Bauprojekt

Erstellen Sie ein Programm, das die Verwaltung und Organisation eines Bauprojekts abbildet. Ein Bauprojekt besteht aus verschiedenen Komponenten wie Gebäuden, Gärten, Räumen und Materialien, die in einer hierarchischen Struktur angeordnet sind.

Ein Projekt kann mehrere Gebäude oder Gärten beinhalten. Ein Gebäude besteht aus Räumen und Räume werden mit Materialen gebaut. Erstelle eine einheitliche Schnittstelle zum Hinzufügen, aber sorge dafür dass ein Fehler bei falscher Verwendung ausgegeben wird.

Das Projekt soll eine zentrale Verwaltungsklasse besitzen, die Änderungen wie das Hinzufügen oder Entfernen von Bauelementen (Gebäude/Gärten/..) registriert und an zuständige Manager weiterleitet. Dazu sollen die Projektkomponenten wie Gebäude und Gärten eigenständig benachrichtigen, wenn sie selbst oder ihre untergeordneten Elemente verändert werden.

Um die Erstellung und Verwaltung der verschiedenen Komponenten einheitlich und nachvollziehbar zu gestalten, sollen für jedes Bauelement eigene Fabrikklassen verwendet werden. Dadurch wird sichergestellt, dass die richtigen Komponenten korrekt erstellt und zum Projekt hinzugefügt werden können. Beispielsweise wird ein Gebäude mithilfe einer speziellen Gebäudefabrik und ein Garten durch eine eigene Gartenfabrik erzeugt.

Zudem soll eine Protokollierungsfunktion in Form eines zentralen Loggers bereitgestellt werden, die alle Änderungen an den Komponenten des Bauprojekts aufzeichnet und diese in einer Datei speichert. Die Verwaltungsklasse für das Projekt ermöglicht es außerdem, mehrere Beobachter zu registrieren, die automatisch über alle Änderungen informiert werden.

Das Programm soll damit die Möglichkeit bieten, eine komplexe Projektstruktur einfach und effizient zu erstellen und zu verwalten, wobei alle Änderungen an einem zentralen Ort gemeldet und protokolliert werden.

Detailinfos:

Composite-Baumstruktur für ein Bauprojekt mit Observer-Pattern und Factory-Methode

Implementieren Sie ein Programm, das ein Bauprojekt mit einer hierarchischen Struktur modelliert. Verwenden Sie dabei das **Composite-Pattern**, um hierarchische Beziehungen zwischen den Bauelementen zu definieren, und das **Observer-Pattern**, um Änderungen im Projekt an die Beteiligten weiterzugeben. Um die Erzeugung der Komponenten zu flexibilisieren und die Kopplung zu reduzieren, nutzen Sie das **Factory-Pattern** für die Instanziierung der Bauelemente.

Anforderungen

1. **Projekt und Komponenten:**
    - Ein **Projekt** (Project) enthält mehrere **Projektkomponenten** (IProjectComponent).
    - Eine Projektkomponente kann entweder ein **Composite-Element** (z. B. Gebäude, Raum) oder ein **Leaf-Element** (z. B. Material, Garten) sein.
    - Ein **Gebäude** (Building) kann **Räume** (Room) enthalten.
    - Ein **Garten** (Garden) wird als eigenständiges Element behandelt und kann keine weiteren Elemente enthalten.
2. **Composite-Pattern:**
    - Verwenden Sie das Composite-Pattern, um die hierarchische Struktur der Projektkomponenten abzubilden.
    - Implementieren Sie eine abstrakte Basisklasse ACompositeElement für Komponenten, die weitere untergeordnete Komponenten enthalten können.
    - Implementieren Sie eine abstrakte Klasse ALeafElement für Elemente ohne untergeordnete Komponenten.
3. **Observer-Pattern:**
    - Das Projekt (Project) soll als **Subject** agieren, das Änderungen an seine **Observer** (z. B. ProjectManager oder ConstructionManager) weiterleitet.
    - Die Observer sollen benachrichtigt werden, wenn ein Element hinzugefügt oder entfernt wird.

