Курсова работа по ООП – проект 13, XML парсер

Павел Григоров, СУ-ФМИ, КН, ФН 82038

Връзка към проекта - <https://github.com/PKGri/OOP_Course_Project_XML_Parser>

1. Въведение

XML парсерът е програма, която дава възможността да се работи с данни записани във файловия формат XML (e**X**tensible **M**arkup **L**anguage - разширяем маркиращ език). В действителност тези файлове представляват текстови файлове, в който данните са записани по определена конвенция. Основната функционалност е прочитането на файла и съответно представянето им в паметта на компютъра по начин, който позволява да бъдат извършвани множество действия, включващи извличане на конкретна информация, редактиране на някои от данните, записване на данните отново в XML файлов формат, както и осъществяването на някои основни XPath заявки.

XPath заявките са изрази, които указват на програмата каква информация да извлече и да представи на потребителя.

Първата основна задача е прочитането на един XML файл. За тази цел трябва да се измисли как трябва да бъде представяна в паметта текстовата информация от файла, за да могат след това да бъдат извършват операции с нея.

Следващата задача е да се определи какви точно операции да бъдат поддържани и по какъв начин да се подходи с тяхното осъществяване в кода. От тези операции, може би най-сложната е извличането на конкретна информация чрез XPath заявки, затова като последна стъпка в разработката оставям точно тази функционалност.

Междинни цели са осъществяването на работа с команден ред, за улеснена работа с всичките заредени данни, както и придаването на атрибут с ключ “*id”* и с уникална стойност съответстваща на всеки елемент, за да могат някои операции да бъдат извършвани върху точно определен елемент.

По-долу тази документация описвам как съм подходил в достигането си на тези цели, някои от проблемите, които са се явили в процеса на разработката и как съм се справил с тях. Ще бъдат описани всичките класове и повечето от по-сложните функции, които са част от кода. Разбира се, на края на този документ се намират хипервръзки към отделните източници, които ми бяха от полза в изготвянето на проекта.

1. Преглед на предметната област

Тук ще представя и дам дефиниции на специфичните термини, които ще бъдат срещани надолу в документацията.

**Общи:**

* **Регулярен израз –** последователност от знаци, която дефинира шаблон за търсене. В кода е използвана стандартната библиотека <regex>.

**XML специфични:**

* ***“Таг“* –** всяко нещо във файла, което се съдържа между скоби < >
* **Отварящ таг –** таг, чрез който се маркира началото на нов елемент - < >
* **Затварящ таг –** таг, чрез който се маркира краят на елемент - </ >
* **Самозатварящ се таг** или  **празен таг–** таг, който отговаря на празен елемент - < />
* **Таг на версията –** таг в началото на файла, който може да бъде пропускан; Този таг указва каква версия на XML е използвана за файла.
* **Ключ –** името на елемент или атрибут. Намира се в началото на отварящ таг.
* **Елемент –** елементът включва цялата информация от начален таг до съответния затварящ таг
* **Коренов елемент** или **корен –** първоначалният елемент, отворен чрез първия таг след таг на версията, ако има такъв. Файлът може да съдържа само един коренов елемент.
* **Елемент-дете** или **дете –** елемент, който е вложен в друг елемент
* **Родителски елемент** или **родител –** елемент, който има поне едно дете
* **Текстов елемент –** елемент, който няма деца, но между отварящия и затварящия си таг има текстово съдържание.
* **Празен елемент –** елемент, който няма нито деца, нито текстово съдържание
* **Атрибут на елемент –** атрибутът съдържа информация относно специфичен елемент. Атрибутите са представени в отварящия таг на елемента им, след неговия ключ. Всеки елемент може да притежава множество от атрибути.
* **Стойност на атрибут –** информацията, която даденият атрибут пази за съответния си елемент.
* **Идентификатор** или “***id”,***или “***ID”* –** атрибут, който по условието на проекта се придава на всеки елемент и има уникална стойност.

**XPath специфични:**

* **Път –** указаният чрез *XPath-*израз път до елемент, наподобяващ път на директория в компютъра.
* **Възел –** всяка част от *XPath-*израза, която отговаря на елемент или атрибут
* **Предикат –** условие, което даден възел-елемент трябва да изпълнява, за да се продължи към края на израза. Намира се в скоби […] след ключа на елемента.

За главната цел, която е четенето на XML файла, първото, което направих, беше да създам клас за елемент. За по-лесна работа с атрибутите на елемента съответно създадох и клас за атрибут. След това създадох клас за самия файл, така че да са по-добре структурирани прочетените данни.

Написах логиката си за работа с команден ред, за да мога по-лесно да изпробвам различните операции, които трябва да извършвам, след което почнах да реализирам изискваните команди.

