# Dr. Evil

I used to use a PC, but it was designed by a freakin' idiot.

Now I use a Mac, allowing me to control the "lasers" on my "death star" with ease.

I'm Dr. Evil, and I'm aspiring to take over the world.

# OOP: I Did It Again

Lessons Learned from Parsing ANSI X12 271
and Tips on Object-Oriented Programming

by Sam Livingston-Gray

# About The Speaker

- I started working with Access exactly six years ago. After ~3 weeks, I realized I needed to learn VBA to do the things I really wanted to do...
- and thus a developer was born.
- This presentation grew out of my last full-time job, working with Jerry Porter (another PAUG member).
- I'm now pursuing a BSCS, starting at PCC.

# About This Talk

- In 2001, I was asked to write some code to parse and import text data in ANSI X12 271 format. *(Avoid this format like the plague!)*
- I'll walk you through the various stages of this project, relating the problem to object-oriented design at key points as we go.
- There will be no demo -- the running code is too complicated for a 1-hour talk, and doesn't have a UI.
- We'll pause for questions between sections.

# Agenda

- We'll start with *hierarchical data* and *OOP*.
- I'll talk about one of the better hierarchical formats (XML) and contrast it with one of the worst (X12).
- We'll look at my first attempt to parse X12 data, which included one fairly simple object.
- We'll look at version two, which used a few more objects, and compare the versions.
- If there's time, maybe we can get to some abstract generalizations and hand-waving...

# Fundamentals: Hierarchical Data

- Hierarchical databases apply meaning to data not just by what's in the fields, but also by where that data is stored.
- Probably the best-known example: the filesystem. C:\My Documents\PAUG\Presentations\X12.ppt
- This talk deals with several different hierarchical structures (XML, X12, and some nested objects). **These all have the same logical structure as your filesystem.**

# Fundamentals: Classes and Objects

- Let's take suburbia as an example. A class module is like the blueprint for all houses in a subdivision (e.g., the House class).
- An object is like one house on a street.
  - It's built using the class module as a plan:
    Dim myHouse as House
    Set myHouse = New House
  - Each object can have its own properties:
    myHouse.PaintColor = vbGreen
    myHouse.RoofType = vbComposite

# Questions?

- That's it for the intro of hierarchical data and OOP.
- Next up:  XML (eXtensible Markup Language)
- (Show of hands: who's worked with XML? HTML?  If not, come early next time!)

# XML:
# A Hierarchical Text Format

- Basic unit of an XML file: the tag. This corresponds to a folder in the filesystem example.
- Special syntax defines the beginning and ending of a tag (confusingly, these are also called the start tag and end tag). Start and end tags are surrounded by angle brackets (<>), and end tags start with a slash (/). ex: <**tag**>some data</**tag**>
- Anything between the start and end tags is part of that tag.

# XML Syntax

## Some other rules

- Tags can be nested (nested tags are also called child tags).

  <**foo**>some data
  <**bar**>more data</**bar**> ← *child tag*
  </**foo**>

- Tags can have attributes as well as child data.

  <**foo id="root"**>data</**foo**>
  ↑attribute

- Tags with no data are indicated as: <**tag** />
- Two simple examples follow...

# XML Example

```
<library>
    <author>
        <name>Sam Livingston-Gray</name>
        <books>
            <book>
                <title>Great American Novel</title>
            </book>
            <book>
                <title>Horrible American Novel</title>
            </book>
        <books>
    </author>
</library>
```

# XML example using attributes

```
<library>
    <author name="Sam Livingston-Gray">
        <books>
            <book title="Great American Novel" />
            <book title= "Horrible American Novel" />
        </books>
    </author>
</library>
```

# XML: Easy To Parse

- When parsing XML, a tag's level is always known.
- If one start tag is encountered after another start tag, the second tag is ***always*** a descendant of the first...
- ...because the first one's end tag hasn't been encountered yet. Anything after that end tag is ***not*** a descendant of the original tag.
- The file can thus be read properly ***without any external information about its structure.***

# Parsing XML: SAX & DOM

- There are two main schools of XML parsing.
  - SAX (Simple API for XML): parser makes one forward-only pass through the document, firing events as it reaches new tags. Pros: fast and small. Cons: code must remember every element or it's lost forever.
  - DOM (Document Object Model): The entire document is read into memory and can then be accessed through a hierarchy of objects. Pros: allows arbitrary access to data. Cons: big and slow.