4. **Factory-Pattern für die Erstellung der Elemente:**
    - Implementieren Sie ein Interface IProjectElementFactory<T>, um die Erstellung von Projektkomponenten zu kapseln.
    - Definieren Sie konkrete Factory-Klassen (z. B. BuildingFactory und GardenFactory) zur Erzeugung von spezifischen Komponenten.
    - Die Project-Klasse soll eine generische Methode CreateAndAdd<T> implementieren, welche über die entsprechende Factory-Instanz das gewünschte Element erstellt und zum Projekt hinzufügt.
5. **Schnittstellen und Constraints:**
    - Alle Projektkomponenten sollen einheitlich über das Interface IProjectElement behandelt werden. Dieses Interface definiert grundlegende Eigenschaften wie Name und Project.
    - Die CreateAndAdd<T>-Methode in der Project-Klasse soll sicherstellen, dass nur gültige Elemente zum Projekt hinzugefügt werden können.
6. **Log-Funktionalität:**
    - Verwenden Sie eine Singleton-Klasse MyLogger, um alle Operationen in Bezug auf das Hinzufügen und Entfernen von Komponenten zu protokollieren.
    - Der Logger soll am Ende alle protokollierten Nachrichten in eine Datei schreiben.

Hinweise

- Stellen Sie sicher, dass die hierarchischen Beziehungen korrekt eingehalten werden. Zum Beispiel darf ein Building keine Garden-Instanzen als untergeordnete Komponenten enthalten.
- Vermeiden Sie if-else-Konstrukte zur Überprüfung des Typs in der CreateAndAdd-Methode. Verwenden Sie stattdessen das Factory-Pattern, um die Kapselung der Objekterstellung zu gewährleisten.
- Implementieren Sie zwei Observer-Klassen (ProjectManager und ConstructionManager), die jeweils über Änderungen im Projekt benachrichtigt werden.

Unit-Tests

Unit-Tests für das Bauprojekt-Management-System, die verschiedene Aspekte der Implementierung abdecken:

1. **Test_Adding_Room_To_Building:** Überprüft, ob ein Raum korrekt zu einem Gebäude hinzugefügt wird.
2. **Test_Removing_Room_From_Building:** Überprüft, ob ein Raum korrekt aus einem Gebäude entfernt wird.
3. **Test_Adding_Material_To_Room:** Überprüft, ob Material korrekt zu einem Raum hinzugefügt wird.
4. **Test_Removing_Material_From_Room:** Überprüft, ob Material korrekt aus einem Raum entfernt wird.
5. **Test_Logger_Singleton_Behavior:** Überprüft, ob die Logger-Klasse als Singleton implementiert ist.
6. **Test_Logging_Addition:** Überprüft, ob der Logger einen Eintrag für das Hinzufügen eines Raums erstellt.
7. **Test_Logging_Removal:** Überprüft, ob der Logger einen Eintrag für das Entfernen eines Raums erstellt.
8. **Test_ProjectManager_Notification:** Überprüft, ob der Projektmanager benachrichtigt wird, wenn der Zustand des Projekts geändert wird.
9. **Test_ConstructionManager_Notification:** Überprüft, ob der Bauleiter benachrichtigt wird, wenn der Zustand des Projekts geändert wird.
10. **Test_Writing_Log_To_File:** Überprüft, ob der Logger einen Eintrag korrekt in eine Log-Datei schreibt.
11. **Test_Adding_Same_Room_To_Building_Does_Not_Duplicate:** Überprüft, dass ein Raum nicht mehrfach hinzugefügt wird.
12. **Test_Removing_Non_Existent_Room_From_Building:** Überprüft, dass das Entfernen eines nicht existierenden Raums keine Fehler verursacht.
13. **Test_Logging_Multiple_Entries:** Überprüft, dass der Logger mehrere Einträge für verschiedene Aktionen enthält.
14. **Test_Notification_Contains_Correct_Message:** Überprüft, dass die Benachrichtigung die korrekte Nachricht enthält.
15. **Test_Logger_Resets_Entries_After_Write:** Überprüft, dass die Logger-Einträge nach dem Schreiben in die Log-Datei zurückgesetzt werden.

