log count: {logger1.LogCount}");

Console.WriteLine();

// Test 3: Verify shared state

Console.WriteLine("Test 3: Verifying shared state across instances");

Console.WriteLine("All logs from logger1:");

foreach (string log in logger1.GetAllLogs())

{

Console.WriteLine($" {log}");

}

Console.WriteLine($"\nLog count from logger2: {logger2.LogCount}");

Console.WriteLine($"Log count from logger3: {logger3.LogCount}");

Console.WriteLine();

// Test 4: Thread safety test (basic)

Console.WriteLine("Test 4: Basic thread safety test");

System.Threading.Tasks.Parallel.For(0, 5, i =>

{

Logger threadLogger = Logger.Instance;

threadLogger.LogInfo($"Message from thread {i}");

});

Console.WriteLine($"Final log count after parallel execution: {logger1.LogCount}");

Console.WriteLine();

}

}

/// <summary>

/// Example usage class demonstrating real-world scenario

/// </summary>

public class ApplicationService

{

public void ProcessData()

{

Logger logger = Logger.Instance;

logger.LogInfo("Starting data processing");

try

{

// Simulate some processing

System.Threading.Thread.Sleep(100);

logger.LogInfo("Data processing completed successfully");

}

catch (Exception ex)

{

logger.LogError($"Error during data processing: {ex.Message}");

}

}

}

/// <summary>

/// Another service class to demonstrate singleton usage across different classes

/// </summary>

public class DatabaseService

{

public void ConnectToDatabase()

{

Logger logger = Logger.Instance;

logger.LogInfo("Attempting to connect to database");

// Simulate connection

bool connected = true; // Simulate successful connection

if (connected)

{

logger.LogInfo("Successfully connected to database");

}

else

{

logger.LogError("Failed to connect to database");

}

}

}

/// <summary>

/// Main program class

/// </summary>

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Singleton Pattern Example - Logger Implementation");

Console.WriteLine("================================================\n");

// Run singleton tests

SingletonTest.RunTests();

// Demonstrate real-world usage

Console.WriteLine("=== Real-world Usage Example ===");

ApplicationService appService = new ApplicationService();

DatabaseService dbService = new DatabaseService();

appService.ProcessData();

dbService.ConnectToDatabase();

// Show final log summary

Logger finalLogger = Logger.Instance;

Console.WriteLine($"\nFinal Summary:");

Console.WriteLine($"Total messages logged: {finalLogger.LogCount}");

Console.WriteLine("\nPress any key to exit...");

Console.ReadKey();

}

}

}

A screenshot of a computer

AI-generated content may be incorrect.

Exercise 2:Implement the Factory Method Pattern

using System;

using System.Collections.Generic;

namespace FactoryMethodPatternExample

{

/// <summary>

/// Abstract base class defining the common interface for all document types

/// </summary>

public abstract class Document

{

public string FileName { get; protected set; }

public DateTime CreatedDate { get; protected set; }

public string DocumentType { get; protected set; }

protected Document(string fileName)

{

FileName = fileName;

CreatedDate = DateTime.Now;

}

// Abstract methods that concrete document classes must implement

public abstract void Create();

public abstract void Open();

public abstract void Save();

public abstract void Close();

public abstract void Print();

// Common method for all documents

public virtual void DisplayInfo()

{

Console.WriteLine($"Document Info:");

Console.WriteLine($" Type: {DocumentType}");

Console.WriteLine($" File Name: {FileName}");

Console.WriteLine($" Created: {CreatedDate:yyyy-MM-dd HH:mm:ss}");

Console.WriteLine();

}

}

/// <summary>

/// Concrete implementation for Word documents

/// </summary>

public class WordDocument : Document

{

public int WordCount { get; private set; }

public int PageCount { get; private set; }

public WordDocument(string fileName) : base(fileName)

{

DocumentType = "Microsoft Word Document";

WordCount = 0;

PageCount = 1;

}

public override void Create()

{

Console.WriteLine($"Creating Word document: {FileName}");

Console.WriteLine("- Setting up Word document template");