# Questions?

- That's it for the XML intro.
- Next up:  ANSI X12

# Comfortable?  Let's Fix That

- Now, imagine a file format without end tags.

```
<author>Sam Livingston-Gray
    <books>
        <book>
            <title>Great American Novel
        <book>
            <title>Horrible American Novel
```

- I've indented the above so you can see the relationships.  Let's see it as an X12-style file...

# X12-Style Data

AUTHOR*42*Sam Livingston-Gray@
BOOKS*42@
BOOK*42*1@
TITLE*Great American Novel@
BOOK*42*2@
TITLE*Horrible American Novel@

- Note the delimiters:
  - @ marks a ***segment*** (⇔ start tag).
  - * marks an ***element*** (⇔ attribute).
  - I call the zero$^{th}$ element the “base element.”

# So What?

- Here's the problem:  When you read a new segment, how do you know where it is in the hierarchy?
- Answer:  Look it up in the (400-page!) spec.
- As opposed to XML, which can be interpreted on its own, interpreting X12 requires extra information.
- What this means is that you have to encode information about how to handle different segments into your parsing logic (a la SAX).

# Questions?

- That's it for the X12 intro.
- Next up: Version 1

# Spec This (1)

- (from previous slide) “What this means is that you have to encode information about how to handle different segments into your parsing logic (a la SAX).”
- In fact, that’s what the first version did:  the structure of the code reflected the structure of the document.  Every segment corresponded to a Do..Loop structure with a Select Case statement inside it.
- This went 14 layers deep (ick)!

# Version 1: Never Do This!

```
Do
    theParser.ReadNextSegment  // we'll get to this...
    Select Case theParser.Element(0)
        Case "AUTHOR"
            [Process AUTHOR segment]
        Case "BOOKS"
            Do    // Note nested structure!
                theParser.ReadNextSegment
                Select Case theParser.Element(0)
                    [...] // Repeat 14 layers deep
            Loop
Loop
```

# Version 1 Pros/Cons

- On the upside:
  - This was relatively quick to write -- I had a working version in about 3 days.
  - It also ran fairly fast: 4,000 records/minute, where a record consisted of 8-50 segments.
- However:
  - This was a nightmare to write, let alone maintain. The structure is highly redundant, which means lots of copying/pasting code.
  - ***Any*** change to the spec meant writing and delivering new code.

# OOP Interlude: X12 Parser

- As part of this, I wrote a forward-only parser object that reads one segment at a time. (This was intentionally modeled after SAX.)
- Key methods/properties:
  - OpenInputFile: Tells parser where to find data.
  - ReadNextSegment: Reads to the next @ delimiter and splits that string on the * delimiter for inclusion in the Element array.
  - Element(index): Exposes individual elements.

# Parser In Action

```
Do
    theParser.ReadNextSegment
        Select Case theParser.Element(0)
            Case "AUTHOR"
                [Process AUTHOR segment]
            Case "BOOKS"
                Do     // Note nested structure!
                    theParser.ReadNextSegment
                    Select Case theParser.Element(0)
                        [...] // Repeat 14 layers deep
                Loop
Loop
```

# Questions?

- That's it for Version 1.
- Next up: Version 2

# Take Two: The Map

- Key realization: Because X12 and XML are both hierarchical, you can use XML (the good one) to represent the structure of X12 (the bad one)!
- By keeping a "map" of the structure as an XML document -- and keeping track of where you are in the map -- you can figure out how to interpret new X12 segments.
- Advantage: By changing the map, you can change how the program behaves without rewriting code.

# XML Map Example

```
<x12map>
    <segment BaseElement="AUTHOR">
        <segment BaseElement="NAME">
            <field element="1" target="AUTHORNAME" />
        </segment>
        <segment BaseElement="BOOKS">
            <segment BaseElement="BOOK">
                <segment BaseElement="TITLE" />
            </segment>
        </segment>
    </segment>
</x12map>
...and so on.
```

# Version 2A

- Load the map from the table into a string
- Use the XML DOM parser to turn the map string into a hierarchical set of objects
- Tell the X12 parser to read through its data. At each new segment, find the object in the XML DOM hierarchy that represents our place in the map. This will tell us how to interpret the tag.

