Extending Blaise Capabilities in Complex Data Collections

Paul Segel and Kathleen O'Reagan,Westat
International Blaise Users Conference, April 2012, London, UK

Summary: *Westat Visual Survey (WVS) was developed for use on studies with deeply nested hierarchical data models. We discuss the use of WVS in CAPI studies, benefits and issues, and lessons learned.*

Introduction

Westat Visual Survey (WVS) was developed to support longitudinal CAPI studies with very deeply nested hierarchical data models. It uses a standard Blaise data model with an additional language field text to combine the Blaise rules engine with Blaise Component Pack (BCP) API calls in order to navigate through the complex rostering. The benefits and issues with that approach are summarized. As a web application, WVS was able to separate the presentation from the rules engine, and backend SQL database storage. Extending navigation outside the rules engine poses some challenges. Lessons learned are discussed including managing the development life cycle with portions of the product in multiple platforms, maintaining compatibility with Blaise versions, and developing automated testing procedures.

Design Considerations

CAPI instruments with deeply nested data structures such as those collecting information about multiple families, multiple persons in each family, and each person's specific history across various event topics present programming challenges. With complex navigation rules, Blaise is the logical choice of platform, but situations occurred where the data model was too large to prepare due to the iterative rostering of entities (persons, families, events, etc.), collection of details about the rostered items and maintenance of relationships between the rostered entities.

Design features for the WVS development included: retaining the Blaise metadata and rules engine, supporting variable length rosters, representing and navigating the deeply hierarchical data structure, and providing a multi-tiered architecture with the Blaise engine providing the business rules, a native Web presentation layer and a data access layer to a SQL Server database. Version control for all components of the system and project were important as well. WVS also includes the ability to search large directory databases, perform CARI recording for CAPI instruments, fine tune performance and support automated testing. Another design goal was to be able to integrate the Blaise and WVS development process and utilize the standard utilities available to support and manage a CAPI project. These design considerations are addressed in more detail below and the overall WVS design is shown in Figure 1.

