МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Кафедра вычислительной техники

Курсовая работа по дисциплине:
«»
на тему:
«Шаблон проектирования компоновщик»

Группа:
Студенты: Преподаватель: Романов Е.Л.

Новосибирск, 2014

Содержание

1 Теоретические сведения

Паттерн «компоновщик» (*composite*) относится к классу структурных. Паттерн компонует объекты в древовидные структуры, позволяющие представить иерархии вида часть-целое. В такой иерархии объекты могут быть условно разделены на два больших класса: индивидуальных и составных. Индивидуальные объекты (примитивы) представляют собой листовые вершины древовидной иерархии, а составные (контейнеры), напротив, являются внутренними узлами и могут включать в себя другие объекты, как индивидуальные, так и составные. Программа должна по-разному обращаться с примитивами и контейнерами, хотя пользователь, как правило, работает с ними обоих типов единообразно. Необходимость различать эти типы усложняет программу. Компоновщик решает эту проблему, позволяя клиенту единообразно работать и с примитивами, и с контейнерами.

Хорошей демонстрацией паттерна, приведенной в [1], является программа работы с графикой. Такая программа позволяет пользователю сгруппировать мелкие элементы рисунка (линии, геометрические фигуры, надписи и т.д.) в более крупные компоненты, их – в еще более крупные и т.д. Налицо иерархическая структура, описанная в предыдущем абзаце. Рассмотрим на ее примере работу паттерна компоновщик. Общая диаграмма классов для паттерна приведена на рисунке 1.

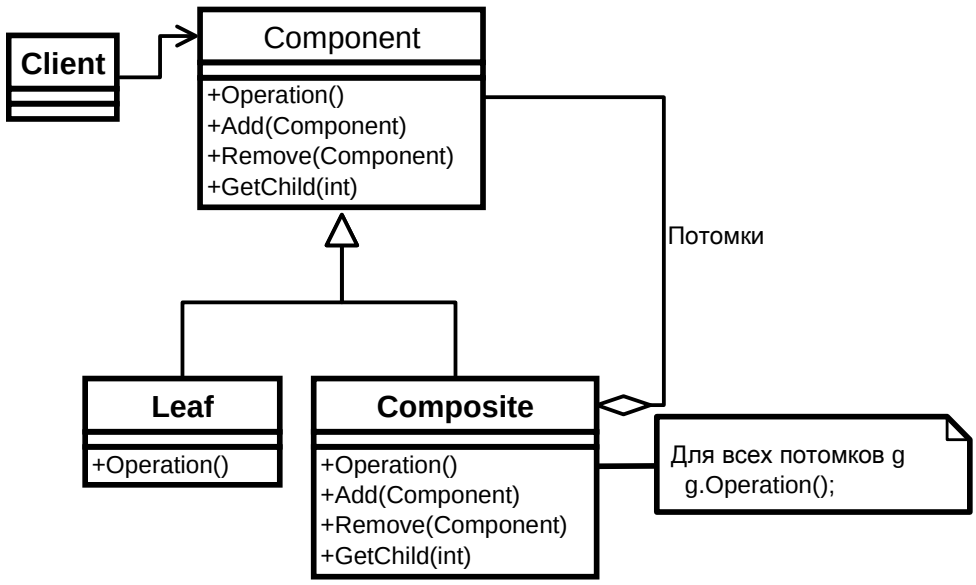

Рисунок 1 – Диаграмма классов паттерна компоновщик

Ключевым является абстрактный класс Component, представляющий одновременно и примитивы, и контейнеры. В нем объявлены операции, специфичные для всех объектов (в нашем примере такой операцией может быть операция отрисовки графического

объекта), а также общие для всех составных объектов, например операции для управления потомками. [1] перечисляет следующие функции этого класса:

- объявление интерфейса для копонуемых объектов;
- предоставление подходящей реализации операций по умолчанию, общей для всех классов;
- объявление интерфейса для доступа к потомкам;
- определение интерфейса для доступа к родителю компонента (необязательно).

От класса Component наследуются примитивы (в случае графического редактора сюда можно отнести классы Line, Rectangle, Text и т.п.; на диаграмме они все представлены классом Leaf) и составные объекты (класс Composite на диаграмме). Функции класса Leaf:

- представление листовых узлов в дереве композиции;
- определения поведения примитивов;

Функции класса Composite:

- определение поведения компонентов, имеющих потомков;
- хранение компонентов-потомков;
- реализация относящихся к управлению потомками операций класса Component.

