Configuration Management and Advanced Testing Methods for Large, Complex Blaise Instruments

by Steven Newman and Peter Stegehuis
Westat, USA

Abstract

Blaise instruments that Westat creates for its clients are often large and complex. These instruments can include hundreds of files, including source code files, external data sources, and configuration files. Additionally, instrument development is performed by teams under tight schedules. Efficient development and testing requires frequent builds (at least once a week) of the instruments. Thus, these instruments create severe file management challenges. It is imperative to maintain version control of all files, generate and track problem reports, and efficiently test the instrument after bugs are repaired and new features added.

This paper describes some of the applications and methods that we are using to manage version control, error reporting, and automated testing. We discuss how four kinds of tools have been adapted to help manage these functions during the development of instruments. For version control we discuss Microsoft's *Visual SourceSafe*™ and Mortice Kern System's (MKS) *Source Integrity*.™ For person-based interactive error reporting, we discuss our own error reporting system and its integration with MKS *Track Integrity*.™ For automated testing of instruments and automated testing of Blaise as a system, we discuss *WinRunner*™ by Mercury Interactive. The paper discusses the strengths and weaknesses of each type of tool in the context of Westat's experience.

1. Introduction

Configuration management is the repetitive process of source code revision, testing, integration, and problem management. Poor configuration management often causes serious and frustrating problems for software development. To manage a Blaise project successfully, it is essential to track every change to every module in both large and small systems. Version control, in its simplest definition, means keeping one official copy of each source module within the project. There are many commercially available version control programs that can perform this task. This paper examines two commonly used version control applications: Microsoft's *Visual SourceSafe*™ and Mortice Kern System's (MKS) *Source Integrity.*™

More difficult then managing version control is the arduous process of testing and tracking problems in instruments. Westat has developed an error reporting system that allows testers and other users to automatically record critical information about a bug or problem in an instrument *during data entry*. The error reporting is invoked in the Data Entry Program (DEP) by an extra menu option, *Error Report*, which has a short-cut key assigned to it. The *Error Report* menu option launches a Delphi executable program. The DEP automatically passes parameters to the executable, a feature already supported by Blaise. This is extraordinarily useful, as it 1) automatically records critical information such as the primary key, field name, field value, and meta file name; 2) eliminates the use of paper; and 3) greatly reduces the overall burden of error reporting while increasing its accuracy.

Simply discovering and recording errors is not sufficient. Errors must also be reviewed, assigned, corrected, and tested. Tracking error resolution is complicated, labor intensive, and prone to management problems. With that in mind, Westat has integrated its unique error reporting system with a mature and robust error tracking system, *Track Integrity* by MKS. With this integration, errors are accurately recorded *and* entered into a tracking system with a few keystrokes. This paper explains Westat’s automated error reporting and tracking process.

Finally, this paper addresses automated testing of Blaise instruments and automated testing of Blaise as a system and product itself. Automated testing of instruments can be very problematic, as test scripts can be difficult to maintain during development when the instrument is frequently changing. However, after an instrument has become reasonably stable, there is little doubt that automated testing is an extremely useful tool for regression testing and maintenance. Testing Blaise as a product has also been a focus at Westat. As a stable application, it is a perfect candidate for automated regression testing.

2. Version Control

2.1. Importance

Version control of Blaise source code files is the only way to ensure reliable instrument development. The importance of version control cannot be overstated. The entire task of configuration management revolves around the principle that *only one master copy of each source file exists.* There are examples of instruments that were built and shipped without features and bug fixes that developers *knew* they made. Invariably, their changes were indeed made, but were not included in the build because another developer, working on the same module, copied his module over the changes made by the first developer. These types of errors are completely avoidable when version control systems are used.

There are many other advantages of using version control besides the basic function of keeping one official copy of each module. All version control products provide archiving capabilities that not only backup all revisions of each module in the project, but also allows programmers to compare any of the revisions. This is of great value to programmers, who can easily compare changes to modules made by all members of the development team. In this way, debugging is greatly facilitated by version control.

