Софийски университет “Св. Климент Охридски”

ФМИ

Курсов проект

Тема:

JSON Parser

Изготвил:

Веселин Ангелов

София

2022

Глава 1.

Увод

1.1 Описание и идея на проекта

Проектът JSON Parser представлява програма, която валидира и запазва текст в йерархия съответстваща на формата JSON. JSON е текстово базиран отворен стандарт създаден за човешки четим обмен на данни. Произлиза от скриптовия език JavaScript, за представя прости структури от данни и асоциативни масиви, наречени обекти. Този проект е създаден следвайки формата на JSON описан в RFC 8259.

1.2 Цел и задачи на разработката

Целта на проекта е да се валидира текст спрямо текстовия формат JSON. При подаване на файл, програмата проверява дали файла е валиден. Ако не е съобщава за грешките. Ако проверките преминат успешно, програмата прочита файла отново и го съхранява във подходящата структура.

Задачите за проекта бяха разпределени на четири. Първо разучаване на JSON формата, и разписване на задание, по което да се водя при разработка на програмния код. Втората задача беше валидацията на данните, като тя беше разпределена на подзадачи, които отговаряха на типовете данни. Третата задача беше да запазя данните в подходящата структура, а последната да имплементирам командите за манипулация на въпросната структура.

1.3 Структура на документацията

В документацията е разгледана идеята на проекта, предметната област, проектирането му и реализацията.

Глава 2.

Преглед на предметната област

2.1 Основни използвани дефиниции и концепции

Използвани са основните концепции на обектно ориентираното програмиране - енкапсулация, полиморфизъм, наследяване и абстракция.

2.2 Дефиниране на проблеми и сложност на поставената задача

Първият проблем е как ще бъде валидиран текстовият файл - скоби, кавички, знаци, които могат да “счупят” програмата.
Вторият проблем е създаването на обектите от файла. Дублирането на ключовете в обектите не се счита за проблем, а само би предизвикало появата на предупреждение при създаването на обекта. Този проблем няма точен начин за решаване по стандарт. Различните имплементации го третират по различен начин. За да може един JSON да е валиден при различни имплементации е добре да няма дубликати.

2.3 Подходи и методи за решаване на поставените проблеми

Подходът към първият проблем е следният - един JSON документ, може да се състои от обикновен тип стойности, обект или масив.
Стойностите са няколко типа - символен низ, число или един от трите литерали: *false*, *true* или *null*. Във файла може да има само един главен елемент, който е един от всички типове изброени горе. Ако типът е обект или масив, в него може да има вложени други стойности. Обектите се състоят от двойки - ключ и стойност, а масивите от множество стойности. При валидиране се следят скобите за обектите и масивите, кавичките за

