Leveraging the Capabilities of the Blaise Editor Add-In Tool to Improve the Readability of Datamodel Source Code

Chris Oz,
Survey Research Center, University of Michigan

1 Introduction

One of the most pervasive issues that have plagued mankind's communication since the creation of the first written language has been one of readability. Regardless of the language used, books and newspapers need to be written in such a way that the information being conveyed is clear, concise and scannable—clear, so the reader understands what is being expressed; concise, so the thoughts and opinions expressed are not so overly wordy that they are incomprehensible; scannable, so the reader can efficiently read the text, with the minimum amount of re-reading possible. While a writer is not required to use proper grammar, punctuation and sentence structure to express thoughts and opinions, these basic rules of communication prevent the reader from becoming lost, confused, or even disinterested in the subject matter. This is the true litmus test for what makes a book well written, and software follows much of the same tenets. The world of computer code and software development is no different; the code that authors write is a list of instructions in a specific language that the computer is expected to carry out.

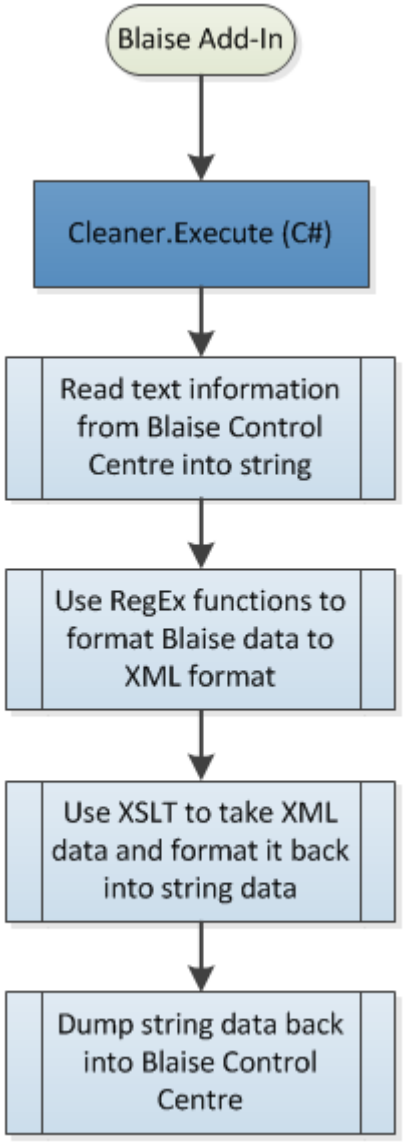
A software developer can write computer code that is for all intents and purposes correct, yet write it poorly enough that another author, unfamiliar with the code, would have a hard time following what is going on. This is a problem, because not only does it take longer to read and comprehend poorly written code, it also takes longer for someone to modify it. The Blaise Control Centre is a powerful and diverse tool for survey authors to create complex and extensible surveys; however, this flexibility allows for authors to write syntactically correct code with little regard for formatting and readability. To promote best practices in survey programming, this paper will discuss how a custom add-in was developed inside the Blaise Control to make code readable, scalable and scannable.

2 Overview & Approach

As just mentioned, this paper describes a solution for reformatting code using a custom add-in inside the Blaise Control Centre. There were multiple goals intended for this endeavor. Firstly, this programming solution needed to be fast, efficient and unobtrusive. Blaise authors creating production survey instruments needed a tool that would not hinder their work. Secondly, the solution needed to be flexible, so that research institutes can modify the formatting rules to their liking. Each research institute has their own standards and best practices for survey programming, so the solution developed in this presentation needed to easily accommodate them all.

A C#.NET class library was developed as the main add-in component for the Control Centre to interact with. This class library contained several sub-functions that serve as a step-by-step list of processes used to reformat Blaise source code.

First, information is read from the Control Centre application into a string variable. Second, regular expressions are applied to the string data to turn it into a strongly-typed XML document. Third, the XML data is passed into an XSLT to determine how the Blaise source code should be formatted. Finally, the formatted source code is "dumped" back into the Control Centre application. Below is a high-level overview of the custom class library that was developed.



Each sub-section of the diagram above will be discussed in detail further in this paper.

3 Retrieve Text from Blaise

The first sub-function that this custom add-in performs is reading the data from the Control Centre into a string variable. Ideally, the preferred method to achieve this would be to hook into the Blaise Control Centre, locate the active document window, and retrieve its contents into a string variable. However, because of the current functionality of the Blaise Control Centre add-ins, a different and slightly less elegant approach was implemented. Microsoft's .NET libraries were used to simulate a set of keyboard commands; select all (Ctrl – A) to select the entire text of the active window, and copy (Ctrl – C) to copy it to Windows' clipboard memory area. Once in the clipboard memory, another .NET library was used to access the clipboard memory. Its contents are transferred from the clipboard into a string variable that will be used throughout the programming of this custom add-in.