Console.WriteLine("- Initializing formatting options");

Console.WriteLine("- Ready for text input");

}

public override void Open()

{

Console.WriteLine($"Opening Word document: {FileName}");

Console.WriteLine("- Loading document content");

Console.WriteLine("- Applying document styles");

}

public override void Save()

{

Console.WriteLine($"Saving Word document: {FileName}");

Console.WriteLine("- Saving in .docx format");

Console.WriteLine("- Preserving formatting and styles");

}

public override void Close()

{

Console.WriteLine($"Closing Word document: {FileName}");

Console.WriteLine("- Checking for unsaved changes");

Console.WriteLine("- Releasing Word resources");

}

public override void Print()

{

Console.WriteLine($"Printing Word document: {FileName}");

Console.WriteLine("- Preparing document for printing");

Console.WriteLine("- Sending to default printer");

}

public void AddText(string text)

{

WordCount += text.Split(' ').Length;

Console.WriteLine($"Added text to Word document. Word count: {WordCount}");

}

public void InsertTable(int rows, int columns)

{

Console.WriteLine($"Inserted {rows}x{columns} table into Word document");

}

}

/// <summary>

/// Concrete implementation for PDF documents

/// </summary>

public class PdfDocument : Document

{

public int PageCount { get; private set; }

public bool IsPasswordProtected { get; private set; }

public PdfDocument(string fileName) : base(fileName)

{

DocumentType = "Portable Document Format";

PageCount = 1;

IsPasswordProtected = false;

}

public override void Create()

{

Console.WriteLine($"Creating PDF document: {FileName}");

Console.WriteLine("- Initializing PDF structure");

Console.WriteLine("- Setting up page layout");

Console.WriteLine("- Configuring compression settings");

}

public override void Open()

{

Console.WriteLine($"Opening PDF document: {FileName}");

if (IsPasswordProtected)

{

Console.WriteLine("- Requesting password authentication");

}

Console.WriteLine("- Loading PDF content");

Console.WriteLine("- Rendering pages");

}

public override void Save()

{

Console.WriteLine($"Saving PDF document: {FileName}");

Console.WriteLine("- Optimizing file size");

Console.WriteLine("- Embedding fonts and images");

Console.WriteLine("- Finalizing PDF structure");

}

public override void Close()

{

Console.WriteLine($"Closing PDF document: {FileName}");

Console.WriteLine("- Releasing PDF resources");

Console.WriteLine("- Clearing memory cache");

}

public override void Print()

{

Console.WriteLine($"Printing PDF document: {FileName}");

Console.WriteLine("- Preparing high-quality output");

Console.WriteLine("- Maintaining document fidelity");

}

public void SetPassword(string password)

{

IsPasswordProtected = true;

Console.WriteLine("Password protection enabled for PDF document");

}

public void AddDigitalSignature(string signerName)

{

Console.WriteLine($"Digital signature added by: {signerName}");

}

}

/// <summary>

/// Concrete implementation for Excel documents

/// </summary>

public class ExcelDocument : Document

{

public int WorksheetCount { get; private set; }

public List<string> WorksheetNames { get; private set; }

public ExcelDocument(string fileName) : base(fileName)

{

DocumentType = "Microsoft Excel Spreadsheet";

WorksheetCount = 1;

WorksheetNames = new List<string> { "Sheet1" };

}

public override void Create()

{

Console.WriteLine($"Creating Excel document: {FileName}");

Console.WriteLine("- Setting up workbook structure");

Console.WriteLine("- Creating default worksheet");

Console.WriteLine("- Initializing grid layout");

}

public override void Open()

{

Console.WriteLine($"Opening Excel document: {FileName}");

Console.WriteLine("- Loading workbook data");

Console.WriteLine("- Parsing formulas and functions");

Console.WriteLine("- Rendering spreadsheet grid");

}

public override void Save()

{

Console.WriteLine($"Saving Excel document: {FileName}");

Console.WriteLine("- Saving in .xlsx format");

Console.WriteLine("- Preserving formulas and formatting");