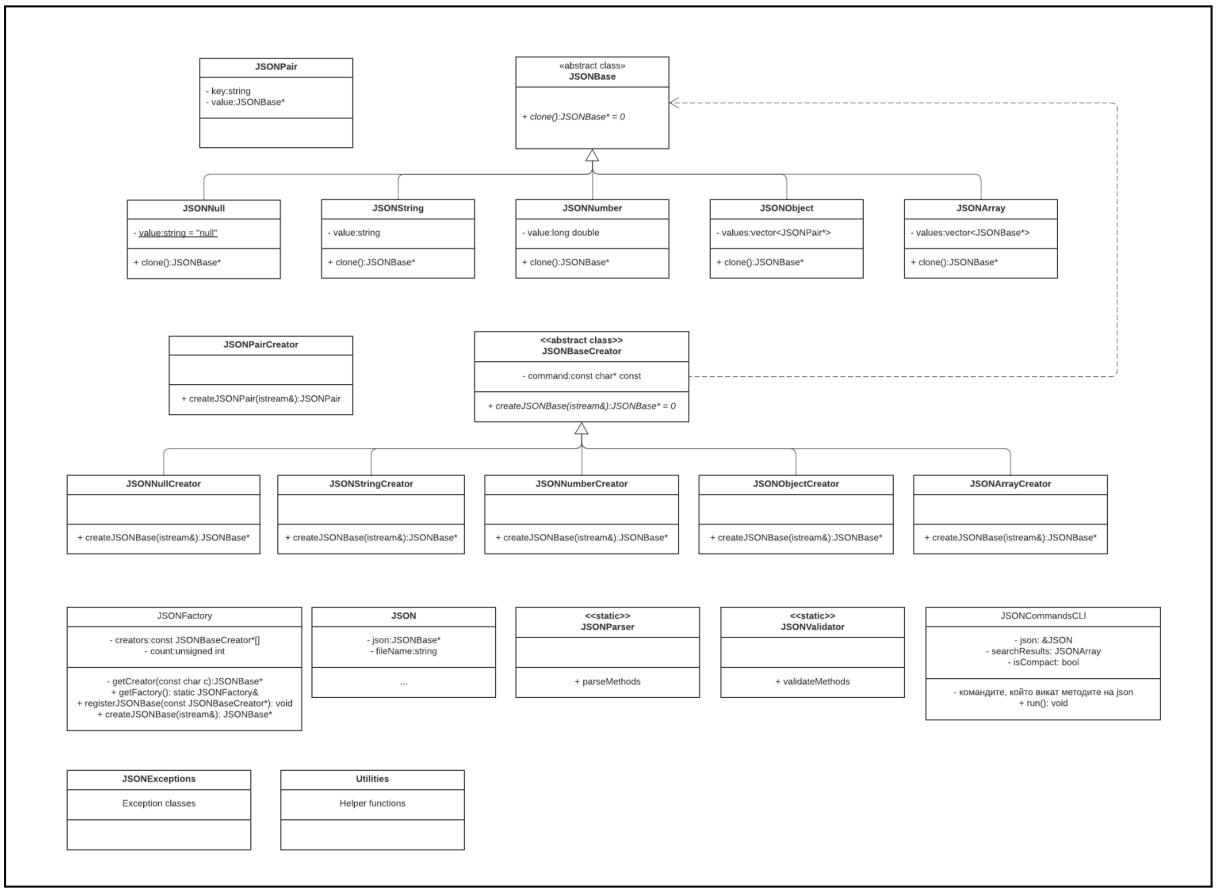
низовете, цифрите за числата (също така и минус), а за литералите се следят първите им букви.

За вторият проблем подходът е подобен на валидацията но се изпускат проверките тъй като вече знаем, че всичко с файла е наред. Използван е обектно ориентирания похват “Фабрика”. Проблемът с дубликатите съм избрал да го реша като презаписвам стойността на дубликата с последно срещнатата двойка с еднакъв ключ. По този начин е сигурно, че JSON текста ще бъде валиден.

Глава 3.

Проектиране

3.1 Обща архитектура



Фигура 1. - Диаграма с класовете на проекта

Архитектурата на проекта е спрямо йерархията на JSON формата. Имам един базов абстрактен клас, който е за типовете стойности, които може да има в един JSON. Той се наследява от класове, които представляват тези типове. Също така има клас JSONPair, който съдържа в себе си двойка от ключ, който е символен низ и стойност, която е указател към JSONBase. Главният клас, който съхранява JSON обекта е с наименование JSON, в него има указател към базовият клас. От този клас започва програмата. Той обединява в себе си и използва другите помощни класове - JSONValidator, JSONParser, JSONFactory. При създаването на обекта от тип JSON, в конструктора му се извикват последователно валидирането на файла, ако то премине без грешки, следва запазването му във обекти от коректния тип, посредством JSONFactory.

JSONCommandsCLI представлява абстракция върху JSON обекта. Чрез този се обработва потребителският вход и той определя, коя функция ще се извика, като преди това валидира данните да са коректни. При коректни данни, но не изпълняване за някое условие за коректност на дадена функция тя генерира изключение, което се прихваща от JSONCommandsCLI, за да може да продължи изпълнение на други команди.