Main:

Ergebnis – eine mögliche Implementierung die alle Komponenten testet:

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

    // Create project and observers

    Project project = new Project();
    ProjectManager projectManager = new ProjectManager();
    ConstructionManager constructionManager = new ConstructionManager();

    // Register observers
    project.RegisterObserver(projectManager);
    project.RegisterObserver(constructionManager);

    // Create project components
    //Building building = project.CreateAndAddBuilding("Main Building");
    //Garden garden = project.CreateAndAddGarden("Garden with Fountain");
    Building building = project.CreateAndAdd<Building>(
                        "Huge Building", new BuildingFactory());
    Garden garden = project.CreateAndAdd<Garden>(
                        "Flower Garden", new GardenFactory());
    Room room1 = new Room("Conference Room");
    Room room2 = new Room("Office Room");
    Material material1 = new Material("Steel");
    Material material2 = new Material("Concrete");

    // Add materials to rooms
    room1.Add(material1);
    room2.Add(material2);

    // Add rooms to building
    building.Add(room1);
    building.Add(room2);


    Console.WriteLine(projectManager.GetNotifications());
    Console.WriteLine(constructionManager.GetNotifications());

    // Write log to file
    MyLogger.Instance.WriteLogToFile("project_log.txt");
}
```

Implementierung Unit-Tests

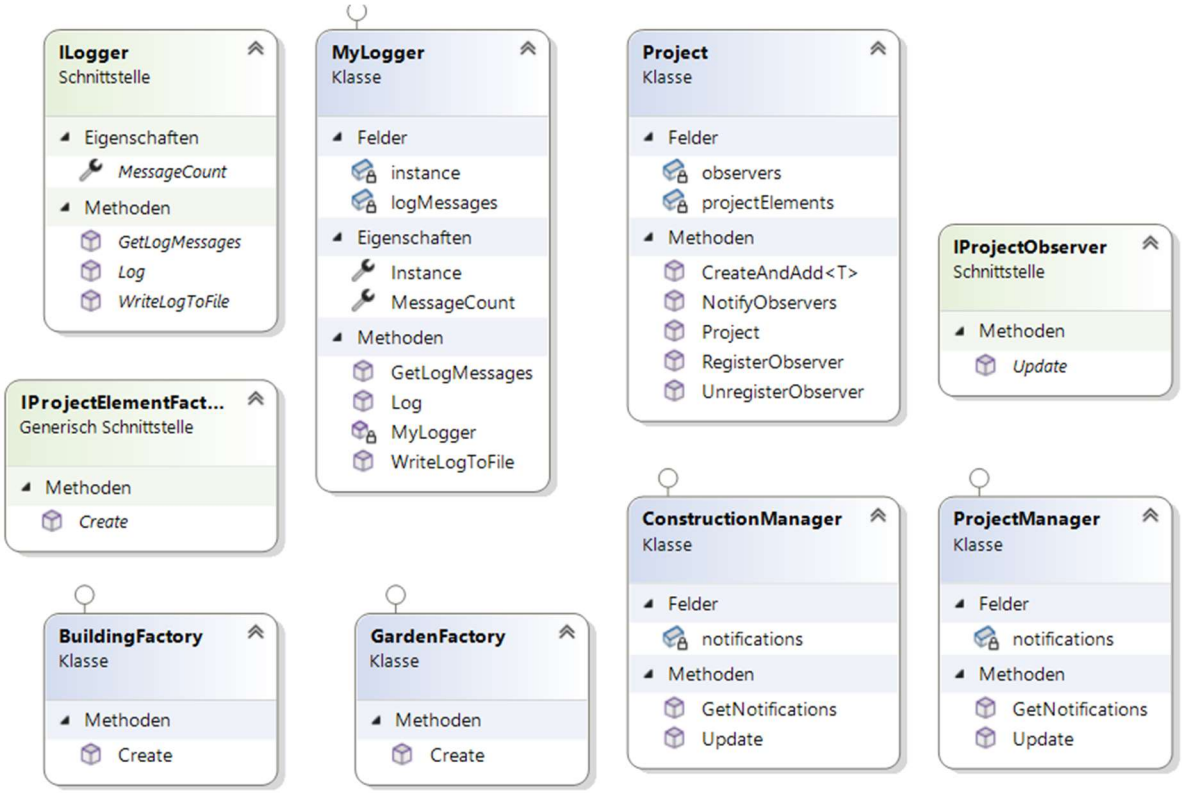
```
using NUnit.Framework;
using BuildingCompositeObserverPattern;
using System.IO;
using System.Linq;