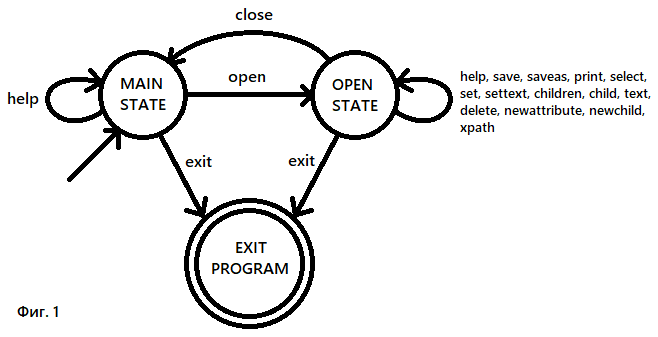
1. Проектиране и реализация

В програмата са създадени три класа, които да ми отговарят за различни данни:

* class Attribute
* class xmlElement
* class xmlFile

Освен тях има клас *XPathNode*, който се използва при Xpath заявките.

Отделно от тях съм направил функции за работата с команден ред.

Първо ще опиша как реализирах **командния ред**. Като цяло той е базиран на цикъл *do{…}while(…)* с още един такъв цикъл вложен в него*.* Тъй като операцията *open* за четене на XML файл не може да се извърши, ако вече има отворен файл, то може да разглеждаме действието на програмата като детерминиран краен автомат с 3 състояния и преходи – отделните команди възможни в тези състояния. (фиг. 1)

void enterCommand(std::function<bool(std::string)> terminationPredicate,

std::function<bool(std::string)> commandList)

{

do

{

do

{

std::cout << '>';

std::cin >> command;

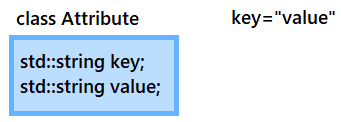
} while (!commandList(command));

executeCommand(command);

} while (!terminationPredicate(command));

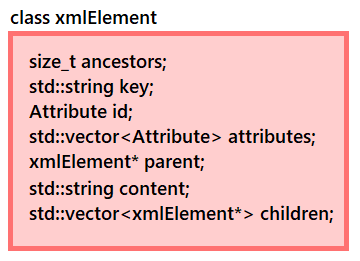
} **Фиг. 2**

За повече четливост и за да се избегнат повторения на код, използвах функции от по-висок ред. (фиг. 2)

Класът ***Attribute*** е много семпъл, съдържа по един низ за ключа и стойността на атрибута. (фиг. 3)

Форматът по който се задава атрибут вътре в таг е *key=”value”.*

Фиг. 3



Класът ***xmlElement*** е малко по-сложен.

Всяка инстанция съдържа член-данни за своите:

* брой предшественици
* низ за ключа
* идентификатор
* вектор от атрибути
* указател към своя родител (ако има)
* низ за текстовото си съдържание
* вектор от указатели към децата си - също инстанции на този клас

Фиг. 4

Функцията, която прочита от файла и създава атрибутите на един елемент е *createAttributes*, която използва регулярни изрази за разпознаването на ключовете и стойностите на атрибути от отварящия таг на еле

void xmlElement::createAttributes(const std::string& tag)

{

std::regex keyRegex("[^ \n\t&<>]+[ \n\t]\*=");

std::regex valueRegex("\"[^&<>\"]+\"");

std::sregex\_iterator currentKeyMatch(tag.begin(), tag.end(), keyRegex);

std::sregex\_iterator currentValueMatch(tag.begin(), tag.end(), valueRegex);

std::sregex\_iterator lastMatch;

while (currentKeyMatch != lastMatch && currentValueMatch != lastMatch)

{

Attribute toAdd;

std::smatch kmatch = \*currentKeyMatch;

std::smatch vmatch = \*currentValueMatch;

std::string key(kmatch.str());

std::string value(vmatch.str());

key.pop\_back();

std::stringstream ss(key);

ss >> key;

value.erase(value.begin());

value.pop\_back();

toAdd.setKey(key);

toAdd.setValue(value);

if (key == "id") setID(value);