Version control also solves the serious issue of code divergence. Code divergence occurs when there is an instrument release that must be maintained and at the same time the development team needs to add new features and bug fixes for a future release. Code divergence can also occur when an entirely new or experimental instrument is developed based on a revision from the main development path. Code divergence is solved by two methods: labeling and branching. When a project is labeled, every module in the project is given a logical label such as *Version 2.12.0054*. This enables all modules with a given label to be retrieved whenever necessary (for example, to rebuild an older version of the instrument). Branching is the process of taking one file or project in two different directions at once. By labeling and/or branching projects when instruments are built, version control solves the code divergence problem while providing an easy way to build and maintain old versions of an instrument.

Version control also provides an environment that allows for automating the build process. Batch files can be created that confidently use the most recent version of the files needed for the instrument. Automated builds can be written to not only build the instrument, but also to launch automated testing and distribute the instrument to testers and other users.

Version control forces structure on a project. Version control applications require the instrument modules to be in a special directory structure defined within the version control application. Invariably, this forces the version control administrator, with the assistance of developers, to create a logical group of folders in which to keep all the files necessary to build an instrument. The latter point is also significant—in order to build an instrument from the master version control copy, *all* required modules must be present. This frequently forces developers to assess which files are necessary, and becomes a catalyst for more efficient instrument development.

Finally, though difficult to measure, version control is unquestionably cost effective. Version control applications cost anywhere from $600 to $2000 per license, but the losses incurred by releasing inferior products and discovering bugs late in development are far greater. Studies[1] have shown that the later an error is found, the more it costs to fix, and the cost increases exponentially. In one example, software development costs were approximately $75 per instruction and maintenance costs were approximately $4000 per instruction. Version control helps to catch errors earlier. This has been confirmed by Westat's own experience, where projects that were late in implementing version control experienced more costly and difficult problems.

In summary, version control:

- preserves source code integrity by keeping one master copy of each source file
- facilitates debugging by allowing easy comparison between revisions
- solves the problem of code divergence
- provides an environment for automating builds
- forces structure on the project
- is cost effective

2.2. Two Major Version Control Products

There are many version control products on the market. In this section we briefly discuss two of them: Microsoft's *Visual SourceSafe (VSS)* and *MKS's Source Integrity (SI)*. Both of these products provide robust and reliable version control.

In both version control applications, *master* projects are created on servers. Administrators create the master project, manage security settings, create users, and assign user privileges. Developers then "mirror" the master project on their local drives. When a developer wants to modify a module, the module is "checked out" of the master project, giving the developer exclusive editing rights to the module. This means that the developer has a writeable version of the file to edit on his or her local drive.

VSS calls the local areas where developers work *working directories*. SI calls them *sandboxes*. This conceptually and practically embodies the main differences between the two version control applications. SI manages code divergence by creating different types of sandboxes, called *variant sandboxes*. Thus, in SI, a new release is typically maintained as a variant sandbox, which is, in fact, another physical group of directories on a server or other computer. VSS manages code divergence by *branching*. Like the variant sandbox, branching is the process of taking one file or project in two different directions at once. Up to the point of branching, files share a common history because they were one and the same. After branching, they diverge. Each direction is a different development path. VSS tracks branches by making each development path a different project within the VSS working environment.

Both applications provide excellent history tracking, revision numbering, labeling, and tools for viewing and comparing revision changes. For projects that are time limited and that do not consist of years of upgrades and maintenance, VSS is more than adequate. Ultimately, SI probably provides the most flexibility, but with the added cost of complexity.

Another consideration for implementing either VSS or SI is the time required for learning and configuring the systems and for training personnel. For either system, there must be a skilled administrator, probably a mid- or senior-level analyst or developer, who understands the concepts of version control. If the administrator has previous experience with version control programs, he or she can master the basics of either application in a week or less. Developers who are clients on the systems can be taught the basics in less than an hour.

Both systems integrate with error tracking systems. VSS integrates with *Visual Intercept*, by Elsinore Technologies. SI is integrated with another MKS product, *Track Integrity*, a highly configurable change management system that tracks errors, proposals, and work instructions. For those who are interested in a completely integrated change management system that includes proposal and specification integration, problem tracking, and work instructions, as well as version control, SI is more strongly indicated.