Console.WriteLine("- Updating calculation chain");

}

public override void Close()

{

Console.WriteLine($"Closing Excel document: {FileName}");

Console.WriteLine("- Checking for unsaved changes");

Console.WriteLine("- Releasing Excel resources");

}

public override void Print()

{

Console.WriteLine($"Printing Excel document: {FileName}");

Console.WriteLine("- Preparing spreadsheet for printing");

Console.WriteLine("- Adjusting page breaks and scaling");

}

public void AddWorksheet(string worksheetName)

{

WorksheetNames.Add(worksheetName);

WorksheetCount++;

Console.WriteLine($"Added worksheet: {worksheetName}");

}

public void InsertChart(string chartType)

{

Console.WriteLine($"Inserted {chartType} chart into Excel document");

}

public void AddFormula(string cell, string formula)

{

Console.WriteLine($"Added formula '{formula}' to cell {cell}");

}

}

/// <summary>

/// Abstract factory class defining the factory method

/// </summary>

public abstract class DocumentFactory

{

// Factory method - to be implemented by concrete factories

public abstract Document CreateDocument(string fileName);

// Template method that uses the factory method

public Document ProcessDocument(string fileName)

{

Console.WriteLine("=== Document Processing Started ===");

// Use factory method to create document

Document document = CreateDocument(fileName);

// Common processing steps

document.Create();

document.DisplayInfo();

Console.WriteLine("=== Document Processing Completed ===\n");

return document;

}

}

/// <summary>

/// Concrete factory for creating Word documents

/// </summary>

public class WordDocumentFactory : DocumentFactory

{

public override Document CreateDocument(string fileName)

{

Console.WriteLine("WordDocumentFactory: Creating Word document...");

return new WordDocument(fileName);

}

}

/// <summary>

/// Concrete factory for creating PDF documents

/// </summary>

public class PdfDocumentFactory : DocumentFactory

{

public override Document CreateDocument(string fileName)

{

Console.WriteLine("PdfDocumentFactory: Creating PDF document...");

return new PdfDocument(fileName);

}

}

/// <summary>

/// Concrete factory for creating Excel documents

/// </summary>

public class ExcelDocumentFactory : DocumentFactory

{

public override Document CreateDocument(string fileName)

{

Console.WriteLine("ExcelDocumentFactory: Creating Excel document...");

return new ExcelDocument(fileName);

}

}

/// <summary>

/// Document manager that demonstrates the Factory Method pattern usage

/// </summary>

public class DocumentManager

{

private readonly Dictionary<string, DocumentFactory> \_factories;

public DocumentManager()

{

// Register available factories

\_factories = new Dictionary<string, DocumentFactory>

{

{ "word", new WordDocumentFactory() },

{ "pdf", new PdfDocumentFactory() },

{ "excel", new ExcelDocumentFactory() }

};

}

public Document CreateDocument(string documentType, string fileName)

{

string type = documentType.ToLower();

if (\_factories.ContainsKey(type))

{

return \_factories[type].CreateDocument(fileName);

}

else

{

throw new ArgumentException($"Unsupported document type: {documentType}");

}

}

public void ListSupportedTypes()

{

Console.WriteLine("Supported document types:");

foreach (string type in \_factories.Keys)

{

Console.WriteLine($" - {type}");

}

Console.WriteLine();

}

}

/// <summary>

/// Test class to demonstrate the Factory Method implementation

/// </summary>

public class FactoryMethodTest

{

public static void RunTests()

{

Console.WriteLine("=== Factory Method Pattern Test ===\n");

// Test 1: Direct factory usage

Console.WriteLine("Test 1: Direct Factory Usage");

Console.WriteLine("-----------------------------");

DocumentFactory wordFactory = new WordDocumentFactory();

DocumentFactory pdfFactory = new PdfDocumentFactory();

DocumentFactory excelFactory = new ExcelDocumentFactory();

Document wordDoc = wordFactory.ProcessDocument("Annual\_Report.docx");

Document pdfDoc = pdfFactory.ProcessDocument("User\_Manual.pdf");