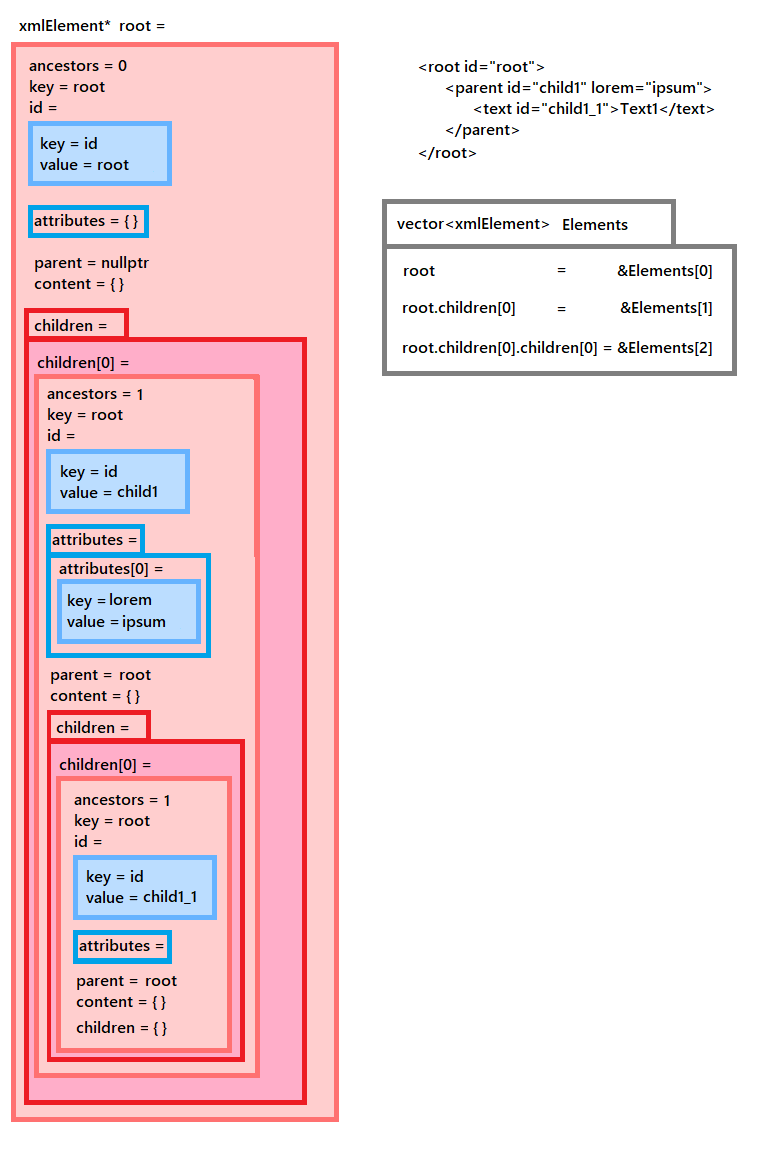
else addAttribute(toAdd);

currentKeyMatch++;

currentValueMatch++;

}

} **Фиг. 5**

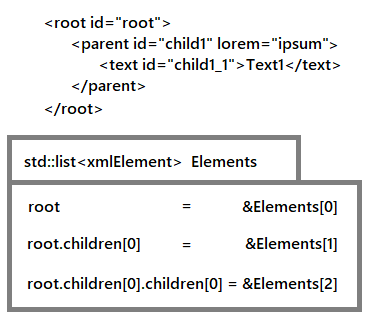
Обект от класа ***xmlFile***съдържа цялата заредена информация от един файл. Член-данните му са:

Фиг. 6

Фиг. 7

* низ за името/директорията на файла
* низ за XML версията на файла
* списък от елементите, инстанции на класа xmlElement
* указател към кореновия елемент, който е първият елемент в списъка

Първоначално вместо списък, съдържащ елементите, имаше вектор, но когато правех функцията за добавяне на ново дете към елемент видях, че ще се наложи списък, тъй като исках в контейнера всички елементи да бъдат подредени по реда им на четене/записване, а във вектор няма лесен начин за добавяне на елемент на позиция, различна от последната.

Както предполага името на проекта, може би най-трудната част беше извличането на информацията от файл. Множество функции се ползват за тази цел – за четене на таг, на текстово съдържание, създаване на елемент. Има много проверки за валидността на потока, някои от тях позволяват дори грешно форматиране да бъде прочетено и записано правилно. На края на функцията *open*, която е водещата за четенето, отвореният файл се затваря и информацията остава заредена в паметта на компютъра. При изпълняване на командата *close* се освобождава паметта от заредената информация чрез функцията *clear*. На фиг.6 и фиг. 7 е изобразено как примерно съдържание се записва въз основа създадените елементи и техните взаимовръзки.

Голямо препятствие се оказа функцията *uniquifyID* за придаване на уникални идентификатори на всеки елемент. Не съм много доволен от реализацията ѝ и в бъдеще ще я преразгледам. Проблемът ѝ е, че минава през всички елементи с цикъл, като вътре в този цикъл отново минава през всички елементи, намирайки такива с еднакви стойности на идентификаторите, после минава с трети цикъл през намерените еднакви идентификатори, като конкатенира всеки с различно число.

