**Лабораторная работа №3**

**«Реализация одного из структурных паттернов проектирования»**

**Цель работы:** познакомиться со структурными паттернами проектирования и применить паттерн Компоновщик (Composite).

**Продолжительность работы:** 4 часа.

**Структурные паттерны**

В структурных паттернах рассматривается вопрос о том, как из классов и объектов образуются более крупные структуры.

Структурные паттерны уровня *класса* используют наследование для составления композиций из интерфейсов (interface) и реализаций (class). Например, в языке C++ разрешено множественное наследование реализаций для объединения нескольких классов в один. В результате получается класс, обладающий свойствами всех своих родителей. В C# недопустимо множественное наследование реализации, но допустимо множественное наследование интерфейсов, что позволяет организовывать композиции интерфейсов.

В структурных паттернах уровня *объекта* используют композицию объектов для получения новой функциональности. Дополнительная гибкость в этом случае связана с возможностью изменить композицию объектов во время выполнения, что недопустимо для статической композиции классов.

Среди паттернов проектирования (GoF) выделяют 8 структурных. Из них всего один является паттерном уровня класса – *Адаптер (Adapter)*. Он представляет собой программную обертку над уже существующими классами и предназначен для преобразования их интерфейсов к виду, пригодному для последующего использования в новом программном проекте.

Все остальные являются паттернами уровня объектов:

* Мост (Bridge) – разделяет абстракцию и реализацию, позволяя менять их независимо друг от друга;
* Компоновщик (Composite) – группирует схожие объекты в древовидную структуру, позволяя работать с ней так, будто это единичный объект;
* Декоратор (Decorator) – позволяет динамически добавлять объекту новую функциональность;
* Фасад (Facade) – предоставляет простой интерфейс к сложной системе классов (например, библиотеке), что облегчает ее использование;
* Легковес (Flyweight) – использует разделение для эффективной поддержки множества объектов;
* Заместитель (Proxy) – позволяет подставлять вместо реальных объектов специальные объекты-заменители. Это объекты перехватывают обращения к оригинальному объекту и позволяют сделать что-то *до* или *после* передачи обращения оригиналу.

Теперь давайте подробнее познакомимся с паттерном Компоновщик (Composite).

**Паттерн Компоновщик (Composite)**

Компоновщик – структурный паттерн уровня объектов, который позволяет сгруппировать множество объектов в древовидную структуру, а затем, работать с ней так, как будто это единичный объект.

Данный паттерн имеет смысл только тогда, когда основная модель вашей программы может быть представлены в виде дерева.

**Пример из жизни**

Например, у нас есть два объекта: *Папка* и *Файл.* В *Папке* могут находиться как *Файлы*, так и другие *Папки,* в которых в свою очередь могут находиться *Файлы* или *Папки* и так далее.

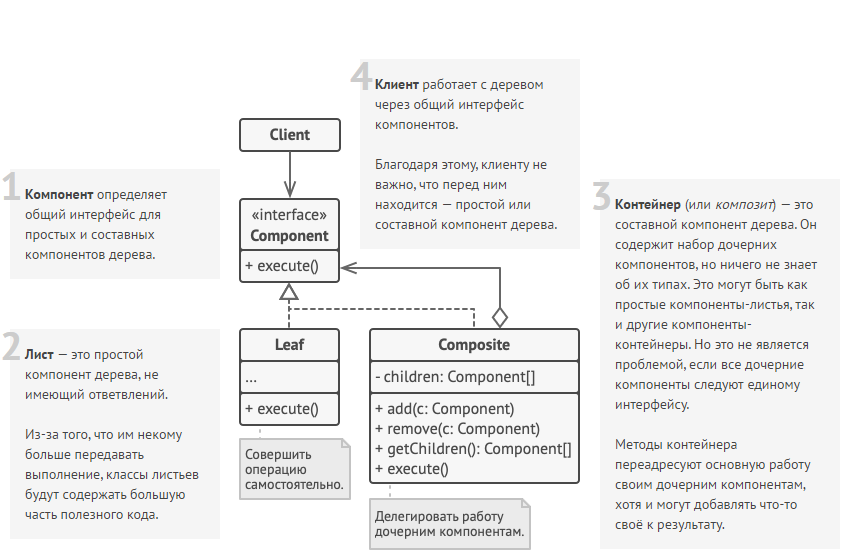
Допустим, перед нами стоит задача посчитать суммарный вес файлов в папке. Заранее мы не знаем количество уровней вложенности папок, поэтому перебрать всё простым циклом не получится.