Клиент взаимодействует с объектами, используя интерфейс класса Component. Если получателем запроса от клиента является объект Leaf, он и обрабатывает запрос. Если же получателем запроса является составной объект, то он, как правило, перенаправляет запрос своим потомкам, возможно, выполняя некоторые дополнительные операции (например, при отрисовке составного объекта в графическом редакторе вызываются методы отрисовки для всех потомков этого объекта).

[1] определяет следующие результаты применения паттерна:

- компоновщик *определяет иерархии классов, состоящие из примитивных и составных объектов*. Из примитивных объектов можно составлять более сложные, которые, в свою очередь, участвуют в еще более сложны композициях и т.д. Любой клиент, ожидающий примитивного объекта, может работать и с составным;

- компоновщик *упрощает архитектуру клиента*. Клиенты могут единообразно работать, как с примитивами, так и с составными объектами. Обычно клиенту неизвестно, работает ли он с индивидуальным или с составным объектом. Это упрощает код клиента, поскольку нет необходимости писать функции,

ветвящиеся в зависимости от типа объекта, с которым они работают;

- компоновщик *облегчает добавление новых видов компонентов*. Новые подклассы классов Composite и Leaf будут автоматически работать с уже существующими структурами и клиентским кодом.
- компоновщик *способствует созданию общего дизайна*.

2 Описание разработки

2.1 Общее описание

В рамках курсовой работы предполагается реализация паттерна компоновщик на примере JSON-парсера. JSON представляет собой текстовый формат обмена данными, основанный на JavaScript. Данные в JSON могут иметь один из следующих типов[2]:

- число;
- строка (заключается в кавычки);
- логическое значение (представляется литералами true и false);
- null-значение (представляется литералом null);
- JSON-объект. Представляет собой набор пар ключ-значение, заключенный в фигурные скобки. Ключ является строкой, заключенной в кавычки, значение – данные любого JSON-типа (в том числе и JSON-объекты и JSON-массивы); Ключ отделяется от значения двоеточием, а между парами ставится запятая;
- JSON-массив. Представляет собой упорядоченный набор данных любых JSON-типов (в том числе JSON-объектов и JSON-массивов), заключенный у квадратные скобки. Между элементами массива ставится запятая.

Как можно видеть из этого описания, JSON-документ имеет иерархическую структуру, в которую входят как примитивные объекты (строки, числа, логические значения и null), так и контейнеры (массивы или объекты). Корнем иерархии обязательно должен являться контейнер, т.е. хранить в JSON-документе просто значение какого-то примитивного типа нельзя, оно обязательно должно находиться в массиве или объекте.

Подобная иерархическая структура делает представление JSON-документа в памяти отличной иллюстрацией паттерна компоновщик. Действительно, в этом случае имеются и два типа объектов (составные и примитивные), и потребность в едином интерфейсе для работы с ними. Ниже будет приведено описание JSON-парсера, разработанного с

использованием паттерна компоновщик.

2.2 Пакет jsonparser

Парсер целиком реализован в рамках этого пакета. Диаграмма классов пакета приведена на рисунке 2.

