Софийски университет „Св. Климент Охридски“

Факултет по математика и информатика

Курс по Обектно-ориентирано програмиране за специалност Информатика

Летен семестър на учебната 2019/2020 година

JSON-Parser

Автор: Мирослав Николаев Парапанов

ФН:45560

ГРУПА: 1

1. Увод

1.1. Описание и идея на проекта

Проектът е изцяло целево насочен към обработване на и валидиране на информация в JSON формат. Идеята е всеки потребител, когато получи някакви заявка дали ще е от интернет, или от файл тя да бъде класифицирана по дадения формат. Потребителя има възможност да провери дали информацията е в JSON формат(команда : validate), да зададе стойност по даден ключ (ако стойността е в JSON формат също, команда : set <path> <value>), показване на информацията в комадния прозорец (команда : print), търсене по ключ (команда : search <path>), създаване на ключ със стойност(команда : create <path> <value>) и изтриване на стойност (команда : delete <paht>). Въпреки че проектът е в пъти по-опростен от съвременните JSON parser-и, той е функционален и добре структуриран.

1.2. Цел и задачи на разработката

Една от основните цели на задача бе да се построи максимално добре работеща и оптимизационна архитектура. Също така капсуловането на информацията и съхранението под формата на последователен файл също бе предизвикателство .Използвани са множество от идеи, които да поставят едно по-абстрактно виждане върху самото приложение и ,разбира се, те са съвместими с ООП (Обектно-ориентирано програмиране) парадигмите. Използвани са patterns(патърни / модели) за REPL (read-eval-print loop), чрез които се симлифицира за по-нататъшно развитие и опримизиране на проекта. Разделното компилиране, управлението на паметта, глобалните и статичните променливи, и реализирането на помощни класове с цел подобряване на функционалността са задачи, които са имплементирани във приложението.

С цел също да се намали сложността на повечето задачи са използвани помощни функции.

1.3. Структура на Документацията

Първо ще се разгедат архитектурата и йерархията на класовете и как по-точно са свързание тяхните функционалности. Ще покажем как всеки клас си има собствена фунция и се грижи, в зависимост от това как са имплементирани методите му, за входните и изходните данни.

2. Преглед на предметната област

2,1 . Основни дефиниции и концепции, които ще бъдат използвани

2.1.1. Клас

2.1.2. Полиморфизъм

2.1.3. Конструктори, Деструктори

2.1.4. Единично и множествено наследяване

2.1.6. Абстрактни класове

2.1.7. Виртуални и чисто виртуални функции

2.1.8. Файлове

2.1.9. Command Pattern (патърн / модел)

2.1.10. Стилизиране на кода

2.1.11. Smart pointers (умни указатели)

2.1.13. Гит хранилище (GitHub) (miroslavpar)

2.2. Дефиниране на проблеми и сложност на поставената задача:

2.2.1. Работа с последователни файл и в конкретност определянето срещу всеки ключ какъв Json обект седи.

2.2.2. Изготвяне на интерфейс класове, които в бъдеще ще послужат за по-лесно развитие на проекта.

2.2.3. Комуникацията между отделните класове и архитектурата.

2.2.4. Използване на smart pointers (умни указатели) и правенето на такива

2.3. Подходи и методи за решаване на проблемите:

2.3.1. Използване на абстрактни класове за по-абстрактен вид върху приложението

2.3.2. Използване на смарт пойнтъри за по - лесен достъп до класовете команда

2.3.2. Използване на интърфейс-класове за по-нататъшно развитие на проектакато за пример би могло да се даде обработване на XML, HTLM и т.н. файлове

2.4. Потребителски изисквания:

2.4.1. Валидност на входните данни, което означава да се следва примера от точка 1.Увод за правилно използване на командите

2.4.2. Save As командата да се пише слято с малки букви (saveas)

2.4.3 Преди работа с коя да е команда файла да се отваря.

3. Проектиране

3.1. Обща архитектура :