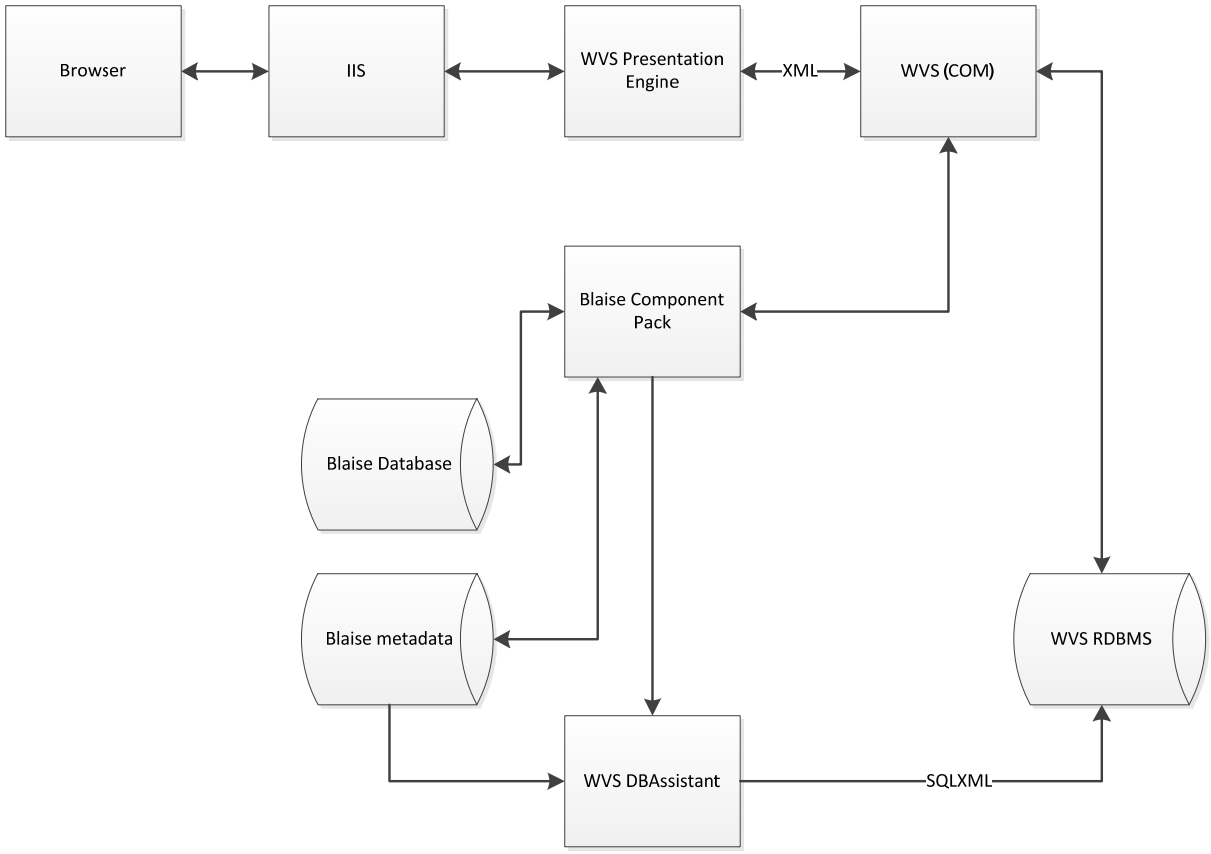

Figure 1. Blaise WVS Design

Retain the Blaise metadata and rules engine

Our instrument development processes are built to support Blaise instruments including specification development, program coding, testing, and documentation. Apart from the complex rostering, Blaise supports the structure and flow of the instrument and retaining the metadata and rules engine works well. Blaise is also used within the detail items and loops, even within the complex question sequences for navigating around non-response.

In order to support the additional features of WVS, we developed a WVS language that was added to the metadata items. The WVS language included such items as definition for roster creation, navigation, and any special presentation specification, similar to the MML multi-media language.

Support for variable length rosters

In order to support roster lists of any length WVS requires language support, a data structure, and a link that remains consistent with the Blaise database. WVS maintains a normal Blaise bdb database file for the rules engine. Although this database would not contain all the data associated with a case, it would be an accurate snapshot to support asking the rules engine what item was next on the route at any given time.

In addition, WVS maintains a SQL Server database with the full case data. Since all the data are collected in normal Blaise blocks, the database structure can be automatically derived from the Blaise data model. The SQL Server schema does, however, try to reflect the entity relationships, and contains additional references to the block location in the Blaise instrument. Much of the action in the data access layer is maintained through SQL bulk loads and driven by SQL XML schema files.

WVS maintains a distinction between defining a roster entity and collecting details about that entity. From an instrument design perspective that distinction helps especially if roster details are collected in multiple locations in the instrument.

Represent and navigate the deeply hierarchical structure

WVS supports two ways of collecting details about a rostered entity or roster navigation: normal Blaise looping and an "Instance Navigator" approach. If the roster data collection is mapped to an array of Blaise blocks, normal Blaise loops can be used to navigate the roster collection. WVS adds some constructs to control the sequencing of the loop, but the routing does not require any action from the interviewer.

Some of the rosters, deeply nested in a hierarchy, would not fit within the Blaise data model size constraints, as large as they are. For these rosters, WVS re-uses a single Blaise block (including any sub-blocks) in the data model, retrieving and storing the appropriate roster entity from the SQL database as the flow enters and exits the block. Although, the roster entities can be sequenced automatically for collection, usability studies suggested that it was preferable to allow interviewers to direct the sequence.

To support that selection, WVS creates an "Instance Navigator" screen that displays each roster entity as shown in Figure 2 with a description and status of the collection, typically labeled "complete", "partial", "not started". An option can prevent re-entry into completed instances, if needed to manage complexity. The ability to select the instance facilitates backing up into previous instances quickly, for example.

Figure 2. Instance Navigator Screen Example

When a new roster collection instance is created, WVS has to maintain the completion status across all instances. WVS performs several activities to do this:

- Write the current instance to the SQL Server database as instances themselves can be nested. The writing requires concordance between the Blaise rules engine and a WVS declaration specifying the levels of the data structure.
- Assess all instances against the rules engine for completion. Since instances in WVS are allowed to interact, the process of assessing the completion status is dynamic, although another WVS declaration can allow previous status to be reported. In the full dynamic version, any current instance is cleared in the Blaise data model (fields reset to an unanswered status), current instance values are retrieved from the database for the entire block structure, and then the block structure is traversed looking for unanswered items.

- Select a new instance. When an instance is selected by the interviewer, the clearing and loading is performed, if needed, for the selected instance.

This process is repeated whenever the routing enters the instance block from either the forward or backward direction.
Deleting a roster item in this scenario also introduces complexity. The same roster, e.g. people in the household, can be used as the basis of collection in multiple locations in the instrument. When a roster entity is deleted, checks are made to assure that the deletion is handled appropriately in all locations.

