# Паттерн Композит (Composite)

**Назначение:** компонует объекты в древовидные структуры для представления иерархий часть-целое. Позволяет клиентам единообразно трактовать индивидуальные и составные объекты.

## Мотивация

Во многих приложениях часто возникает потребность в древовидных структурах данных, одни узлы которых являются "листовыми", а другие – содержат "потомков" - дочерние узлы. При этом часто возникает потребность в "унифицированном протоколе" работы с такими структурами данных, когда интерфейс объектов является одинаковым для одиночных и составных узлов. Композитные структуры активно применяются для создания элементов управления пользовательского интерфейса, деревьев выражений и при решении ряда других задач.

При импорте лог файлов может возникнуть необходимость задавать бизнес-правила, которые будут определять, нужно ли сохранять запись или нет. Данные правила могут быть простыми: проверять важность записи по свойству Severity, или отвергать очень старые записи, свойство DateTime которого меньше определенной величины. Но поскольку такие правила должны накладываться одно на другое, то рано или поздно возникнет потребность в композиции этих правил.

На помощью может прийти DDD (\*) и паттерн Спецификация, который для данной задачи будет выглядеть так (рис. 4.1):

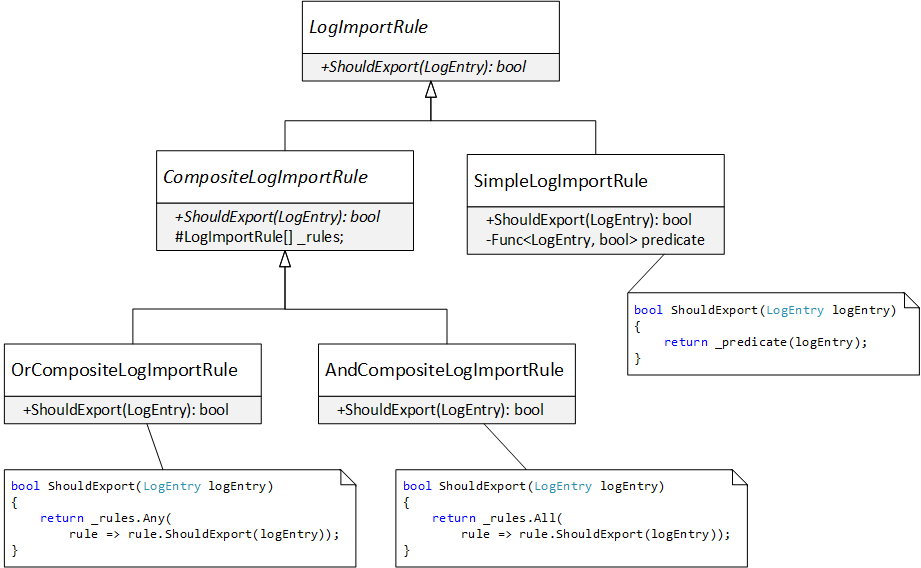


Рисунок 4.1 - Использование паттерна Спецификация для импорта лог-файлов

(\*) DDD - Domain-Driven Design, Проблемно-ориентированное проектирование, с которым можно познакомиться в замечательной книге Эрика Эванса "Domain-Driven Design: Tackling Complexity in the Heart of Software".

Спецификация позволяет гибко комбинировать бизнес-правила, создавая более сложные правила из набора простых условий. Правила моделируются классом LogEntryRule и его наследниками. Простые правила (SimpleLogEntryRule) определяются на основе предиката Func<LogEntry, bool>, переданного пользователем. Составные правила моделируются классом CompisiteLogImportRule и его наследниками. Самыми простыми составными правилами являются AndCompositeLogImportRule и OrCompositeLogImportRule, которые объединяют переданный набор правил по "И" и по "ИЛИ" соответственно.

Для удобства работы, можно создать класс LogRuleFactory, с набором фабричных методов для создания и объединения правил (листинг 4.1):

public static class LogRuleFactory  
{  
 public static LogImportRule Use(Func<LogEntry, bool> predicate)  
 {  
 return new SingleLogImportRule(predicate);  
 }  
  
 public static LogImportRule Or(this LogImportRule left, LogImportRule right)  
 {  
 return new OrLogImportRule(left, right);  
 }  
  
 public static LogImportRule And(this LogImportRule left, LogImportRule right)  
 {  
 return new AndLogImportRule(left, right);  
 }  
  
 public static LogImportRule RejectOldEntriesWithLowSeverity(TimeSpan period)  
 {  
 return   
 Use(le => le is ExceptionLogEntry)  
 .Or(Use(le => (DateTime.Now - le.EntryDateTime) > period)  
 .And(Use(le => le.Severity >= Severity.Warning)))  
 .Or(Use(le => (DateTime.Now - le.EntryDateTime) <= period));  
 }  
}

Листинг 4.1 - Фабричный метод для создания правил сохранения записей

Фабричные методы, такие как Use, Or и End предназначены для создания встроенного языка, который позволяет получить более читабельные правила. Основной фабричный метод - RejectOldEntriesWithLowSeverity, создает составное правило, которое позволит экспортировать записи с исключениями, но отвергать старые записи с низким приоритетом.