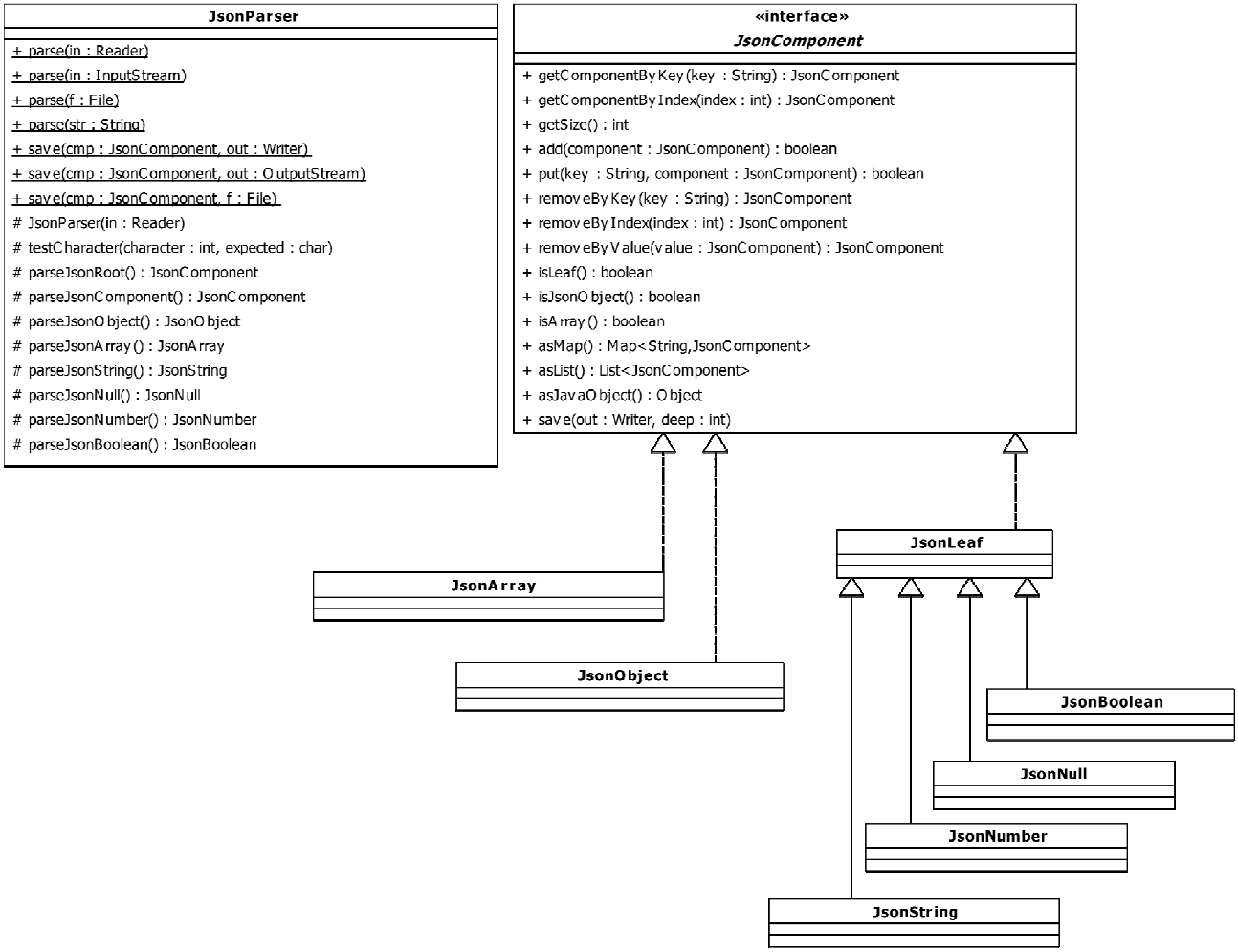

Рисунок 2 – Диаграмма классов пакета `jsonparser`

Как можно видеть, структура классов близка к структуре, приведенной на рисунке 1. Ключевым является интерфейс `JsonComponent`. Интерфейс представляет как примитивы, так и контейнеры. Кратко рассмотрим, какие методы входят в интерфейс.

Методы работы с потомками (листинг 1). Данные методы могут быть вызваны не только для составных объектов, но и для примитивов. Такой вызов обеспечивает прозрачность, но не несет семантической нагрузки, поэтому необходимо определить реакцию примитивных компонентов на него. При разработке рассматривались два варианта: выбрасывание исключения при попытке обращения к потомкам примитива и простое игнорирование такого запроса. Первый вариант более безопасен, однако он требует значительного усложнения кода – клиент будет вынужден реализовывать

обработку таких исключений и формировать ветвления, что фактически сведет на нет преимущества паттерна компоновщик. Поэтому был выбран второй вариант. В приведенной реализации примитивы способны принимать запросы контейнеров, однако игнорируют их, возвращая null в операции получения/удаления потомка и false (провал операции) в операции добавления. Также запрос будет проигнорирован, если контейнер принявший его не поддерживает такое действие (например, попытка получить элемент по ключу от массива)

```
/**
 * Возвращает потомка по ключу. Если такого ключа нет или
 * объект не поддерживает получение потомков по ключу,
 * возвращает null
 */
public JsonComponent getComponentByKey(String key);

/**
 * Возвращает потомка по индексу. Если такого индекса нет
 * или объект не поддерживает получение потомков по индексу,
 * возвращает null
 */
JsonComponent getComponentByIndex(int index);

/**
 * Количество потомков в составном объекте
 */
int getSize();

/**
 * Добавление компонента
 */
boolean add(JsonComponent component);

/**
 * Добавление компонента по ключу. Заменяет уже существующее
 * значение и НЕ возвращает его.
 */
boolean put(String key, JsonComponent component);

/**
 * Удаление объекта по ключу
 */
JsonComponent removeByKey(String key);

/**
 * Удаление объекта по индексу
 */
JsonComponent removeByIndex(int index);

/**
 * Удаление объекта по значению (первого)
 */
JsonComponent removeByValue(JsonComponent value);
```