namespace BuildingCompositeObserverPatternTests {
    [TestFixture]
    public class ProjectTests {
        private Project project;
        private ProjectManager projectManager;
        private ConstructionManager constructionManager;
        private Building building;
        private Room room;
        private Material material;

        [SetUp]
        public void Setup() {
            project = new Project();
            projectManager = new ProjectManager();
            constructionManager = new ConstructionManager();
            project.RegisterObserver(projectManager);
            project.RegisterObserver(constructionManager);
            building = new Building(project, "Main Building");
            room = new Room("Office Room");
            material = new Material("Concrete");
        }

        [Test]
        public void Test_Adding_Room_To_Building() {
            building.Add(room);
            Assert.Contains(room, building.Children);
        }

        [Test]
        public void Test_Removing_Room_From_Building() {
            building.Add(room);
            building.Remove(room);
            Assert.IsFalse(building.Children.Contains(room));
        }

        [Test]
        public void Test_Adding_Material_To_Room() {
            room.Add(material);
            Assert.Contains(material, room.Children);
        }

        [Test]
        public void Test_Removing_Material_From_Room() {
            room.Add(material);
            room.Remove(material);
            Assert.IsFalse(room.Children.Contains(material));
        }

        [Test]
```

```
        public void Test_Logger_Singleton_Behavior() {
            var logger1 = MyLogger.Instance;
            var logger2 = MyLogger.Instance;
            Assert.AreSame(logger1, logger2, "Logger should be a singleton
instance.");
        }


        [Test]
        public void Test_Logging_Addition() {
            // Vor dem Test sicherstellen, dass der Logger keine alten Nachrichten
enthält
            MyLogger.Instance.WriteLogToFile("temp.txt"); // Schreibt alle
aktuellen Logs in eine temporäre Datei und leert die Liste
            File.Delete("temp.txt"); // Löscht die temporäre Datei

            // Eine neue Building-Instanz erstellen
            Building newBuilding = new Building(project, "New Test Building");

            // Einen Raum zur neuen Building-Instanz hinzufügen
            Room newRoom = new Room("Test Room");
            newBuilding.Add(newRoom);

            // Überprüfen, ob der Logger die korrekte Nachricht hinzugefügt hat
            string expectedLogEntry = $"Added {newRoom.GetDetails()} to
{newBuilding.Name}.";
            Assert.IsTrue(MyLogger.Instance.MessageCount > 0, "Logger should
contain at least one message.");

        }



        [Test]
        public void Test_Logging_Removal() {
            // Vorhandene Anzahl der Protokolleinträge speichern
            int initialCount = MyLogger.Instance.MessageCount;

            // Raum erstellen und zum Gebäude hinzufügen
            Room room = new Room("Classroom");
            building.Add(room);

            // Überprüfen, ob ein neuer Protokolleintrag für das Hinzufügen
erstellt wurde
            Assert.AreEqual(initialCount + 1, MyLogger.Instance.MessageCount,
"Logger should have one more entry after adding a room.");

            // Raum aus dem Gebäude entfernen
            building.Remove(room);

            // Nach dem Entfernen sollte die Anzahl der Protokolleinträge gleich
der Anzahl nach dem Hinzufügen sein
            Assert.AreEqual(initialCount + 2, MyLogger.Instance.MessageCount,
"Logger should still have one entry after removing a room.");
        }