Паттерн Компоновщик позволяет использовать рекурсивную композицию таким образом, что клиент работает с примитивными (*файлами*) и составными объектами (*папками*) единым образом.

Мы будем использовать единый интерфейс с общим методом получения суммарного веса. *Файл* просто вернет свой вес, а *Папка* посчитает вес каждого вложенного объекта внутри себя и вернет суммарный результат. Если одним из внутренних объектов окажется *Папка*, то она тоже начнет перебирать всё свое содержимое и так далее, пока не будут подсчитаны все составные части.

Для клиента главное, что теперь ничего не нужно знать о структуре *Папки*. Он просто вызывает метод подсчета суммарного веса и получает цифру, в независимости от того, как много *Файлов* и *Папок* внутри.

**Структура**



**Пример реализации**

Давайте применим паттерн Компоновщик (Composite) для примера из жизни, описанного выше – напишем код, который позволяет нам подсчитать общий вес всех файлов в папке.

Но для начала обсудим важный момент. При проектировании с использованием *Компоновщика* часто встает вопрос куда поместить методы добавления/удаления дочерних компонентов. С одной стороны их можно разместить в интерфейсе Component. Благодаря этому все компоненты дерева станут одинаковыми для клиента. Однако это нарушает один из принципов ООП – *принцип разделения интерфейса.* Вызов этих методов для примитивных объектов считается ошибочным и небезопасным. Другой подход изображен на UML диаграмме выше – методы добавления/удаления реализованы только для составных объектов. Выбирать между удобством и безопасность необходимо в контексте задачи.

Реализация паттерна Компоновщик (Composite)

using System**;**using System**.**Collections**.**Generic**;**namespace CompositePattern  
**{** public interface IComponent  
 **{** *// Метод для получения веса* public float GetSize**();** *// Позволяет узнать, может ли данный объект  
 // иметь вложенные объекты* public bool IsComposite**();  
 }** public class File : IComponent  
 **{** public float GetSize**()  
 {** *// Тут определяется размер файла.  
 // Для простоты представим,  
 // что все файлы весят 3 условные единицы* return 3**;  
 }** *// Дочерний объект не может иметь вложенные объекты* public bool IsComposite**()  
 {** return false**;  
 }  
 }** public class Folder : IComponent  
 **{** protected List<IComponent> Children = new List<IComponent>**();** */\* Логика подсчета размера папки следующая:  
 вес папки = сумма весов всех объектов внутри  
 Метод рекурсивно пройдется по всем объектам  
 и сложит их вес. \*/* public float GetSize**()  
 {** float totalSize = 0**;** foreach **(**IComponent component in Children**)  
 {** totalSize += component**.**GetSize**();  
 }** return totalSize**;  
 }** public void Add**(**IComponent component**)  
 {** Children**.**Add**(**component**);  
 }** public void Remove**(**IComponent component**)  
 {** Children**.**Remove**(**component**);  
 }** *// Композит может иметь вложенные объекты* public bool IsComposite**()  
 {** return true**;  
 }  
 }** internal static class Program  
 **{** public static void Main**(**string**[]** args**)  
 {** Folder folder = new Folder**();** *// основная папка  
   