Листинг 1. Методы работы с потомками интерфейса `JsonComponent`

Методы проверок (листинг 2). Данные методы позволяют проверить, является ли конкретный компонент примитивом, JSON-объектом или JSON-массивом.

```
/**
 * Проверяет, является ли компоннет листовым
 */
boolean isLeaf();

/**
 * Проверяет, является ли компонент объектом JSON
 */
boolean isJsonObject();

/**
 * Проверяет, являтеся ли компонент массивом JSON
 */
boolean isArray();
```

Листинг 2. Методы проверок

Методы преобразования (листинг 3). Данные методы позволяют получить Java-представления того или иного JSON-элемента. Например, строка будет преобразована в String, объект – в Map и т.д. Также эти методы позволят получить потомков контейнера в виде списка или карты.

```
/**
 * Представляет элемент как неизменяемую карту.
 * Возвращает null, если объект не может быть представлен в
 * виде карты
 */
Map<String, JsonComponent> asMap();

/**
 * Представляет элемент как неизменяемый список.
 * Возвращает null, если объект не может быть представлен в
 * виде списка
 */
List<JsonComponent> asList();

/**
 * Представляет элемент как неизменяемый объект Java.
 * Возваращает карту или список для композитов, объект соотв.
 * типа для листовых и null для null
 */
Object asJavaObject();
```

Листинг 3. Методы преобразования

Метод save (листинг 4). Метод позволяет сохранить иерархию, корнем которой является данный компонент в поток.

```
/**
 * Сохранение элемента в поток. Функция выводит данные
 * рекурсивно.
 * Каждый объект не делает ведущих отступов, предполагая, что
 * их за него формирует его предок.
```

```
     * Точно также не формируются хвостовые отступы и переносы
     * строк - это тоже задача предка.
     * @param out поток, в который следует сохранить элемент
     * @param deep глубина вложенности. Определяет число символов
     *         табуляции, которые будут предварять данный элемент.
     *         Определяется глубина вложенности <b>данного
     *         выводимого объекта</b>.
     *        Для корня - 0, для элементов корня - 1 и т.д.
     */
    void save(Writer out, int deep) throws IOException;
```

Листинг 4. Метод сохранения

Интерфейс `JsonComponent` реализуется абстрактным классом `JsonLeaf`. Данный класс является предком всех примитивов и предоставляет общие для них реализации методов (например, реакция на методы работы с потомками). От `JsonLeaf` наследуются классы примитивных типов данных:

- `JsonNumber` – число
- `JsonString` - строка
- `JsonBoolean` – логическое значение
- `JsonNull` – null-значение

Эти классы являются обертками соответствующих типов данных, представляемых Java (за исключением класса `JsonNull` - он не хранит данных и используется просто как заглушка).

Составные объекты представлены классами `JsonObject` и `JsonArray`, реализующими интерфейс `JsonComponent`. `JsonObject` хранит данные в `LinkedHashMap`, где в качестве ключей выступают объекты класса String, а в качестве значений – объекты интерфейса `JsonComponent`. `JsonArray` хранит данные в `ArrayList`, параметризованном интерфейсом `JsonComponent`.

Непосредственно операция парсинга JSON реализована в классе `JsonParser`. Данный класс предоставляет набор статических методов, позволяющих выполнять сериализацию и десериализацию JSON (листинг 5).

```
    /**
     * Парсит из Reader'a
     */
    public static JsonComponent parse(Reader in) throws
              IOException, JsonParseException

    /**
     * Парсит из Stream'a
     */
    public static JsonComponent parse(InputStream in) throws
              IOException, JsonParseException
```

```
    /**
     * Парсит из файла
     */
    public static JsonComponent parse(File f) throws IOException,
              JsonParseException

    /**
     * Парсит из строки
     */
    public static JsonComponent parse(String str) throws
              IOException, JsonParseException

    /**
     * Сохраняет в Writer
     */
    public static void save(JsonComponent cmp, Writer out) throws
              IOException

    /**
     * Сохраняет в Stream
     */
    public static void save(JsonComponent cmp, OutputStream out)
              throws IOException

    /**
     * Сохраняет в файл
     */
    public static void save(JsonComponent cmp, File f) throws
              IOException
```

Листинг 5. Статический инетрфейс парсера

Непосредственно парсинг реализован в нестатических методах данного класса. которые объявлены как `protected`. Таким образом пользователь общается с парсером через статический интерфейс, а уже статические методы создают объект `JsonParser` и подают ему на вход данные.