[1] Testing Computer Software, Kaner,Falk,Nguyen, pg 31, Boehm study

2.3. Blaise-specific Considerations

When placing Blaise projects under version control, it is important to consider *all* files in the project, including the data files. Typically, version control systems only retain source code modules, include files, specification files, and other *source* files. Files that are the result of compiles or are otherwise generated, such as Blaise data files or object files by C compilers, are typically *not* included in the version control system. However, Blaise projects frequently include generated files that are essential to the project and that rarely change once they are created. Examples include lookup tables (external Blaise files), Maniplus shells, and Manipula setups that connect to other programs. These are candidates to include under version control, keeping in mind that a goal is to be able to build an instrument completely from the contents of the version control project. Ultimately, including these files under version control can be evaluated on a case by case basis.

2.4. Possible Blaise Enhancements to Help With Version Control

There are problems associated with identifying the version of Blaise instruments. These are most evident when new instrument versions are distributed (with and without new data models) and when problems are reported about an instrument.

Configuration management procedures assign a version number to an instrument, such as *3.12.0008*, where *3* is the major version number, *12* is the minor version number, and *0008* is the build number. (This is similar to the scheme that the Blaise system itself currently uses.) In this scheme, the build number is incremented at each build. The minor version number is incremented for minor instrument changes or milestones, such as when the data structure changes. The major version is usually reserved for major functionality such as adding multimedia to an instrument.

When distributing updated versions of an instrument, electronically or otherwise, managing the update process for hundreds of interviewers is a formidable task. This becomes even more difficult when the update also includes a change in the data structure. In this case, previously collected data cannot be read using the new data model, requiring the data to be restructured. Without clear identification of the instrument version, these tasks become very difficult.

The inability to identify an instrument version number also causes problems when errors are discovered in an instrument. At this time it becomes imperative that the instrument version be recorded along with the problem report. Not all interviewers receive their updates at the same time, therefore, we cannot assume that all interviewers are running the same instrument at the same time.

These scenarios lead us to the follow questions:

1. When a user receives an update or reports a problem, what instrument version is being used and how can it be found?
2. For a Blaise data file, how do we know which instrument version it was created with and which Blaise data model to use to read it?

Resolving these questions is facilitated when instrument versions are easily known. Interviewers can be confident that they are using the latest version, and data model “matching” can be managed.

To help resolve these questions, in some projects we now code the instrument version as a field of the data model. This version field has to be updated manually every time there is a new build. We then display the contents of the version field on the screen of the questionnaire –typically on some easy to remember place, such as the first question in the questionnaire. In this case, if an interviewer reports a problem, then he or she can press the *Home* key while in the DEP and read the instrument version. This is not the preferred way to display the instrument version. It would more robust and convenient if we did not have to hard code it in the data model and if it were accessible from any screen in the questionnaire.

A possible Blaise enhancement to make the instrument version more accessible would be to include it in the MI file of a prepared data model. The MI file already stores the date and time of preparation. If the instrument version were an element of a Blaise project file (bpf), the major and minor parts of the version could be set explicitly and the build number portion could be automatically incremented every time the main data model was parsed successfully. The instrument version could then be written to the MI file. The DEP could then use a menu option, such as *Help* or *About*, to display the instrument version number. In this case, the Structure Browser could also display the version, since it reads the MI file.

Solving the problem of data model matching is more difficult. Hard coding the instrument version in a data field does not solve this problem, because even if the version is part of the data record, we can’t read the data until the instrument version is known. Furthermore, having interviewers send the MI file, which is frequently very large, with every data file is not practical. We currently have a non-elegant solution for this dilemma—we automatically create a text file with the version number of the instrument. The interviewer sends the text file along with the Blaise data file (BD file). We can read the text file and know which data model version to use to read the BD file.

A better solution can be achieved by using the MI file. In the same way that we suggested storing the instrument version in the MI file, it may be possible to copy it from the MI file to a generally accessible part of the BD file. In this case, a Blaise enhancement could provide a way to read the information from the BD file, perhaps by using Manipula. This could be read regardless of which version of a data model the BD file was created with. This would still require a two-step process—first reading the version number from the BD file and then pointing to the right MI file—but it would be more manageable and less error-prone than using text files.