 // две вложенные папки* Folder subfolder1 = new Folder**();** folder**.**Add**(**subfolder1**);** Folder subfolder2 = new Folder**();** folder**.**Add**(**subfolder2**);** *// в subfolder1 вложена еще одна папка* Folder subfolder11 = new Folder**();** subfolder1**.**Add**(**subfolder11**);** *// в этих папках есть файлы* File file1 = new File**();** folder**.**Add**(**file1**);** File file11 = new File**();** subfolder1**.**Add**(**file11**);** File file21 = new File**();** subfolder2**.**Add**(**file21**);** File file22 = new File**();** subfolder2**.**Add**(**file22**);** File file111 = new File**();** subfolder11**.**Add**(**file111**);** *// Тогда суммарный вес равен* Console**.**WriteLine**(**$"Weight of subfolder1 = {subfolder1**.**GetSize**()**}"**);** Console**.**WriteLine**(**$"Weight of subfolder2 = {subfolder2**.**GetSize**()**}"**);** Console**.**WriteLine**(**$"Total weight = {folder**.**GetSize**()**}"**);  
   
 }  
 }  
}**

**Применимость**

Используйте паттерн Компоновщик (Composite):

* Если вам нужно представить древовидную структуру объектов;
* Если клиент должен работать с примитивными и составными объектами единым образом.

**Преимущества и недостатки**

**Преимущества:**

* В систему легко добавлять новые примитивные и составные объекты, так как используется общий базовый класс *Component*;
* Код клиента имеет простую структуру – примитивные и составные объекты обрабатываются похожим образом.

**Недостатки:**

* Использование рекурсии.

**Отношения с другими паттернами**

* *Строитель* позволят пошагово создавать дерево *Компоновщика*;
* *Цепочка обязанностей* часто используется вместе с *Компоновщиком* для передачи запроса от дочерних компонентов к родителям;
* Для обхода дерева *Компоновщика* используется *Итератор*;
* С помощью *Посетителя* можно выполнить какое-то действие над всем деревом *Компоновщика*;
* *Компоновщик* и *Декоратор* имеют похожие структуры классов из-за того, что оба построены на рекурсии. Однако *Декоратор* оборачивает только один объект, а узел *Компоновщика* может иметь много дочерних объектов. Также *Декоратор* добавляет вложенному объекту новую функциональность, а *Компоновщик* не добавляет, но «суммирует» результаты всех своих детей.

**Задание**

Разработать UML диаграммы (диаграмму классов и диаграмму последовательности) и с помощью паттерна Компоновщик решить следующую задачу.

Требуется обеспечить контроль загрузки и готовности к отправлению самолета.

Информация о загрузке самолета представлены в таблице ниже.

|  |  |  |
| --- | --- | --- |
| **Тип пассажиров** | **Количество пассажиров, чел.** | **Допустимый вес багажа, кг.** |
| Пилот | 2 | Не имеют багажа |
| Стюардесса | 6 | Не имеют багажа |
| Пассажир первого класса | Не более 10 | Без ограничений |
| Пассажир бизнес-класса | Не более 20 | Не более 35 |
| Пассажир эконом класса | 150 | Не более 20 |

Таблица . Информация о загрузке самолета

Багаж пассажиров от 5 до 60 килограмм.

Есть максимально допустимая загрузка самолета багажом. При превышении – багаж снимается с рейса. Причем снять багаж можно только у пассажиров эконом класса.

В данной задаче удобно ввести следующие объекты:

* Примитивные объекты – пассажир, пилот и стюардесса;
* Составные объекты – первый класс, бизнес-класс, эконом класс.

В результате работы программы должна быть создана карта загрузки самолета с указанием перевеса багажа и информация о багаже, снятом с рейса.

**Требования к отчету**

Отчет к лабораторной работе должен содержать код работающей программы, а также результат выполнения программы.

**Вопросы**

1. Объясните целесообразность применения паттерна для решения задачи лабораторной работы.
2. Какие паттерны являются родственными паттерну *Компоновщик*?