3.1.1. Използван е Command Pattern, който опростява модела за писане. Имаме два абстрактни класа. Единият е ICommand, който се грижи за името на командата и една чисто виртуална функция чрез,която ще се извършва полиморфизмът., а другия е JsonManager, който ще се грижи изцяло за функционалността на данните.

```
class ICommand {
protected:
    std::string commandName;
    std::vector<std::string> arguments;

public:
    ICommand(){};
    ICommand(const std::string& commandNameTemp): commandName(commandNameTemp)
    virtual void execute(TXTPlaneManager &) {};
    void setArguments(std::vector<std::string>& arguments) {
        this->arguments = arguments;
    }
    std::string& convertToString() {
        return commandName;
    }
    virtual ~ICommand() {};
};
```

```
using namespace std;
class JsonManager {
private:
    string dbFile;
    stringstream jsonInfo;
    bool isFileOpened;
    JsonObject* workingJson;
private:
    //The main functionality :
    JsonArray* readArray (stringstream&)const;
    JsonValue* readValue (stringstream&)const;
    JsonObject* readObject(stringstream&)const;
    string readString(stringstream&)const;
    JsonValue* readNumber(stringstream&)const;
    JsonValue* readJson(stringstream&)const;

private:
    // Helping functions:
    void readWhitespace(stringstream&)const;
    bool isWhiteSpace(char)const;
    bool isOperation(char)const;
    bool isDigit(char)const;
    bool readLiteral(stringstream&,const string&)const;
    double geteNumber(char, stringstream&)const;
    double getNumber(char,stringstream&)const;
public:
    JsonManager();
    void openFile(string&);
    bool validateJsonFile();
    void print()const;
    void search(const string&)const;
    void help() const;
    void closeFile();
    void save();
    void saveAs(const string&);
    void exitFromFunction()const;
    void set(const string&, stringstream&);
    void deleteByPath(const string&);
    void create (const string&,stringstream&);
};
```

Използван е също така смарт пойнтър и в конкретност

std::unique_ptr<T> понеже е доста по - лесно поради факта, че не нужно собственоръчно да се грижиш за паметта а, тя автоматично се трие след това.Също така при този вид пойнтър няма как да се направи нова референция и за опростение на дивелопъра operator= и копи-конструкторът са изтрити.

В Invoker класа единственото нещо, което се случва е още при създаването на обект от този вид той да запълва вектора от смарт пойнтъри с командите като ги прави смарт пойнтър и при написването на дадената команда тя да се оцени и да се приложи дадения метод , на когото принадлежи командата!

```
std::vector<std::unique_ptr<ICommand>> commands;
```

```
class Invoker {
private:
    vector<unique_ptr<ICommand>> commands;
    JsonManager jsonManager;
    void parseLine(string&);
    void applyArguments(const string& commandName, vector<string> arguments);
public:
    Invoker();
    void run();
};
```

```
Invoker::Invoker() {
    commands.push_back(make_unique<OpenFileCommand>());
    commands.push_back(make_unique<ValidateCommand>());
    commands.push_back(make_unique<SearchCommand>());
    commands.push_back(make_unique<PrintCommand>());
    commands.push_back(make_unique<HelpCommand>());
    commands.push_back(make_unique<CloseCommand>());
    commands.push_back(make_unique<SaveCommand>());
    commands.push_back(make_unique<SaveAsCommand>());
    commands.push_back(make_unique<ExitCommand>());
    commands.push_back(make_unique<SetCommand>());
    commands.push_back(make_unique<DeleteCommand>());
    commands.push_back(make_unique<CreateCommand>());
}
void Invoker::run() {
    string line;
    for(;;){
        cout << "$: ";
        getline(cin, line);
        parseLine(line);
    }
}
void Invoker::parseLine(string& line) {
    stringstream stream;
    stream << line;
    string commandName;
    stream >> commandName;
    vector<string> arguments;
    string argument;
    while(stream >> argument) {
        arguments.push_back(argument);
    }
    applyArguments(commandName, arguments);
}
void Invoker::applyArguments(const string &commandName, vector<string> arguments) {
    int counterForCommands = 0;
    for(auto& command: commands) {
        if(command->convertToString() == commandName) {
            command->setArguments(arguments);
            command->execute(jsonManager);
            counterForCommands++;
        }
    }
    if(counterForCommands == 0){
        cout << "WRONG COMMAND\n";
```

4. Реализация и тестване :

4.1. Реализация на класове:

4.1.1. По горе на картинките са показани част от по важните класове , които са задължителен слой от архитекртурата

5. Заключение:

5.1.Крайният продукт е работещо приложение, което отговаря на ООП парадигмите и желанията на клиента.

5.2. Оптимизацията на проекта може да се осъществи като се добавят нови бази от данни или се разработят нови команди, които подобряват базата от данни.Могат също така да се добавят нови функционалности и поради факта, че кодът е написан за бъдещо разширение на този продукт, няма опасност от спиране на извървашнето на старите такива.

6. Използвана Литература:

6.1.https://www.youtube.com/

6.2.http://www.cplusplus.com/reference/

6.3. https://www.wikipedia.org/

6.4.https://www.geeksforgeeks.org/

6.5. https://stackoverflow.com/