3.2 Най-важни извадки от кода

```
JSON::JSON(const std::string &fileName) : fileName(fileName)
{
   std::ifstream file;

//    1. OPEN file
   file.open(this->fileName);
   if (!file.is_open()) throw std::runtime_error("File failed");

   try
   {
//    2. Validate json
       JSONValidator::validate(file);
//    3. Move cursor to the beginning of file
       file.clear();
       file.seekg(0);
//    4. Parse json
       parseJSON(file);
//    5. Close file
       file.close();
   }
   catch (...)
   {
       delete json;
```

```
        file.close();
        throw;
    }
}
```

Фигура 2. - Конструкторът на JSON класа

На **Фигура 2.** е изобразен кода, който изпълнява главната функционалност. Първо отваря файла за четене, след което валидира файла, връща се в началото му, като преди това възстановява доброто му състояние и създава обекта.

```
void JSONValidator::validateType(std::istream& in)
{
    char c;
    in >> c;

    in.unget(); // Unget the last char because we need the data for further
validating

    if (c == '-' || (c >= '0' && c <= '9')) validateNumber(in);
    else if (c == 't' || c == 'f') validateBool(in);
    else if (c == 'n') validateNull(in);
    else if (c == '"') validateString(in);
    else if (c == '{') validateObject(in);
    else if (c == '[') validateArray(in);
    else throw InvalidTypeException("Expected correct JSON value.", in.tellg());
}
```

Фигура 3. - Функцията, която определя типа на стойностите

Тази функция определя какъв тип е дадена стойност и съответно определя, кой следващ валидатор да изпълни, ако бъде изпълнен валидаторът за обект или масив, ако в тях има други елементи тази функция ще бъде викната отново (рекурсивно), за да провери типа на стойностите в тях.

```
const JSONBaseCreator* JSONFactory::getCreator(const char c) const
{
    for (int i = 0; i < count; ++i)
    {
        if (strchr(creators[i]->getKeyChars(), c) != nullptr)
        {
            return creators[i];
        }
    }
```

```
    return nullptr;
}

JSONFactory& JSONFactory::getFactory()
{
    static JSONFactory theFactory;
    return theFactory;
}

void JSONFactory::registerJSONBase(const JSONBaseCreator *creator)
{
    creators[count++] = creator;
}

JSONBase* JSONFactory::createJSONBase(std::istream &in)
{
    char c;
    in >> c;

    const JSONBaseCreator* creator = getCreator(c);

    if (creator)
    {
        in.unget();
        return creator->createJSONBase(in);
    }

    return nullptr;
}
```

Фигура 4. - Фабриката, която създава обектите

Фабриката разпознава даден обект както валидатора, спрямо първия символ на елемента, като проверява във масив от символи дали, съвпада с някой от тях и така определя, какъв тип обект да създаде.

Глава 4.

Реализация, тестване

4.1 Реализация на класове

Базовият клас за типовете - JSONBase

Наследници:

- JSONObejct
- JSONArray
- JSONNumber
- JSONString
- JSONBool
- JSONNull

Специален клас за двойка ключ и стойност - JSONPair

Помощни класове:

- JSONValidator
- JSONParser
- JSONExceptions
- JSONFactory
- JSONCommandsCLI
- Utilities

4.2 Управление на паметта и алгоритми. Оптимизации

Оптимизирано е при създаване на обект, който наследява JSONBase, в JSONPair, направо се добавя по указател, а не се копира, защото обекта създаден от фабриката е указател и няма смисъл да бъде копиран по стойност, а директно се добавя, същото е направено и за JSONObject класа, там се добавят двойките по указател във вектора, вместо по стойност, което не налага извикването на копиращ конструктор, когато се добавя и когато размерът на масива трябва да се увеличи.

