Project Software Engineering: XML parser tutorial

Kasper Engelen
kasper.engelen@uantwerpen.be

XML (Extensible Markup Language) is een taal waarmee we op een gestructureerde manier gegevens kunnen weergeven in een tekstuele vorm. Deze representatie is dus eenvoudig leesbaar voor de mens en, door middel van een XML-parser, eveneens leesbaar voor een machine.

In dit practicum zullen we leren werken met de "TinyXML" library. TinyXML is een parser tool waarmee je XML bestanden kunt inlezen en er vervolgens informatie kunt uit halen. Het doel van dit tutorial is dat je leert werken met deze tool, zodat je deze kan gebruiken in het project.

TinyXML Tutorial

XML is een relatief eenvoudig data formaat. Het is leesbaar voor de mens en zeer gelijkaardig aan HTML. Hieronder een algemeen voorbeeld van een correcte XML file:

```
<?xml version="1.0" ?>
<root>
     <Element1 attribute1="some value" />
     <Element2 attribute2="2" attribute3="3">
         <Element3 attribute4="4" />
         Some text.
     </Element2>
</root>
```

Download de XML Parser hier: `http://sourceforge.net/projects/tinyxml/`
Plaats deze bestanden in je project:

```
tinystr.cpp
tinyxmlerror.cpp
tinystr.h
tinyxml.h
tinyxml.cpp
tinyxmlparser.cpp
```

Vergeet niet de `cpp`-bestanden ook in de `CMakeLists.txt` toe te voegen!

Om de TinyXML library te gebruiken moeten we `tinyxml.h` includen in onze code:

```
#include "tinyxml.h".
```

Nu kunnen we een document laden. Maak een bestand aan met het XML voorbeeld van hierboven en noem het “test.xml”. We kunnen dit bestand nu inladen met:

```
TiXmlDocument doc;
if(!doc.LoadFile("test.xml")) {
    std::cerr << doc.ErrorDesc() << std::endl;
    return 1;
}
```

Let op: CLion runt standaard de applicatie vanuit de map `cmake-build-debug`, en zal dus ook daar naar bestanden zoeken. Hierdoor zal het `test.xml` bestand niet

gevonden worden. Je kan via *edit configurations…* de *working directory* aanpassen om dit op te lossen.

De variabele `doc` bevat nu alle data. We kunnen de data eruit halen als volgt:

```
TiXmlElement* root = doc.FirstChildElement();
if(root == NULL) {
    std::cerr << "Failed to load file: No root element." << std::endl;
    doc.Clear();
    return 1;
}
```

We hebben nu een variabele die het `root` element bevat. De rest van de data kunnen we aanspreken via dit element op dezelfde manier als in de vorige code: de `FirstChildElement()` methode geeft een `TiXmlElement` pointer terug die wijst naar de eerste *childnode*. Elke klasse die afgeleid is van `TiXmlNode` (waaronder `TiXmlDocument` en `TiXmlElement`) bevat deze methode.

De `FirstChildElement()` methode heeft als optioneel argument een string met de naam van het element dat je zoekt. Deze hadden we niet nodig omdat we wisten dat er maar één `root` was. We weten nu dat onze `root` twee kinderen heeft met een specifieke naam. We zouden dus de `FirstChildElement()` methode kunnen gebruiken met de naam van elk element. Maar, omdat het aantal nodes en hun naam niet altijd gekend is, zullen we algemenere code gebruiken:

```
for(TiXmlElement* elem = root->FirstChildElement(); elem != NULL;
                                        elem = elem->NextSiblingElement()) {
        string elemName = elem->Value();
        ...
}
```

Deze code loopt over de elementen die direct kind zijn van `root`. We specificeren geen naam in de calls naar `FirstChildElement()` en `NextSiblingElement()`. Daarom moeten we controleren wat de naam is. De `Value()` methode is verschillend voor elke klasse die afgeleid is van `TiXmlNode`: voor `TiXmlElement` geeft deze de naam van het element terug.

```
const char* attr;
if(elemName == "Element1") {
        attr = elem->Attribute("attribute1");
        if(attr != NULL)
                ; // Do stuff with it
}
```

De variabele `attr` wordt gebruikt om de opgevraagde attributen bij te houden. Als het opgevraagde attribuut niet bestaat zal de `Attribute()` methode `NULL` terug geven.