Объект класса `JsonParser` хранит в себе поток, из которого читаются данные. Поток оборачивается в класс `LineColumnNumberReader`, унаследованный от стандартного `LineNumberReader`. Данный класс позволяет отслеживать количество считанных строк и номера столбцов, а также выполнять дополнительные сервисные операции, например пропускать пробельные символы. Методы, добавленные в класс приведены в листинге 6.

```
public static class LineColumnNumberReader extends
              LineNumberReader{

    /**
     * Пропускает все пробельные символы пока не
     * наткнется на что-то иное (в т.ч. конец файла)
     */
    public void skipWhiteSpaces() throws IOException
```

```
    /**
     * выполняет skipWhiteSpaces, а затем читает и возвращает
     * символ
     */
    public int skipWhiteSpacesAndRead() throws IOException

    /**
     * Читает следующий символ, но не извлекает его из потока
    public int peek() throws IOException

}
```

Листинг 6. Методы класса `LineColumnNumberReader`

Непосредственно парсинг производится методами класса JsonParser, ориентированными на различные типы данных (листинг 7).

```
/** Проверяет очередной символ и кидает исключение,
 *  если что-то пошло не так */
protected void testCharacter(int character, char expected) throws
              JsonParseException

/** Парсит корень Json-документа ({} или []) */
protected JsonComponent parseJsonRoot() throws IOException,
              JsonParseException

/** Парсит произвольный компонент Json. Анализирует тип компонента
              и вызывает соовтветствующий метод  */
protected JsonComponent parseJsonComponent() throws IOException,
              JsonParseException

/** Парсит json-объект ({}) */
protected JsonObject parseJsonObject() throws IOException,
              JsonParseException

/** Парсит json-массив ([]) */
protected JsonArray parseJsonArray() throws IOException,
              JsonParseException

/** Парсит json-строку */
protected JsonString parseJsonString() throws IOException,
              JsonParseException

/** Парсит null-значение */
protected JsonNull parseJsonNull() throws IOException,
              JsonParseException

/** Парсит json-число */
protected JsonNumber parseJsonNumber() throws IOException,
              JsonParseException

/** Парсит логическое значение */
protected JsonBoolean parseJsonBoolean() throws IOException,
              JsonParseException
```

2.3 Пакет jsonparser.exception

Пакет включает в себя классы для работы с исключительными ситуациям, возникающими в процессе парсинга. В пакет входят 4 класса, приведенные ниже.

- `JsonParseException` – базовый класс для всех исключений, связанных с парсингом
- `JsonUnexpectedCharacterException` – выбрасывается, если парсер встретил символ, который не может находиться в данном месте
- `JsonUnexpectedTokenException` – выбрасывается, если парсер не может распознать элемент документа (например, FALSE вместо false).
- `JsonUnexpectedEndException` – выбрасывается, если парсер встретил конец документа до того, как была считана последняя закрывающая скобка.

Заключение

В ходе работы был изучен паттерн компоновщик. На основе данного паттерна разработан JSON-парсер. Парсер позволяет формировать памяти древовидную иерархию и предоставляет к ней доступ, единообразный как для примитивов, так и для составных объектов.

Применение паттерна компоновщик видится мне целесообразным в случае работы с иерархическими структурами, если требование прозрачности (т.е. единообразия интерфейса работы с примитивами и контейнерами) является более важным, чем требования безопасности кода. В противном случае логичнее будет ограничить список методов интерфейса Component, только теми, которые поддерживаются всеми типами элементов, а для доступа к специфическим возможностям контейнеров и примитивов использовать явные проверки типов.

В настоящее время паттерн широко применяется в различных программных системах. Классическим примером такого применения можно считать системы графического пользовательского интерфейса (например, Java Swing), в которых от класса Component наследуются как примитивы, например, кнопки и текстовые поля, так и контейнеры, например, окна и панели. Такая система позволяет группировать элементы GUI различными способами и управлять ими с помощью единообразных процедур.

Список литературы

1. Гамма Э., Хелм Р., Джонсон Р. ,Влиссидес Дж. Приемы объектно-ориентированного проектирования. Паттерны проектирования . — СПб: Питер, 2001 . — 368с. : ил .
2. Формат JSON. Современный учебник JavaScript. [Электронный ресурс] . - электрон. текст. дан . – режим доступа: http://learn.javascript.ru/json
3. Шилдт Г., Java. Полное руководство , 8-е изд. : Пер. с англ. – М. : ООО «И.Д. Вильямс», 2013. – 1104 с. : ил.