# Pseudocode

Do Until theParser.EOF

theParser.ReadNextSegment

Starting from the map's current context, find the tag in XML map where BaseElement tag matches theParser.Element(0)
***(more on this on the next two slides)***

Query the map to find out what to do with this segment (which fields to read, etc.)

Loop

# O Brother, Where Art Thou?

- It turns out that in X12, when you reach a new segment, it can have one of several relationships to the previous segment.  It can be:
  - The same segment, repeated
  - A sibling
  - A child
  - A parent or ancestor
  - The sibling of an ancestor
- (This is why I prefer XML: you always know how to treat a new tag.)

# Using The Map: You Are Here

- After much discussion and debate, we settled on a simple algorithm that will always find the next segment in the map (if the map is complete):
  - Examine current map location: are we already at the next segment? If so, we're done.
  - Examine all children: is one of them the next segment? If so, we're done.
  - If neither of the above is true, move to self's parent and repeat.
  - If we're at root and no match, map is wrong!

# So What?

- So now we have a way to build a ***map*** (using an XML document) that represents the structure of the X12 data.
- ...which means we can throw away the ***code*** that did the same job of representing structure...
- ...leaving only code for navigating the map...
- ...which means that when the X12 data changes, we just publish a new map -- which won't need nearly as much debugging!

# Version 2A Pros/Cons

- Pros: ***much*** more flexible. Changes to the layout no longer require us to edit a single line of VBA.
- Cons:
  - DOM code is very verbose:
    - 5-10 lines to change context (which may be done 5-6 times before we find the right one)
    - 5-10 lines to see if the current context matches the next segment.
  - Total speed is 4x slower than version 1!

# Questions?

- That's it for version 2A.
- Coming up: version 2B

# Revelation 2

- I'd been using XML DOM because I happened to know it, and because I wanted to do my map editing in XMLSpy (a good, though expensive, XML editor).
- But the performance penalty turned out to be too high, because DOM is a general-purpose tool with lots of powerful features.
- ...so could I write a custom tool that did the same job with less overhead?
- ...like, say, my own hierarchy of objects?

# The Segment Object

- Key properties:
  - BaseElement: Tells us what we expect in the zeroth element. (We check this to see if we're in the right map context while searching.)
  - ChildSegments: Collection of other Segment objects "below" this one in the hierarchy.
  - ParentSegment: Refers to the Segment that has this one in its ChildSegments collection.

# Version 2B

(Changes from 2A highlighted)

- Load the map from the table into a string
- Use the XML DOM parser to turn the map string into a hierarchical set of objects
- ***Turn the XML DOM objects into a set of Segment objects***
- Tell the X12 parser to read through its data. At each new segment, find the object ***in the Segment hierarchy*** that represents our place in the map. This will tell us how to interpret the tag.

# Version 2B Pros/Cons

- Has all the advantages of version 2A in terms of flexibility and speed of development,
- Map navigation code uses 5 times fewer lines,
- And it's twice as fast as 2A right off the bat. After optimization, it was ~2.2 times faster than the same code that used the XML DOM.
- ...This is still 1.8 times slower than the original Select Case set, but the advantages for us made this an acceptable tradeoff.

# Questions?

- That's (almost) it for version 2.
- Coming up (if there's time): memory leaks, metaprogramming, reference counting, and other gibberish

# Next...?

- We discussed using a 'metaprogramming' hack that would allow us the development speed of the map code, coupled with the execution speed of the Select Case structure.
- How? Simple! Use the map code to write the Select Case code!
- ...we didn't bother (yet).

# Remember Sammy Jankis

- Version 2B, however, had a "gotcha" in it: a memory leak. Every time you ran the code, the map hung around in memory until Access closed.

  (If we'd been using VB6, the memory would've been consumed until system shutdown!)
- This was because my Segment structure contained circular references.
- I'll cover reference counting if there's time. (If not, just remember: memory leaks are easy to create if you don't know what you're doing.)

# The End

- Thanks!
- Direct questions to slg@timestream.net
- Looking for a consultant?
  http://timestream.net/resume