We kunnen de attributen van Element2 op dezelfde manier opvragen:

```
else if(elemName == "Element2") {
        attr = elem->Attribute("attribute2");
        if(attr != NULL)
                ; // Do stuff with it
        attr = elem->Attribute("attribute3");
        if(attr != NULL)
                ; // Do stuff with it
}
```

`Element2` bevat ook een kind: `Element3`. We zullen `Element3` zoeken met een loop om te laten zien hoe een meer specifieke loop gebruikt kan worden. Deze loop zal alle elementen skippen die niet '`Element3`' als naam hebben.

```
for(TiXmlElement* e = elem->FirstChildElement("Element3"); e != NULL;
                                e = e->NextSiblingElement("Element3")){
        attr = e->Attribute("attribute4");
        if(attr != NULL)
                ; // Do stuff with it
}
```

Attributen zijn niet de enige manier om data bij te houden in XML. Een andere veelgebruikte manier is tekst dat binnen in een element staat, zoals bijvoorbeeld in `Element2`. Deze tekst wordt door TinyXML bij gehouden in een text node (`TiXmlText`). Omdat er geen `FirstChildText()` methode is gebruiken we de `ToText()` methode om de `TiXmlNode` te casten naar een `TiXmlText` klasse. Als deze `NULL` is, is het geen tekst node. We kunnen de tekst zelf verkrijgen met de `Value()` methode van `TiXmlText`.

```
        for(TiXmlNode* e = elem->FirstChild(); e != NULL; e = e->NextSibling()){
                TiXmlText* text = e->ToText();
                if(text == NULL)
                        continue;
                string t = text->Value();
                // Do stuff
        }
```

Daarmee is heel ons voorbeeld XML bestand geparsed. We kunnen nu het geheugen dat TinyXML gebruikt nu vrijgeven als volgt:

```
doc.Clear();
```

Oefening 1: Lezen en uitprinten van data

In het bestand `eenCD.xml` staat een XML-beschrijving van een CD. Schrijf een functie die het `eenCD.xml` bestand parst. Vervolgens moet je in dit component de ingelezen Artist en Title opvragen en uitprinten in de console output.

Oefening 2: Meer OO Gericht

In de vorige oefening heb je een functie moeten maken die een XML bestand inleest. In deze oefening zal je deze code herstructureren en gebruikmaken van klasses.

(a) Maak een nieuwe klasse aan waarin we een CD definiëren. CD objecten houden elk de Artist, Title, Year en Price van de CD bij. Voorzie getter en setter procedures om de data aan te passen en houd rekening met encapsulatie.

(b) Maak nu een functie waarin er een CD object aangemaakt wordt met de informatie die werd geparst. Let op: het parsen van de XML file mag niet in de constructor van de CD klasse gebeuren!

(c) Voorzie een `print` methode waarmee de Artist, Title, Price en Year uitgeprint kan worden.

Opmerking: Je zal voor deze oefening strings (type `char*`) moeten omzetten naar floats `float` en integers `int`. Op internet kan je vinden hoe je dit kan doen met de C++ standard library.

Oefening 3: Van XML naar ADT

In deze oefening zullen we de vorige oefening uitbreiden zodat we nu een lijst van CD's kunnen inlezen. In het bestand `cdCatalog.xml` staat een lijst van CD's.

(a) Maak een Catalog klasse die een lijst van CD's bijhoudt. Hou rekening met encapsulatie en voorzie de nodige methodes. Er mag geen getter methode zijn die de lijst van CD's in zijn geheel teruggeeft!

(b) De parser functie moet nu worden aangepast zodat deze elke CD in de lijst inleest en hiervoor een CD object aanmaakt. Dit CD object moet dan worden opgeslagen in het Catalog object.

Opmerking: The informatie over een specifieke CD staat nu een nesting level dieper. Hou hiermee rekening wanneer je informatie opvraagt met TinyXML.

Oefening 4: Fouten in XML bestanden

In de bestanden `cdCatalogError.xml` en `eenCDError.xml` vinden jullie XML-bestanden met fouten in. Je systeem dient hierop gepast te reageren door de geschikte errors terug te geven.

Het bestand `eenCDError.xml` bevat een fout in de XML syntax. TinyXML zal zelf een error teruggeven (en stoppen met parsen). Deze dient uiteraard in de console geprint te worden.

Het bestand `cdCatalogError.xml` bevat enkele semantische fouten. Je systeem dient deze fouten te detecteren en zo gedetailleerd mogelijk errors terug te geven.