3. Testing and Error Reporting

"Software testing is defined as the execution of a program to find its faults."[2] More time is spent on testing than in any other phase of software development, and there is no question that software that is not tested will not work. Software testing is difficult to design, time consuming, and repetitive. In addition, testing is only the first part of a cycle of error resolution. The typical process consist of seven steps:

1. testing
2. error discovery
3. error recording
4. assignment
5. fixing
6. unit testing
7. integration

There is great complexity and bureaucracy in tracking these steps. At the most basic level, testers can record the problems they discover on problem sheets. The problem sheets can be sorted, distributed, and tracked. However, this is usually not successful. At the very start, problems are not accurately described and critical information required for resolution is not recorded. Problems recorded on paper are difficult to track and archive efficiently. Automating the process is the only way to ensure success.

3.1. Westat's Automated Error Recording System

With this in mind, Westat developed an error reporting system that can easily record errors during data entry *and* automatically enter the errors in a problem tracking system, *Track Integrity (TI)* by Mortice Kerns Systems. This is extraordinarily useful, as it: 1) automatically records critical information such as the primary key, field name, field value, and meta file name; 2) eliminates the use of paper; and 3) greatly reduces the overall burden of error reporting while greatly increasing its accuracy.

The error reporting executable, a Delphi program, is integrated with the instrument. A menu option that runs the executable is placed in the DEP's *Options* menu and a short-cut key is assigned to it. When a problem or proposal is discovered during data entry, pressing the short-cut key invokes the dialog shown in *Figure 1*.

[2] Managing the Software Process, Watts Humphrey

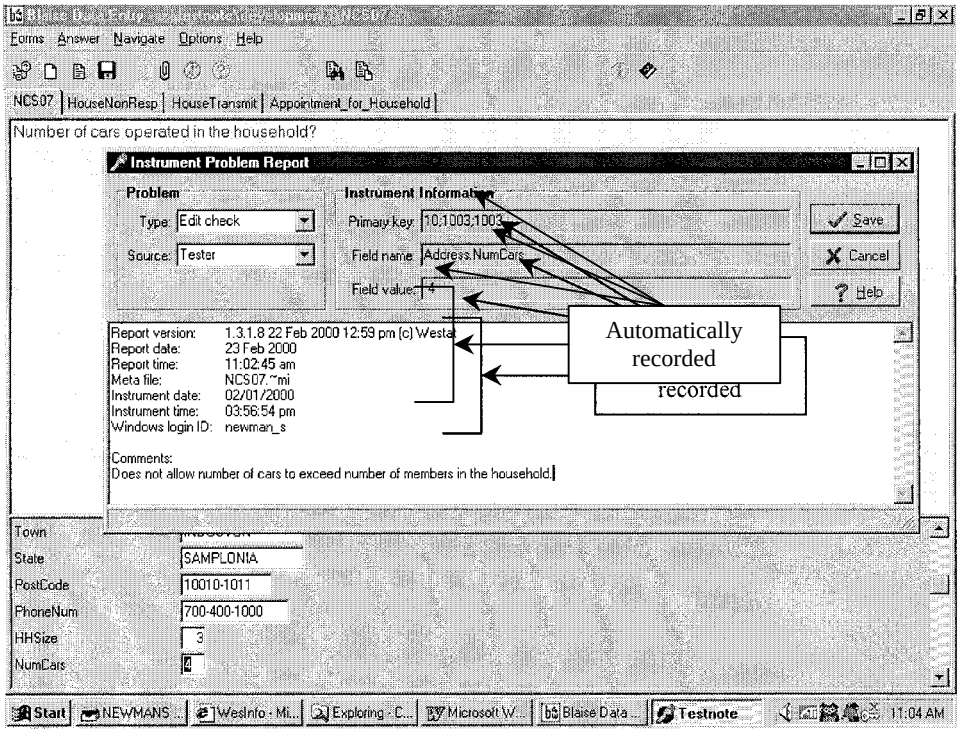

Figure 1: Dialog for Automated Capture of Problem Information

The primary key, field name, field value, report date and time, meta file name, instrument date and time, and the Windows login ID are all captured automatically. The user only selects the *Type*, *Source*, enters a *Comment*, and presses the *Save* button. When the *Save* button is pressed, the problem report is inserted into a TI database. If the TI database is not available, such as when a user is using a laptop, a text file is written which can later be automatically integrated into the database.