Теперь, осталось написать несколько тестов, которые покажут, что данная реализация работает (листинг 4.2):

var rule = LogRuleFactory.RejectOldEntriesWithLowSeverity(  
 TimeSpan.FromDays(7));  
   
LogEntry logEntry = new ExceptionLogEntry();  
Assert.IsTrue(rule.ShouldExport(logEntry));  
  
logEntry = new SimpleLogEntry(){EntryDateTime = DateTime.Now.AddDays(-10)};  
Assert.IsFalse(rule.ShouldExport(logEntry));  
  
logEntry.Severity = Severity.Critical;  
Assert.IsTrue(rule.ShouldExport(logEntry));  
  
logEntry = new SimpleLogEntry()  
{  
 EntryDateTime = DateTime.Now.AddDays(-5),   
 Severity = Severity.Debug  
};  
Assert.IsTrue(rule.ShouldExport(logEntry));

Листинг 4.2 - Тесты правила сохранения лог-файлов

**Паттерн Композит позволяет использовать составные объекты так же, как и одиночные объекты, что делает код использования более простым и понятным**.

## Классическая диаграмма классов паттерна Композит

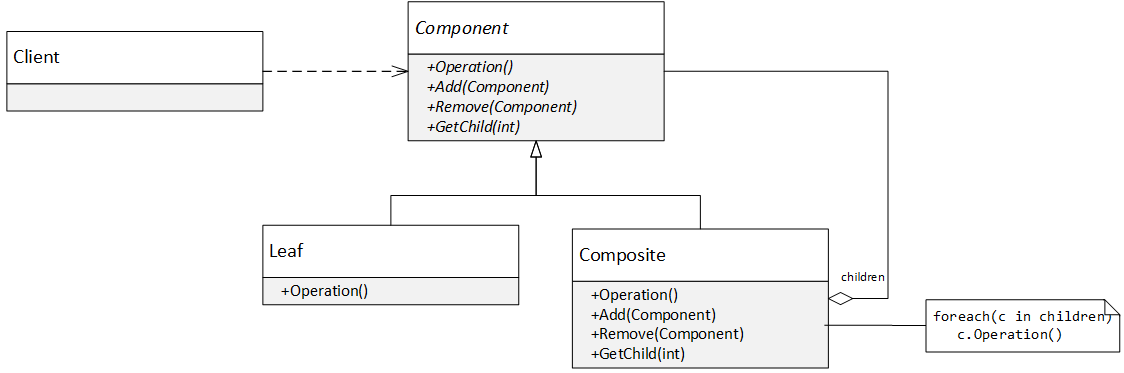


Рисунок 4.2 - Классическая диаграмма классов паттерна Композит

**Участники**

* Component (LogEntryRule) - базовый класс компонента. Содержит операцию (ShouldExport), а также может содержать операции по добавлению/удалению компонентов.
* Composite (CompositeLogImportRule) - составной компонент, который делегирует выполнение основной операции всем дочерним компонентам.
* Leaf (SimpleLogImportRule) - одиночный компонент, который не может содержать дочерних элементов.
* Client - потребитель компонента, который работает с одиночными и составными объектами единообразным способом.

## Обсуждение паттерна Композит

### Интерфейс составного объекта

При работе с паттерном Композит возникает вопрос о том, как формировать составные объекты и каким должен быть интерфейс базового класса Component: должен ли он содержать операции по добавлению/удалению компонентов или нет.

Здесь мы сталкиваемся с компромиссом: чем жертвовать, согласованностью интерфейса компонентов или безопасностью. С одной стороны, клиенты должны работать с простыми и составными объектами единообразным образом, что делает разумным добавление операций Add/Remove в базовый класс Component. С другой стороны, эти методы не могут быть нормально реализованы в классе Leaf, а значит такой дизайн будет нарушать принцип подстановки Лисков (\*).

(\*) Сноска: о принципе подстановки Лисков речь пойдет в четвертой части книги.

Есть три варианта решения этой проблемы, каждый со своими достоинствами и недостатками.

1. **Использование фабричных методов**. Самый простой вариант решить проблему согласованности - сделать классы компонентов неизменяемыми. В этом случае составной объект будет формироваться фабричным методом или конструктором, а необходимость в методах Add/Remove полностью пропадет.
2. **Методы Add/Remove находятся в составном компоненте**. Если формировать составные объекты с помощью конструкторов не удобно, но процесс формирования и использования компонентов четко разделен, то разумно поместить операции по добавлению/удалению компонентов в класс CompositeComponent. В этом случае, часть клиентов будет знать о классе CompositeComponent, но остальные клиенты будут использовать классы Component с более простым интерфейсом.
3. **Методы Add/Remove находятся в базовом компоненте**. Если же базовым сценарием является составной объект, а одиночный компонент является частным случаем, то гораздо проще добавить операции Add/Remove в базовый класс. В этом случае, операции класса Leaf могут генерировать исключение InvalidOperationException или просто ничего не делать, в зависимости от того, является ли вызов операции Add на одиночном компоненте ошибкой или нет.

## Применимость

Композит - это относительно низкоуровневый паттерн проектирования, который лежит в основе других паттернов. Команды объединяются в составные команды, декоратор является составным объектом с одним дочерним элементом, посетитель очень часто обходит составные объекты иерархической формы.

Паттерн Композит применяется для моделирования иерархических структур данных, простые элементы которых объединяются в более сложные компоненты. Паттерн Композит позволяет работать с такими объектами унифицированным образом, скрывая от клиента разницу между одиночным и составным объектом.

## Примеры в .NET Framework

Существует ряд предметных областей, в которых паттерн композит используется практически постоянно. Каждый из представленных ниже классов является примером использования паттерна Композит:

* Построение компонентов пользовательского интерфейса: System.Windows.Forms.Control в Windows Forms, FrameworkElement в WPF, CompositeControl в ASP.NET.
* Работа с Xml: классы XmlNode и XElement.
* Деревья выражений: класс Expression.