Document excelDoc = excelFactory.ProcessDocument("Budget\_2024.xlsx");

// Test 2: Document-specific operations

Console.WriteLine("Test 2: Document-Specific Operations");

Console.WriteLine("-----------------------------------");

// Word document operations

if (wordDoc is WordDocument wd)

{

wd.AddText("This is a sample text for the annual report.");

wd.InsertTable(5, 3);

}

// PDF document operations

if (pdfDoc is PdfDocument pd)

{

pd.SetPassword("secure123");

pd.AddDigitalSignature("John Doe");

}

// Excel document operations

if (excelDoc is ExcelDocument ed)

{

ed.AddWorksheet("Data Analysis");

ed.AddWorksheet("Charts");

ed.InsertChart("Bar Chart");

ed.AddFormula("A1", "=SUM(B1:B10)");

}

Console.WriteLine();

// Test 3: Document Manager usage

Console.WriteLine("Test 3: Document Manager Usage");

Console.WriteLine("------------------------------");

DocumentManager manager = new DocumentManager();

manager.ListSupportedTypes();

try

{

Document doc1 = manager.CreateDocument("word", "Meeting\_Notes.docx");

Document doc2 = manager.CreateDocument("pdf", "Contract.pdf");

Document doc3 = manager.CreateDocument("excel", "Sales\_Data.xlsx");

// Demonstrate polymorphic behavior

Document[] documents = { doc1, doc2, doc3 };

Console.WriteLine("Performing common operations on all documents:");

foreach (Document doc in documents)

{

doc.Open();

doc.Save();

doc.Close();

Console.WriteLine();

}

}

catch (ArgumentException ex)

{

Console.WriteLine($"Error: {ex.Message}");

}

// Test 4: Error handling

Console.WriteLine("Test 4: Error Handling");

Console.WriteLine("---------------------");

try

{

Document invalidDoc = manager.CreateDocument("powerpoint", "Presentation.pptx");

}

catch (ArgumentException ex)

{

Console.WriteLine($"Expected error caught: {ex.Message}");

}

Console.WriteLine();

}

}

/// <summary>

/// Application service demonstrating real-world usage

/// </summary>

public class DocumentService

{

private readonly DocumentManager \_documentManager;

public DocumentService()

{

\_documentManager = new DocumentManager();

}

public void ProcessDocumentRequest(string documentType, string fileName, string operation)

{

try

{

Console.WriteLine($"Processing document request:");

Console.WriteLine($" Type: {documentType}");

Console.WriteLine($" File: {fileName}");

Console.WriteLine($" Operation: {operation}");

Console.WriteLine();

Document document = \_documentManager.CreateDocument(documentType, fileName);

switch (operation.ToLower())

{

case "create":

document.Create();

break;

case "open":

document.Open();

break;

case "print":

document.Print();

break;

default:

Console.WriteLine($"Unsupported operation: {operation}");

break;

}

Console.WriteLine("Request processed successfully!\n");

}

catch (Exception ex)

{

Console.WriteLine($"Error processing request: {ex.Message}\n");

}

}

}

/// <summary>

/// Main program class

/// </summary>

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Factory Method Pattern Example - Document Management System");

Console.WriteLine("==========================================================\n");

// Run comprehensive tests

FactoryMethodTest.RunTests();

// Demonstrate real-world usage

Console.WriteLine("=== Real-World Usage Demonstration ===");

Console.WriteLine("--------------------------------------");

DocumentService documentService = new DocumentService();

// Simulate different document processing requests

documentService.ProcessDocumentRequest("word", "ProjectProposal.docx", "create");

documentService.ProcessDocumentRequest("pdf", "TechnicalSpecs.pdf", "open");

documentService.ProcessDocumentRequest("excel", "FinancialReport.xlsx", "print");

documentService.ProcessDocumentRequest("powerpoint", "Presentation.pptx", "create"); // This will fail

Console.WriteLine("Press any key to exit...");

Console.ReadKey();

}

}

}

A computer screen shot of a black screen

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A computer screen shot of a black screen

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.