The efficiency of this process for recording errors cannot be overstated. Of particular interest is the great amount of information that is recorded automatically. Furthermore, the selections in the *Type* and *Source* list boxes are configurable, so that the list of choices that appear in the *Type* and *Source* list boxes can be easily determined by the administrator. Errors are accurately and easily recorded, which greatly reduces the burden of testers and other users. Because errors are entered into a tracking system, they can be traced throughout their resolution. Once errors are in the database, they can be reviewed, assigned, corrected, and tested.

Frequently, a text description of a problem is not sufficient and a screen capture is required for completeness. Though not currently implemented, it would be very simple to acquire a screen capture on demand. Significantly, the TI database supports attaching a file to a problem report, so the infrastructure already exists to associate files (screen captures or other file types) with a problem.

In the next section we will show an example of how *Track Integrity* records and displays problems and how Westat has integrated its automated error reporting capability shown in *Figure 1* with *Track Integrity*.

3.2. Track Integrity Database Example

Track Integrity is a highly configurable change management application. It controls the software configuration management process by tracking the relationship between three classes of inputs: problems, proposals, and work instructions.

To illustrate how a problem is recorded, we will show an example of the TI database. *Figure 2* shows the problem overview display from TI. *Problem ID 103* (the highlighted entry) is the most recent entry, and is the problem report example recorded in *Figure 1*.

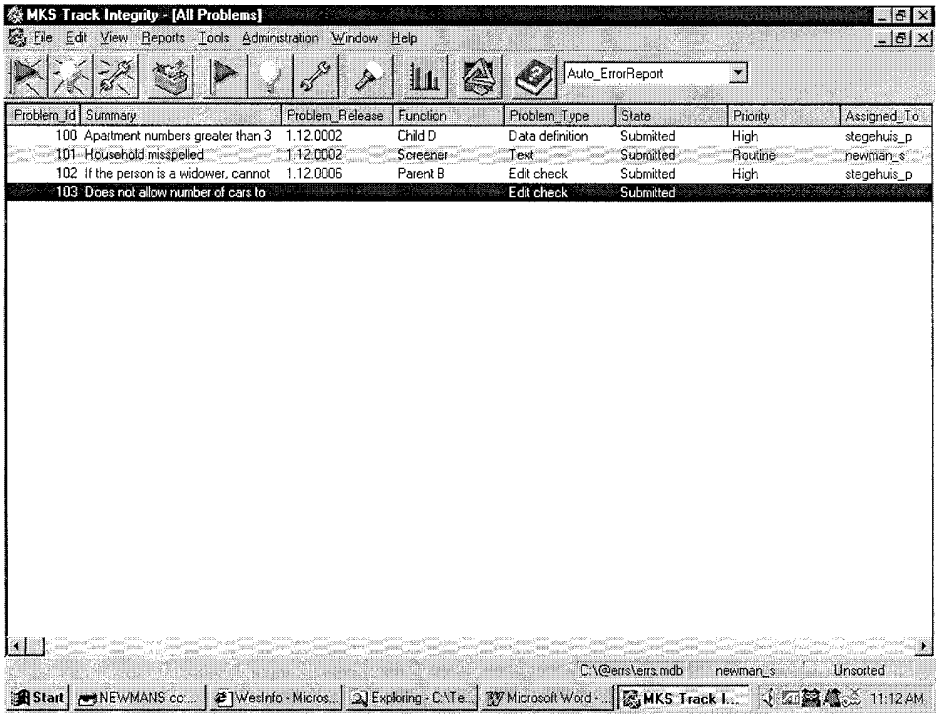

Figure 2: TI Problem Overview Display

Figure 3 shows a detail of *Problem 103*, and contains some of the information passed by Westat's error reporting system, specifically the *Type*, *Summary*, and *Description*.