        [Test]
        public void Test_ProjectManager_Notification() {
            project.NotifyObservers("Project Updated");
```

```
            StringAssert.Contains("Project Manager notified: Project Updated",
projectManager.GetNotifications());
        }

        [Test]
        public void Test_ConstructionManager_Notification() {
            project.NotifyObservers("Building Added");
            StringAssert.Contains("Construction Manager notified: Building Added",
constructionManager.GetNotifications());
        }

        [Test]
        public void Test_Writing_Log_To_File() {
            string filePath = "test_log.txt";
            MyLogger.Instance.Log("Test log entry");
            MyLogger.Instance.WriteLogToFile(filePath);
            Assert.IsTrue(File.Exists(filePath));
            var logContent = File.ReadAllText(filePath);
            StringAssert.Contains("Test log entry", logContent);
            File.Delete(filePath);
        }

        [Test]
        public void Test_Adding_Same_Room_To_Building_Does_Not_Duplicate() {
            building.Add(room);
            building.Add(room);
            Assert.AreEqual(1, building.Children.Count(child => child == room));
        }

        [Test]
        public void Test_Removing_Non_Existent_Room_From_Building() {
            Assert.DoesNotThrow(() => building.Remove(room));
        }

        [Test]
        public void Test_Logging_Multiple_Entries() {
            // Logger vor dem Test zurücksetzen
            MyLogger.Instance.WriteLogToFile("temp_log.txt"); // Bestehende Logs
schreiben und löschen
            File.Delete("temp_log.txt"); // Temporäre Datei löschen

            // Ausgangszustand sicherstellen
            Assert.AreEqual(0, MyLogger.Instance.MessageCount, "Logger sollte vor
dem Testfall keine Einträge enthalten.");

            // Neue Instanzen erstellen, um mögliche zusätzliche Log-Nachrichten
zu vermeiden
            Project project = new Project();
            Building building = new Building(project, "Test Building");
            Room room = new Room("Test Room");

            // Raum hinzufügen und entfernen
            building.Add(room);     // Erwartet: 1 Log-Eintrag
            building.Remove(room);  // Erwartet: 1 Log-Eintrag

            // Debugging: Log-Nachrichten anzeigen
            Console.WriteLine("Aktuelle Log-Einträge:");
            foreach (var message in MyLogger.Instance.MessageCount.ToString()) {
                Console.WriteLine(message);
            }
```

```
            // Anzahl der Log-Einträge überprüfen
            Assert.AreEqual(2, MyLogger.Instance.MessageCount, $"Logger sollte
genau 2 Einträge haben, hat aber {MyLogger.Instance.MessageCount} Einträge.");
        }


        [Test]
        public void Test_Notification_Contains_Correct_Message() {
            project.NotifyObservers("Room Added");
            StringAssert.Contains("Room Added",
projectManager.GetNotifications());
        }

        [Test]
        public void Test_Logger_Resets_Entries_After_Write() {
            string filePath = "test_log_reset.txt";
            MyLogger.Instance.Log("Test reset log");
            MyLogger.Instance.WriteLogToFile(filePath);
            Assert.AreEqual(0, MyLogger.Instance.MessageCount, "Logger should
reset after writing to file.");
            File.Delete(filePath);
        }
    }
}
```

Klassendiagramm:



Implementierung:

```
namespace BuildingCompositeObserverPattern {
    using System;
    using System.Collections.Generic;
    using System.IO;

    public interface ILogger {
        void Log(string message); // Methode zum Protokollieren von Nachrichten
        void WriteLogToFile(string filePath); // Methode zum Schreiben von
Protokolleinträgen in eine Datei
        int MessageCount { get; } // Eigenschaft, die die Anzahl der
Protokolleinträge zurückgibt
        List<string> GetLogMessages(); // Methode zum Abrufen der
Protokolleinträge
    }


    // Logger class (Singleton)
    public class MyLogger:ILogger {
        private static MyLogger instance;
        private List<string> logMessages;
        public int MessageCount { get => logMessages.Count; }

        private MyLogger() {
            logMessages = new List<string>();
        }
```