1. XPath

Считам, че тази функционалност трябва да бъде отделна глава, защото съм си позволил своеволие в интерпретацията на условието на проекта относно нея.

Тъй като в условието примерите, които са дадени за XPath изрази, не са истински такива, ами подобие, което е направено да прилича малко на XPath (предполагам за улесняване работата на студентите), аз реших да ме осъществявам по указания начин функцията *xpath*, а да се опитам да я реализирам по начин, по който наистина наподобява коректни XPath заявки. Затова тук ще опиша по-подробно работата с тази функция. Всичко, което ще описвам ще в отношение към ето това примерно XML съдържание:

<parent>

<child at1=”attribute”>

<text>Text1</text>

<text>Text2</text>

</child>

<child>Text3</child>

</parent>

Синтаксисът със символ ‘/’ показва абсолютен път към даден елемент, подобно на директория в компютъра. Така ако искаме достъп до текстовото съдържание на всички елементи с ключ *text*, които са деца на *child,* то трябва да се напише израз ***/parent/child/text* .** Тъй като ‘/’ показва абсолютен път, не бива да се пропуска нито един елемент-възел по пътя.

**Пример:** ***/child/text*** няма да изведе нищо

Синтаксисът за предикат се указва със скоби [ ] след ключа на елемент-възел.

**Пример: */parent/child[1]/text[2]*** ще даде второто дете *text* на първото дете *child* на елемента *parent*

В минималните изисквания са указани три различни вида предикати. С правилен XPath синтаксис тези предикати изглеждат така:

1. ***/parent/child[1]/text***
2. ***/parent/child[text=Text2]/text***
3. ***/parent/child[@at1]/text***

Първият предикат прави проверка в родителския елемент на текущия възел – търси n-то по ред дете с ключ в този случай ***child*.**

Вторият предикат прави проверка в децата на текущия възел – търси дали има дете с ключ ***text***, чието текстово съдържание е ***Text2.***

Третият предикат е най-семпъл, защото проверява единственo в текущия възел дали той съдържа атрибут с ключ ***at1*.** Всеки от тези изрази ще дава един и същ резултат.

Аз съм прибавил и четвърти предикат, който е да провери дали даденият елемент съдържа дете с опренелен ключ.

**Пример: */parent/child[text]/text***

В програмата ако потребителят иска да направи XPath заявка, преди нея трябва да напише командата ***xpath***.

Сега след като съм обяснил синтаксиса е време да опиша и моята реализацията на тази функционалност.

Тук се появи нуждата от класа ***XPathNode***, който представлява два низа – един за елемента нз възела и един за предиката, ако има такъв. В него има функция ***parseNode***, която взима един низ, който представлява елемент и предикат, като предикат може да няма, отделя елемента от предиката чрез регулярни изрази и на текущия обект му променя двата съответни низа да бъдат с тези стойности.

В класа ***xmlFile*** създадох функцията ***parseXPath****,* която взима въведения XPath израз и чрез регулярни изрази отделя всички възли-елементи и съответните им предикати като с тях създава обекти ***XPathNode***, които вкарва във вектор по реда на срещане и връща този вектор.

Този вектор се изпозлва в основната функция ***xpath***, като с него, нулевата му позиция и кореновият елемент като аргуемнти се вика рекурсивната функция ***traverseXpath.***

В тази функция се проверява дали ключът на подадения елемент съвпада с текущия възел и дали изпълнява предиката му. Ако да, започва да гледа следващия възел и минава през всички деца на досегашния елемент. Ако вече новият възел е последният, извежда информацията на децата, които съвпадат по ключ с него и изпълняват съответния предикат. Ако пък възелът не е последен, вика функцията рекурсивно чрез новия възел и текущото дете.

1. Примери и тестови сценарии

В GitHub репозиторията на проекта съм прикачил и папка с разни XML файлове, които съм ползвал за тестване на програмата си.

Връзка към репозиторията - <https://github.com/PKGri/OOP_Course_Project_XML_Parser>

1. Заключение

В изпълнението си на проекта съм предоставил начин за осъществяване на всяка от исканите операции, като съм добавил и малък брой допълнителни. XPath поддръжката ми не е такава, каквато е описана в условието, но вярвам, че изпълнението на същинската задача има повече предимства от придържането към улеснените условия, давани за студенти.

В бъдеще ще преразгледам функциите, които са по-тежки за изпълнение, дали мога да ги оптимизирам повече и като цяло ще искам да разгледам какво мога да оптимизирам по проекта. Xpath поддръжката има още много, което може да се допълва по нея, това е най-минималното; Има множество други предикати и оператори с разнообразни действия и ползи.