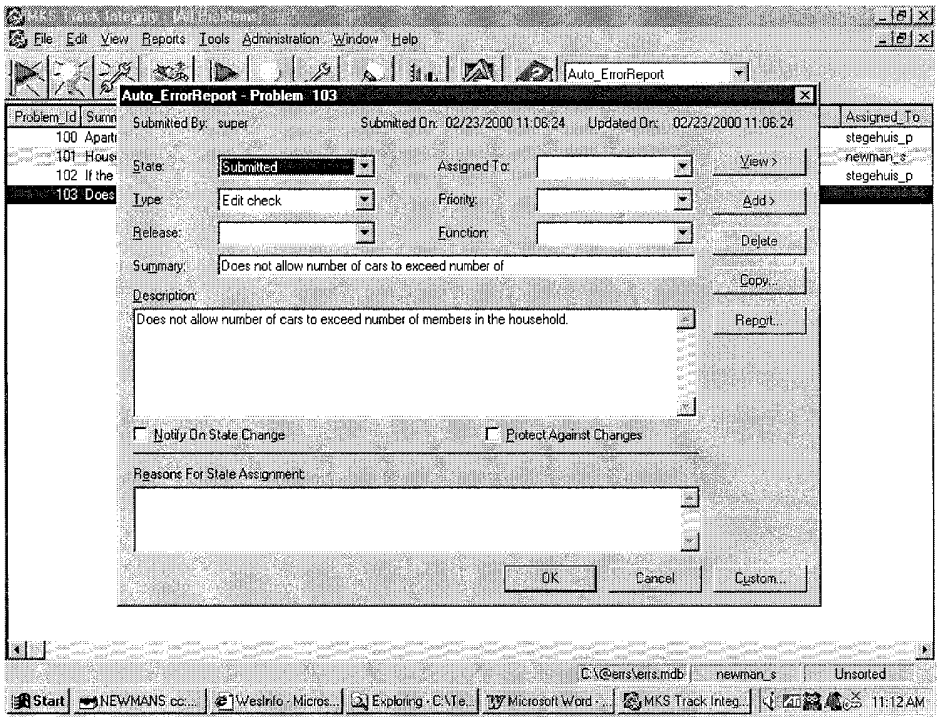

Figure 3: Detail of TI Problem Report

Pressing the *Custom* button in the lower right-hand corner displays the rest of the information transferred by the error reporting system. This is shown in *Figure 4*.