4 Format via regular expressions into XML

Once the text has been extracted from the Control Centre and stored in memory, the next step is to convert the text into some kind of standardized data format. This is an important step, because we

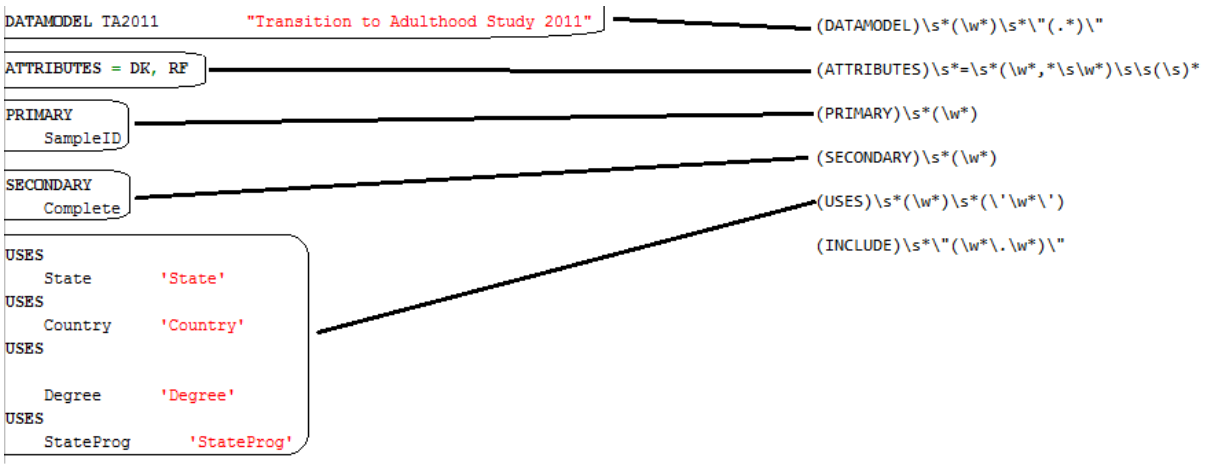
cannot rely on the current formatting inside the Control Centre as a basis for conducting the reformatting that is desired. For example, if two lines of code were in a Blaise instrument that used the USES statement, it would be difficult to account for the layout they currently have, and what they need to be. The picture below describes such a problem.

```
USES
    State       'State'

USES
Country          'Country'


USES
 Degree 'Degree'



USES
StateProg 'StateProg'
```

Example of non-standard formatting.

From this example, it becomes clear that while this may be functionally correct, it is difficult to read, and also difficult to programmatically reformat. This is why a standardized data format like XML is so important.

To achieve this transformation to XML, the original text string is filtered through several regular expressions. Regular expressions are search terms and filters that provide a concise yet flexible means for authors to match strings, and search for character patterns inside a string of text. These regular expressions are what turn the Blaise source code into XML data. Below is an example of the process from Blaise instrument source code to XML transformation



Regular expressions to the right capture certain areas of the Blaise programming, on the left.

As shown in the screenshot, some clever regular expressions were needed to determine where each "chunk" of code starts and ends. It is important to capture these chunks of code accurately, so that the XML data is not malformed. Once the source code is passed through all of the regular expression filters, a complete XML data structure will be created. The XML elements are instrumental for the XSL transformation.

5 Transform XML through XSLT

Thus far, two sub-functions of this custom add-in have been covered. First, the Blaise source code has been retrieved from the Blaise Control Centre and stored in a string variable in memory. Second, the string variable has been filtered through several regular expressions, resulting in an XML formatted data document. The next step in this process is to take the XML data, and process it through an XSLT file to get the final Blaise source that will go back inside the Control Centre application.