4.3 Планиране, описание и създаване на тестови сценарии

Тестовите сценарии са следните - създаваме файл, в който запазваме JSON текста и го подаваме на обекта JSON, който след това подаваме на JSONCommandsCLI, ако желаем да манипулираме обекта чрез командния интерфейс.

Командите, които обекта поддържа са :

- print (search_result) (<index>) - извежда в терминала целия резултат, ако не са подадени аргументи, ако се подаде аргумент “search_result” се извежда резултата от търсенето с командата “search”, а “index” е опционално, ако искаме само определен елемент от масива с резултата.
- search <key> - търсене по даден ключ
- edit <element> <value> - редактира стойността на елемента, елемента се задава с пълния път до него, стойността се задава като някой от валидните JSON типове
- create <element> <value> - създава елемент със зададената стойност, подобно на редакцията, с разликата, че добавя нов обект, а не го редактира.
- remove <element> - премахване на елемент по подаден път до обекта
- move <from> <to> - премества един елемент на мястото на друг
- alignment (tabbed/compact) - с тази команда указваме дали искаме при запазване във файл, дали текста да е възможно най-компактен или да е лесно четим.
- save (<fileName>) (path/key/search_result) (<path>/<key>/(<index>)) - с тази команда запазваме обекта, има различни опции, ако не се подаде нито една, се записва текущото състояние на обекта във неговият файл. Ако искаме да имаме други опции е задължително да окажем файл, в който да запазим, резултата.

  Има три различни типа резултати. Можем да запазим резултата от търсенето, както и отделен елемент от него, ако желаем, можем да запазим директно във файл обект на определено място (“path”), или обекти, отговарящи на ключа “key”.

Може да се тества програмата чрез подаване на грешен JSON формата, за да се провери коректността на валидирането. Например:

```
{
  "Image": {
```

```
    "Width":  800,
    "Height": 600,
    "Title":  "View from 15th "Floor",
    "Thumbnail": {
       "Url":    "http://www.example.com/image/481989943",
       "Height": 125,
       "Width":  100
    },
    "Animated" : false,
    "IDs": [116, 943, 234, 38793]
  }
}
```

Фигура 5. - Невалиден JSON формат

Ако се подаде валиден JSON документ може да се тестват *parsing*, както и командите.

```
{
  "Image": {
    "Width":  800,
    "Height": 600,
    "Title":  "View from 15th Floor",
    "Thumbnail": {
       "Url":    "http://www.example.com/image/481989943",
       "Height": 125,
       "Width":  100
    },
    "Animated" : false,
    "IDs": [116, 943, 234, 38793]
  }
}
```

Фигура 6. - Валиден JSON формат

Примерно изпълнение на кода:

```
> print
> search Image
> print search_result
> edit Image.IDs[0] 123
> create Test {"a": true}
> print
> move Image.Height Test
> save new.json search_result
> save
> end
```

Глава 5.

Заключение

5.1 Обобщение на изпълнението на началните цели

Заданието на проекта е изпълнено без концепцията за итерация, защото не можах да разбера как трябва да работи.

5.2 Насоки за бъдещо развитие и усъвършенстване

За бъдещо развитие бих довършил итерирането през обекта. Също така бих добавил командите да могат да се изпълняват от текстови файл, освен от терминал.

Използвана литература

https://bg.wikipedia.org/wiki/JSON - страница на уикипедия, от където е взето описанието на JSON

https://www.ietf.org/rfc/rfc8259.html - стандарта, който е следван при създаването на проекта

https://refactoring.guru/design-patterns/factory-comparison - информация за дизайн *pattern* тип “Фабрика”

https://cplusplus.com/ - документация на C++

https://jsonformatter.curiousconcept.com/ - JSON валидатор използван за проверка на валидни и невалидни JSON текстове по време на разработка

Презентации и лекции от курса по ООП във ФМИ