Manish Jaiswal

LLD

MASTERING

Composite Design Pattern

swipe

01

What is Composite?

Imagine treating a group of objects like a single object! That's the magic of the Composite Design Pattern. It lets you compose objects into tree structures, representing part-whole hierarchies. Clients can then interact with individual objects or entire groups uniformly.

swipe

02

Why Use Composite?

Unified Treatment: Apply operations to single objects or entire structures seamlessly.

Hierarchical Organization: Model complex systems with nested components, making code more readable.

Flexibility: Easily add new object types to the hierarchy without modifying existing code.

swipe

03

Where to Use Composite?

- When your application deals with tree-like data like file systems, organizational structures, or graphical elements.
- When you need to perform operations on both individual elements and composite groups consistently.

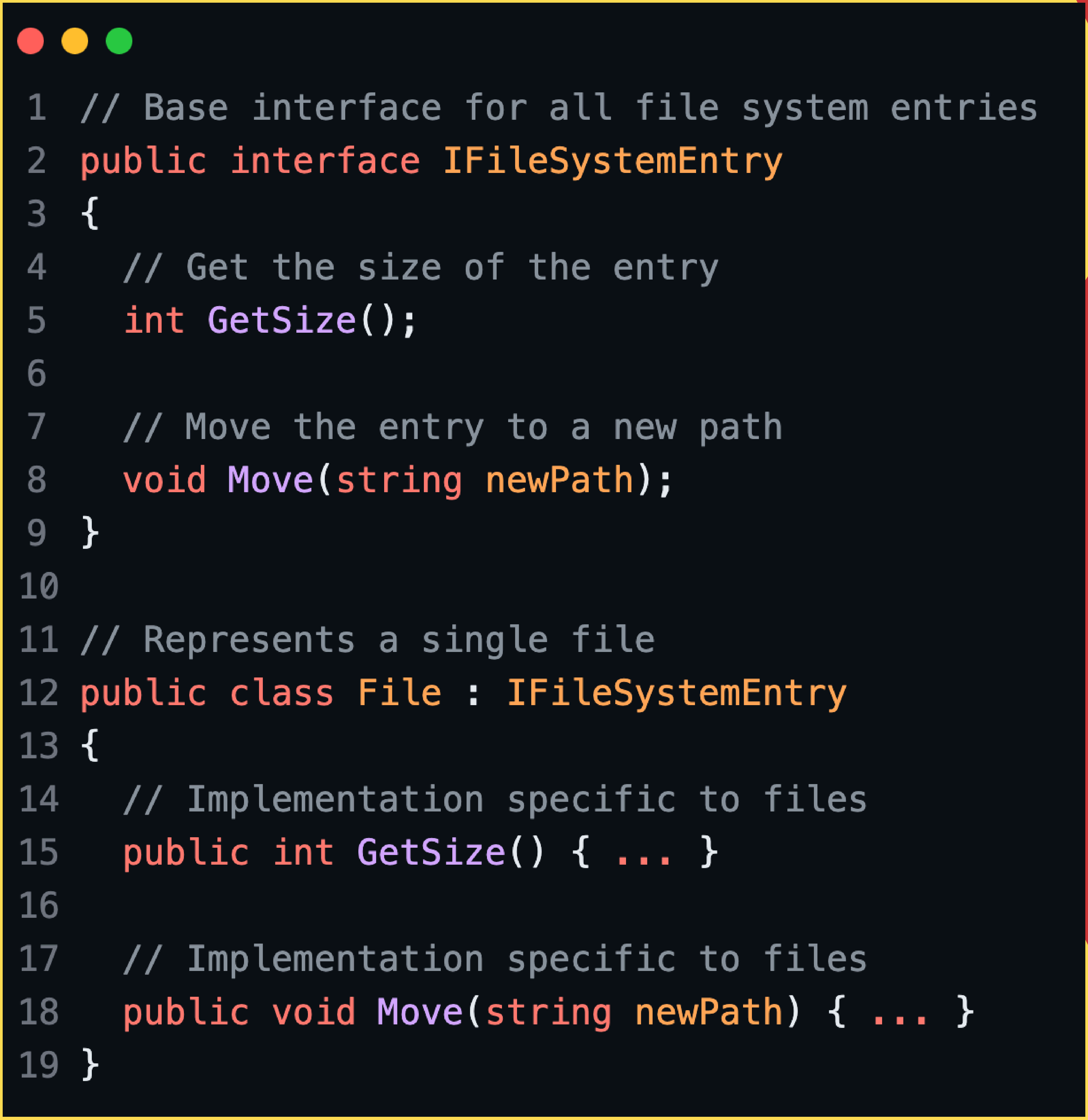
swipe

04

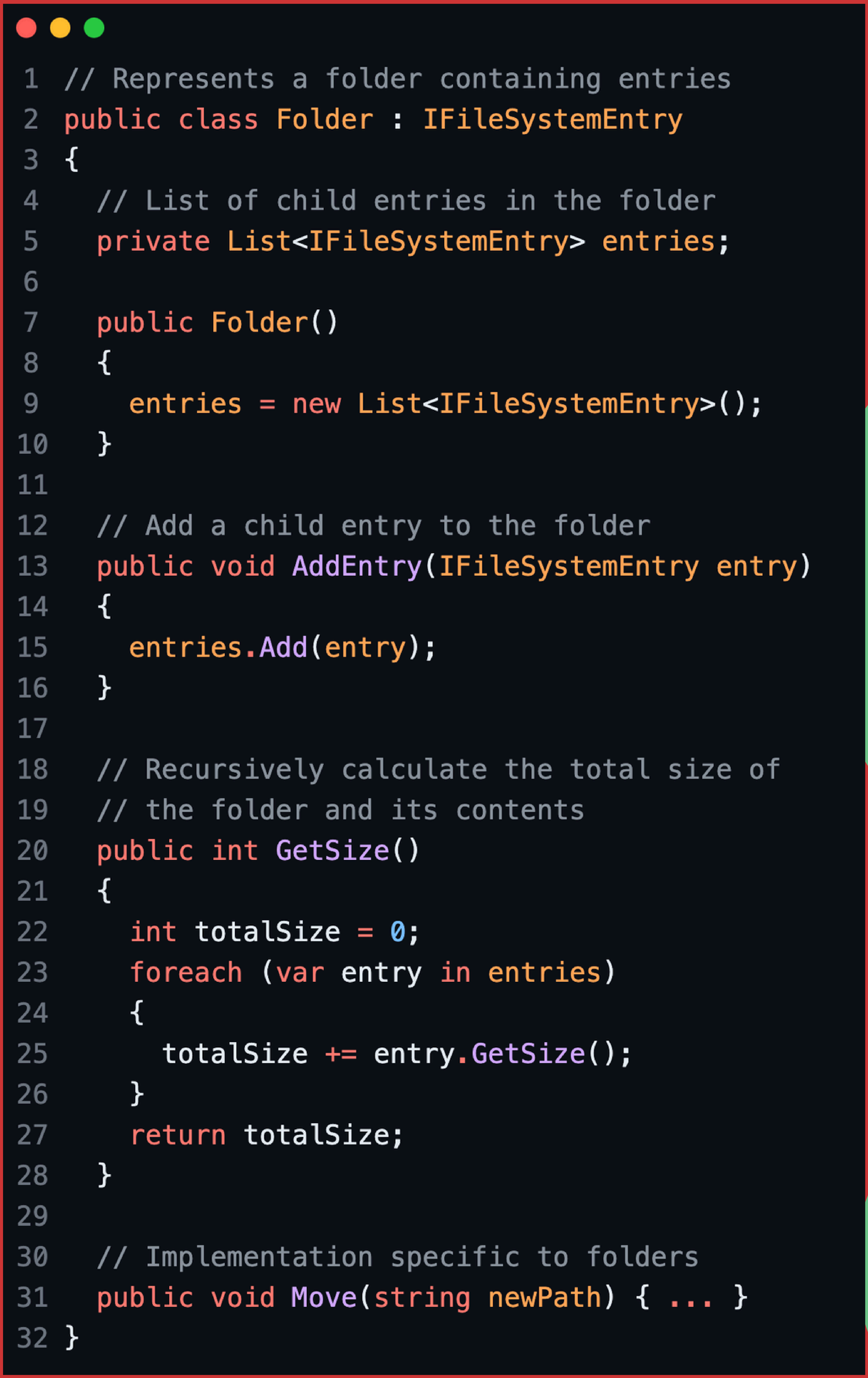
Real-World Example: File System

Think of your computer's file system. Folders act as composites, holding files (leaves) and other folders (sub-composites). Both folders and files inherit from a common base class (e.g., IFileSystemEntry) with operations like "GetSize()" or "Move()".

05 Code



```
// Base interface for all file system entries
public interface IFileSystemEntry
{
  // Get the size of the entry
  int GetSize();

  // Move the entry to a new path
  void Move(string newPath);
}

// Represents a single file
public class File : IFileSystemEntry
{
  // Implementation specific to files
  public int GetSize() { ... }

  // Implementation specific to files
  public void Move(string newPath) { ... }
}
```