An XSLT file is a declarative, data transforming schema used to turn XML type data into data of other types and formats. The XSLT approach was taken for this presentation for two reasons. First, XSLT files were specifically designed for strongly typed markup languages such as XML, and HTML to a lesser extent; other approaches such as the .NET library would be more difficult and time-consuming to implement. Second, XSLT files are easily customizable. This goes back to one of the principal requirements for this project, that the rules for formatting the code should be adjustable for each study center that would use this application. One study center may prefer two tab characters before an INCLUDE statement, where another study center may only wish to have one. Below is an example XSLT.

```
<xsl:template match="root">
  <xsl:apply-templates select="@* | node()" />
</xsl:template>

<xsl:template match="DataModel">
  <xsl:value-of select="'\r\n'" />
  <xsl:value-of select="'DATAMODEL '"/>
  <xsl:value-of select="@ID" />
  <xsl:value-of select="' '"/>
  <xsl:value-of select="concat('&quot;', @Desc, '&quot;')"/>
  <xsl:value-of select="'\r\n'" />
</xsl:template>

<xsl:template match="Attributes">
  <xsl:value-of select="'\r\n'" />
  <xsl:value-of select="'ATTRIBUTES = '" />
  <xsl:value-of select="@Attribute"/>
  <xsl:value-of select="'\r\n'" />
</xsl:template>

<xsl:template match="Primary">
  <xsl:value-of select="'\r\n'" />
  <xsl:value-of select="'PRIMARY '"/>
  <xsl:value-of select="@Key"/>
  <xsl:value-of select="'\r\n'" />
</xsl:template>

<xsl:template match="Secondary">
  <xsl:value-of select="'\r\n'" />
  <xsl:value-of select="'SECONDARY '" />
  <xsl:value-of select="@Key"/>
  <xsl:value-of select="'\r\n'" />
</xsl:template>

<xsl:template match="Uses">
  <xsl:value-of select="'\r\n'" />
  <xsl:value-of select="'USES '"/>
  <xsl:value-of select="'\r\n    '" />
  <xsl:value-of select="@Name"/>
  <xsl:value-of select="'    '"/>
  <xsl:value-of select="@Value"/>
</xsl:template>
```

Sample XSLT style sheet that reformats Blaise statements.

As shown in the screenshot, this XSLT file looks for the “elements,” such as “Primary” and “Uses,” that we created in the first step when we made the XML file. Once the appropriate elements are found, they are replaced with the formatted text. This process is called transforming, and the end result of the transforming process is the formatted text that will ultimately be dumped back into the Blaise Control Centre.

6 Dump clean text into editor

At this point, the formatted text is ready to be transferred back into the Control Centre. As mentioned earlier, the very first step taken in this process was to transfer the text being worked on into the Windows clipboard by way of simulating keystroke commands. For the final step in this process, the text inside the Control Centre will be again selected in its entirety, in the event that the user interacted with the Control Centre during the reformatting process. The formatted text is pasted back into the Control Centre by again making use of simulated keystrokes. At this point, the user can review the formatted code, and decide whether or not it should be committed to use.

7 Lessons Learned

There were several lessons learned in the development of this paper, as well as the development of a proof of concept prototype. First, and perhaps most important, it was discovered that there is no pre-built library provided to authors for accessing multiple open files within the Blaise Control Centre. If one were to try to extend this application to reformat several open files at once it would be a difficult undertaking and require a great deal of advanced programming. At this time, the most efficient approach would appear to be to click on each file and run the custom add-in to clean the source code for each individual file. Furthermore, using the approach of simulating keystrokes to “Select All” and “Copy” allowed authors to input keystrokes manually, potentially breaking the Select All/Copy methods. If there continue to be no Blaise libraries available for accessing open files, it is highly suggested that additional programming be implemented in this custom add-in to temporarily disable the keyboard until the data are stored in the clipboard memory. There are multiple approaches available to achieve this, ranging from writing code to trap keyboard input and prevent it from being processed to using .NET libraries to lock out keyboard input altogether.

Additionally, in the course of developing a prototype, it was discovered that pasting reformatted code back inside the Blaise Control Centre cannot guarantee 100 percent consistent text formatting. For example, if an author was working on a Blaise instrument that had several USES statements, they could manually use spaces and tabs to format the USES statements properly, regardless of the length of the variables being used. Using the custom add-in developed as part of this paper, there is a slight variance in the spacing of these USES statements, because of the length of the variable names. Additional string formatting in the XSLT could address this. These issues, while noticeable, do not appear to provide any significant impediment to further development and usage of this custom add-in.

```
USES
    State    'State'
USES
    Country    'Country'
USES
    Degree    'Degree'
USES
    StateProg    'StateProg'
```

An example showing spacing irregularities between variables and their values.

8 Conclusion

As mentioned in the beginning of this paper, computer code is similar to books, newspapers, or any kind of printed media; it needs to follow standards of readability, scannability, and conciseness in order to maximize productivity and minimize time reading the code. Authors, like writers, need to use their chosen language to provide documents that are not just correct, but cleanly written. This paper has shown in detail how to achieve these best practices for Blaise instrument programming, by using the add-in manager to develop effective third party add-ins. While the Blaise Control Centre does not provide this functionality natively, the custom-add-ins provides a powerful and flexible method for creating custom programming that interacts with the Control Centre. It is hoped that while Blaise continues to evolve and improve as an industry leader for survey programming, more flexibility will be built into the Control Centre as well as the custom add-in manager so that survey authors can continue to strive to use best practices in software development to the benefit of everyone involved in the development and implementation of these instruments.