Provide a multi-tiered architecture

WVS provides a multi-tiered architecture with the Blaise engine providing the business rules, a native Web presentation layer and a data access layer to a SQL Server database. Since WVS separates the rules engine (business rules) from the SQL data access, it further separates the presentation layer. The presentation layer is a dynamically created Web page, using a version of IIS on the CAPI laptop. The Web page performs local field syntax and range checking, and routing within the objects on the page.
A central master WVS component processes the Blaise rules and any associated data to create an XML document describing a screen type. The original version of WVS built on classic ASP used XSLT to transform the XML document into the web page, and the current version uses ASP.Net . The most common screen types include:

- Data Entry screen – supporting multiple fields, possibly with internal routing. In addition to multi-field forms, examples could include an "Other-Specify" screen (the Specify field appears only if an enumerated field has an value of "other"), Quantity-Unit forms where the quantity depends on a previously selected unit.

Figure 3. Entry Screen Example

- Grid screen – built from Blaise tables. The cells can be traversed in any order. An Ajax implementation allows the Blaise rules engine to update selected fields without redrawing the entire page.

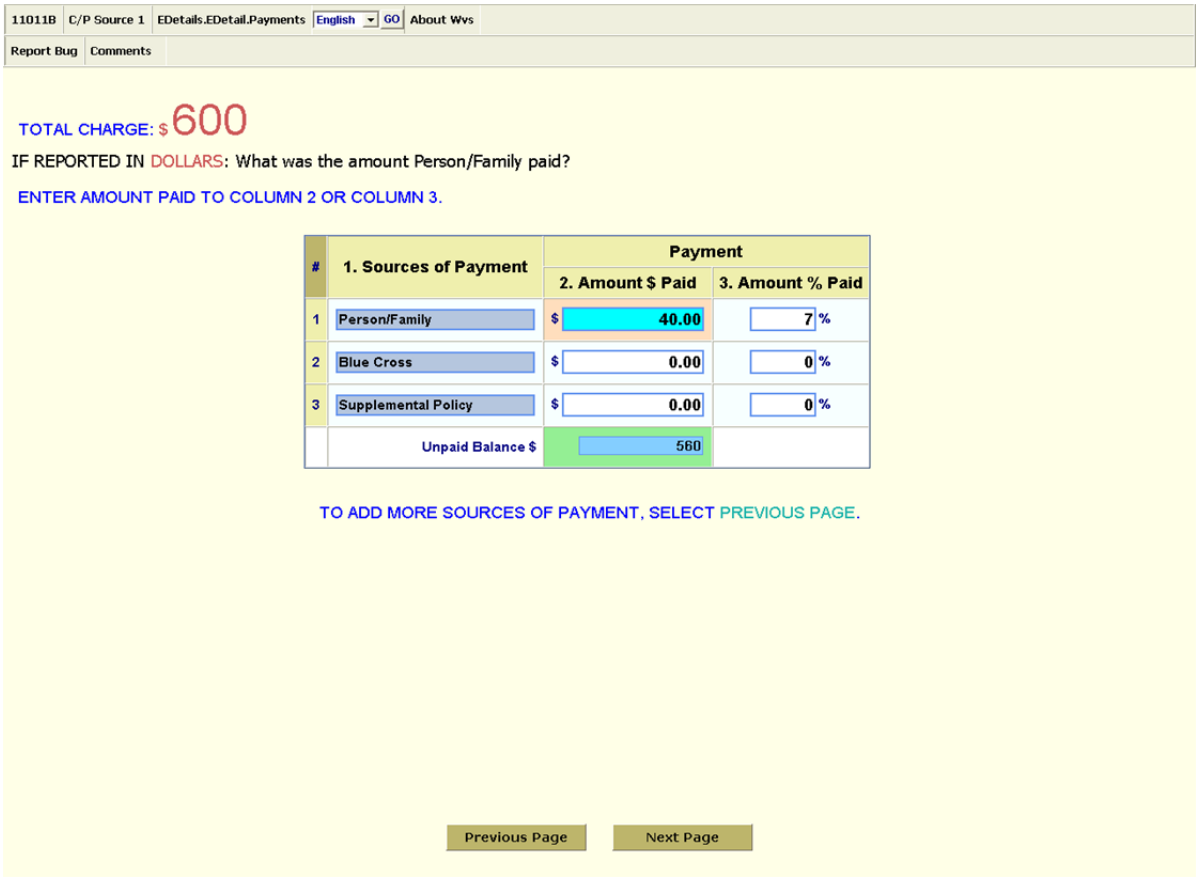

Figure 4. Grid Screen Example

Version control for all components of the project

A project's WVS implementation includes several components in addition to the Blaise instrument. WVS deployments use text versions of these components to facilitate version control, and maintain the ability to compare versions. The components include:

- SQL Server schema (as scripts)
- WVS system configuration items stored in the SQL Server database (as scripts). These configuration items include, for example, tables that manage the roster instances, including their structure and their status.
- SQL XML schemas for the data access

Provide search capability in large directory databases

WVS projects also needed the ability to search large external databases using a variety of search strategies. In the current version, the external databases contains data in fields and WVS flexible specifications for search strategies are defined based on the number of known fields and field values.