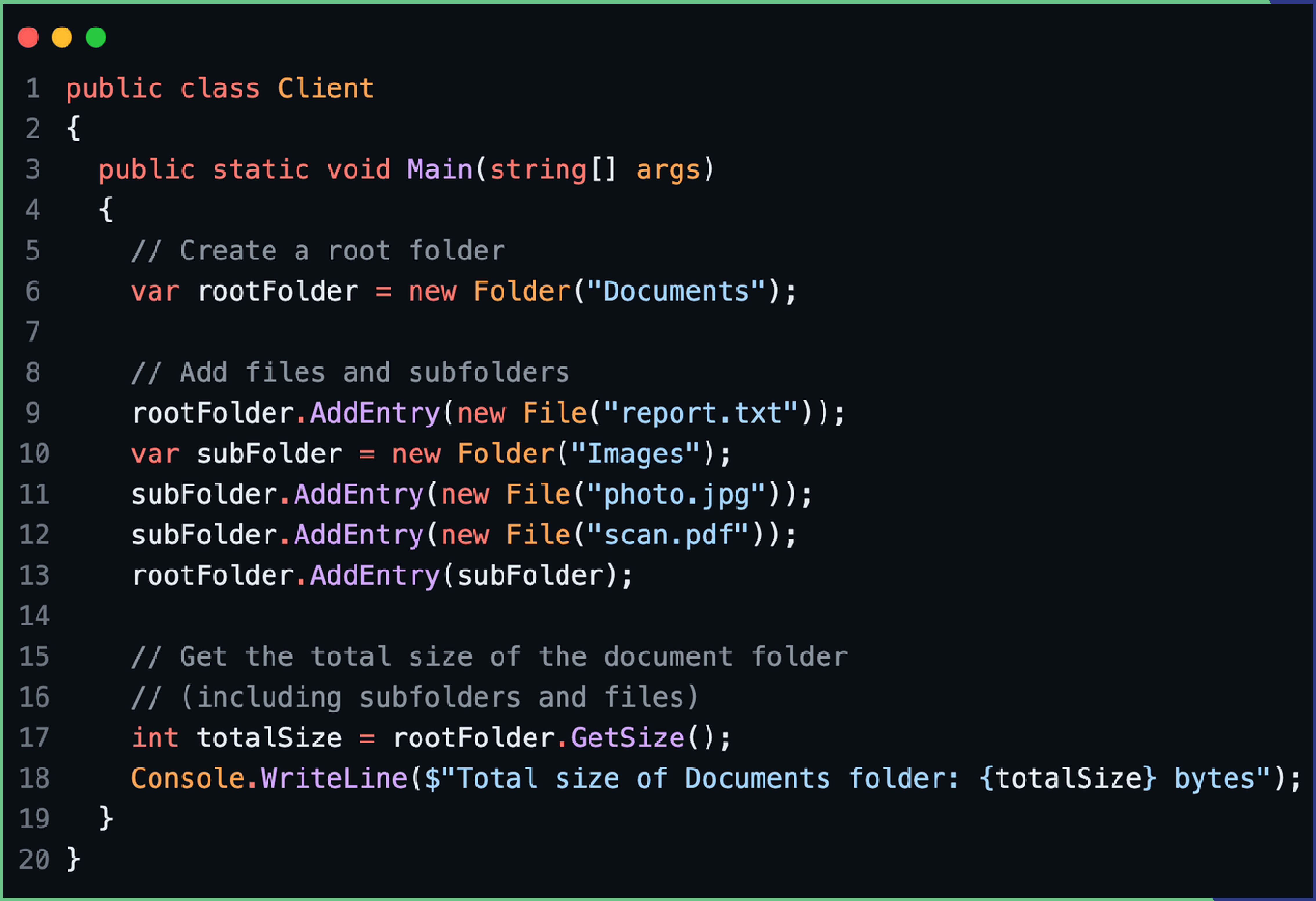
```
// Represents a folder containing entries
public class Folder : IFileSystemEntry
{
  // List of child entries in the folder
  private List<IFileSystemEntry> entries;

  public Folder()
  {
    entries = new List<IFileSystemEntry>();
  }

  // Add a child entry to the folder
  public void AddEntry(IFileSystemEntry entry)
  {
    entries.Add(entry);
  }

  // Recursively calculate the total size of
  // the folder and its contents
  public int GetSize()
  {
    int totalSize = 0;
    foreach (var entry in entries)
    {
      totalSize += entry.GetSize();
    }
    return totalSize;
  }

  // Implementation specific to folders
  public void Move(string newPath) { ... }
}
```

swipe

Manish Jaiswal

LLD

07 Code: Usage



```csharp
public class Client
{
  public static void Main(string[] args)
  {
    // Create a root folder
    var rootFolder = new Folder("Documents");

    // Add files and subfolders
    rootFolder.AddEntry(new File("report.txt"));
    var subFolder = new Folder("Images");
    subFolder.AddEntry(new File("photo.jpg"));
    subFolder.AddEntry(new File("scan.pdf"));
    rootFolder.AddEntry(subFolder);

    // Get the total size of the document folder
    // (including subfolders and files)
    int totalSize = rootFolder.GetSize();
    Console.WriteLine($"Total size of Documents folder: {totalSize} bytes");
  }
}
```

swipe

08

Conclusion

The Composite Design Pattern offers a powerful approach to structuring hierarchical data and operations. By leveraging this pattern, you can create cleaner, more maintainable, and flexible applications.