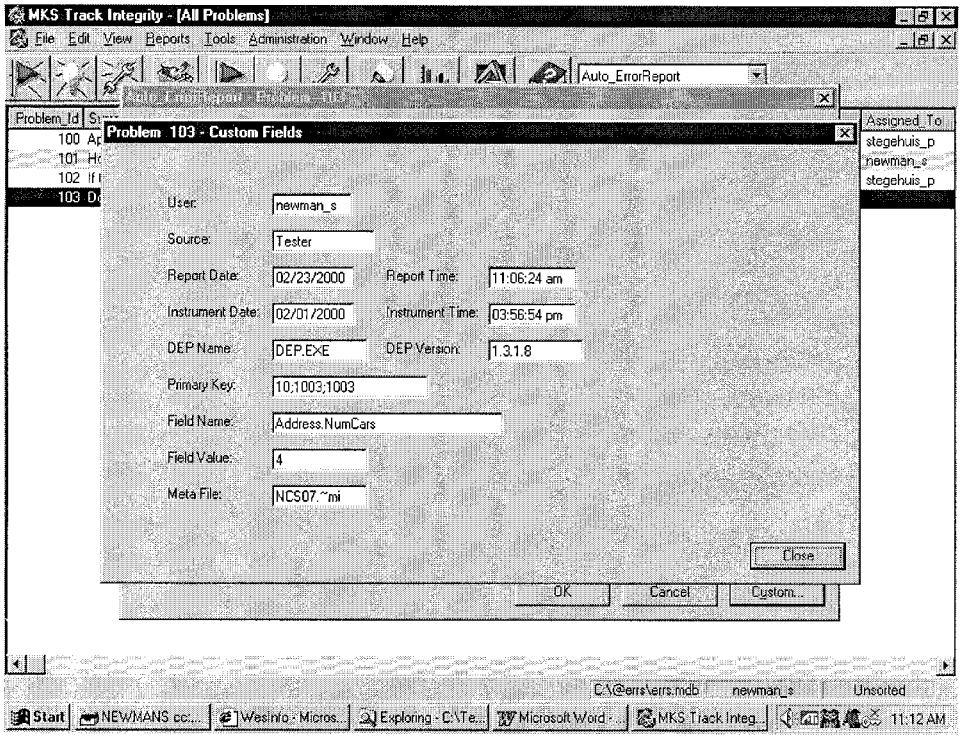

Figure 4: Customized Portion of TI Problem Report

3.3. Conclusion

Automating the error recording process and integrating it with a tracking system can profoundly reduce software development costs while simultaneously producing more error free instruments. By reducing the burden of reporting, testers have more time to execute test plans. Problems tracked in a robust tracking system can be resolved with confidence.

4. Automated Testing

4.1. Benefits of Automated Testing

Software managers and developers are being asked to turn around their products within tighter schedules and with fewer resources. More than 90% of developers have missed ship dates. Missing deadlines is a routine occurrence for 67% of developers. In addition, 91% have been forced to remove key functionality late in the development cycle to meet deadlines.[3] Automated testing can help alleviate these pressures. There are three significant benefits to combining automated testing with manual testing: 1) production of a more reliable instrument, 2) improvement of the quality of the test effort, and 3) reduction of the test effort and minimization of the schedule.[4]

Ultimately, because of the repetitive nature of testing, particularly regression testing, the process must be automated. Although all test phases—unit, integration, and system—can profit from automated testing, it is important to distinguish the types of tests that benefit the most from automation:

1. Performance testing—confirming that response times of the instrument are within acceptable limits
2. Stress testing—ensuring the instrument runs under maximum loads and high volume scenarios
3. Regression testing—performing identical tests on an instrument before and after a bug has been fixed or a new feature added. A regression test allows the tester to compare expected results of a test with the actual results.

[3] Automated Software Testing, Dustin, Rashka, Paul, pg 3
[4] Ibid, pg 37

4.2. An Overview of Automated Testing

Automated testing tools such as *WinRunner*™, by Mercury Interactive, provide a way to record and play back mouse movements and keystrokes. These recordings are called scripts. Scripts are written in special scripting languages, similar to Visual Basic or C, and can be corrected and customized. Scripts that only play back keystrokes and mouse movements would only test an instrument's graphical user interface (GUI). That is, they would only confirm that a new build of an instrument accepts recorded inputs. They could not detect if the *position* of the input fields on the page have changed or if features not affecting the input sequence have changed or been added.

For robust testing, scripts must do more than simply play back mouse movements and keystrokes. They must also do *object testing*. Object testing records the properties of objects in an instrument, including non-visible properties that cannot be tested manually. For example, an *object* or *control* would be any input, display, or interface device within an application, such as edit boxes, list boxes, form pane, buttons, and so on. A *property* would be an attribute of the object such its location on the page, height and width . *Figure 5* is a page from a typical instrument. Using *WinRunner*™, we can capture the properties of the field labeled *Employer*, for which *Westat* has been entered. *Figure 6* shows the dialog displaying the properties of *TInputLine*, which is the name of the control containing the entry *Westat* in *Figure 5*. The list of attributes for *TInputLine* appear as a list in the name column. It includes the height, width, and X,Y pixel positions of the control. If there were a new build and, for some reason the position of the control changed, *WinRunner*™ would report an error.