Enable CARI recording for CAPI instruments

As projects requested CARI recording to assess CAPI interviewer performance and the quality of the question items, WVS integrates a CARI specification and recording capability. Although historically developed independent of the current Blaise CARI features integrated with Blaise DEP, it uses a configuration similar to the CARI settings file.

Support automated testing

Automated testing provided a means of regression testing new versions of the WVS system. We developed a suite of test scripts to exercise critical features of the system. To retain the ability to test within the range of data structures, script maintenance involved modifying existing scripts to exercise new features. Test data for SQL scripts is maintained and version controlled to match the script path execution. The automated testing tool that we use requires a unique identification for the objects on the web page. Also, in order to maintain the testing scripts more easily with small instrument changes, the object identification has to be consistent for a page. The WVS page rendering builds a tag based on the field name that basically satisfies these requirements. Audit trails are also useful in helping to identify updates needed to some of the testing scripts.

Ability to tune performance

Depending on the size of the instrument and its structure, some of the WVS background activities can require substantial execution time, longer than is optimal for data collection. For example, the process of clearing, loading, and traversing roster instances to assess their completeness can require extensive use of the Blaise rules engine. To tune some of these performance issues, WVS audit trails can be mined to locate bottlenecks, and fine tune performance so that execution speed is at normal levels. WVS has evolved specifications at the item level to choose among various techniques for maximizing performance. Of particular benefit was managing the transition between Editing mode (to load data without rules) and Interviewing mode (that applied the rules) in a way that did not impact performance.

Utilities to manage a CAPI project

As a Blaise instrument at the core, WVS instruments can take advantage of existing utilities developed to support the Blaise project life cycle. Some activities, like extracting case data, require WVS extensions to deal with the SQL Server case data.

Integrated Blaise and WVS instrument development process

Instrument development for the WVS system builds on the normal process for other Blaise instruments. The specification for Blaise fields remains the same and the instrument development starts with a normal Blaise instrument. To insure compatibility with the Blaise rule engine, the full WVS instrument can be run with DEP. Additional steps are required to author the WVS-specific portions of the instrument.

Although there is a default presentation for Blaise field types in WVS, additional WVS commands in the WVS language can be added to control the presentation including page layout, within page routing instructions, and item help references. The web page performs field level validation, primarily from the Blaise metadata specification, although additional validation can be added such as date ranges based on the fields on the page. More complex validation uses the normal Blaise rules when the page is sent back to Blaise.

In addition, WVS commands are required to control the loading of data from the WVS database into the Blaise database. The WVS DBAssistant tool creates XML maps between the Blaise block

structure and the WVS database entities. The WVS commands and front and back covers control the retrieval and storage of the WVS database items.
All the components of Blaise WVS instrument development have text specifications. The text files can be maintained directly with source code control applications. Version stamping is used to provide information both during development and testing, but also for problem reporting from the field. The integration with automated testing allows testing new WVS versions against existing instruments. Although more script maintenance may be required, it also allows the testing of instrument changes.

Lessons learned

WVS projects have been successfully fielded and the system has evolved over time. We learned many lessons during this process as noted below.
The Blaise rules engine is remarkably robust. WVS treats it as a black box. We have empirically derived a sense of what activities are expensive and experimented to find techniques to manage the resources.

The Blaise API has provided all the hooks that were needed to build WVS around the Blaise core. It exposed all the objects needed with great efficiency.

There are subtle differences in the Blaise API that can have considerable impact on performance. WVS relies heavily on DEP behavior modes to manage performance as it manipulates the Blaise data model. The timing of when field values are available has proven tricky.

The ability to integrate automated testing has proven invaluable for regression testing new WVS versions. The ability to consistently and uniquely name screen objects is highly desirable.

We think there are many elements of the Blaise 5 design that would simplify or replace many of the WVS implemented features, and we look forward to exploring Blaise 5 further.