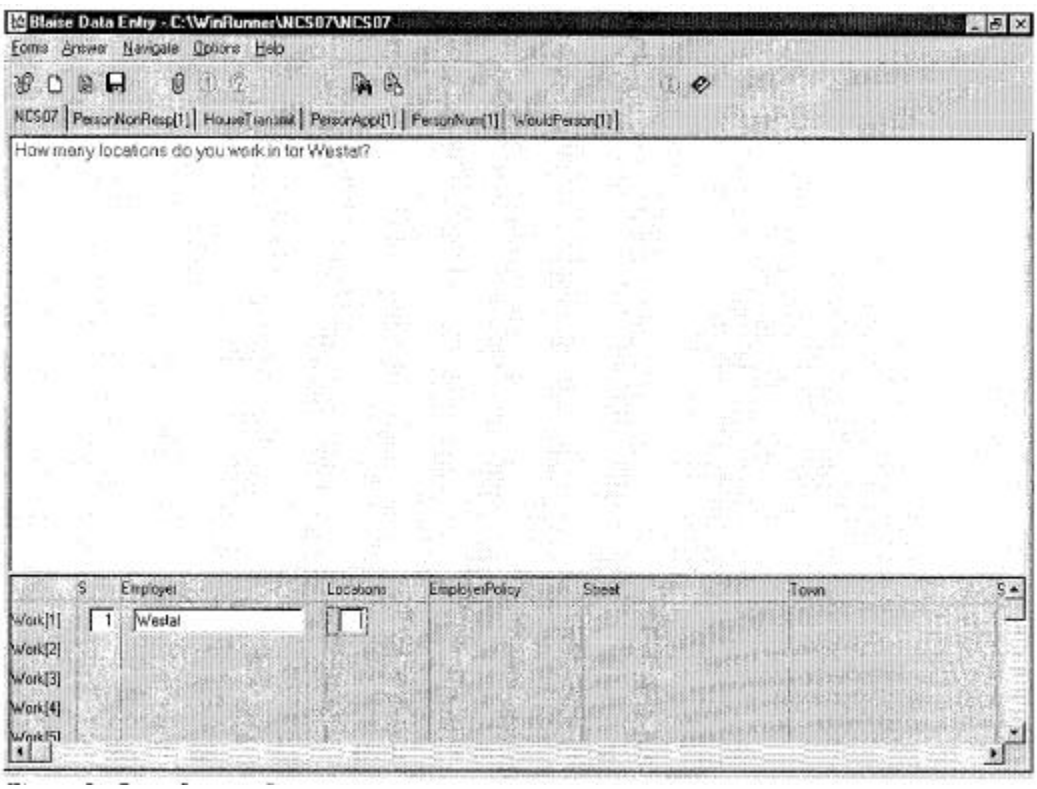

Figure 5: Page from an Instrument

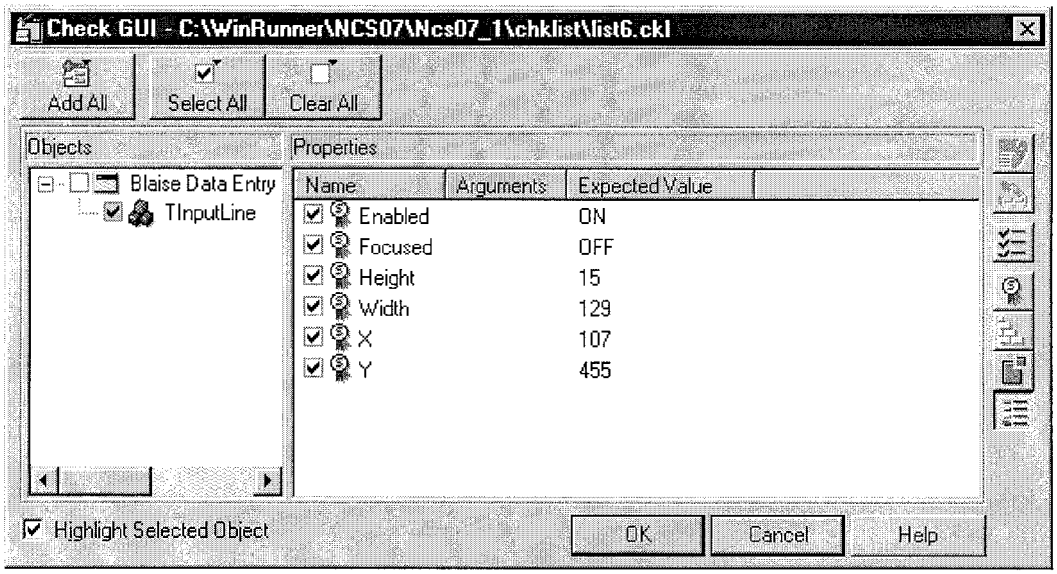

Figure 6: Properties of Object TInputline

Testing tools can also provide image comparisons, where a baseline screen image is obtained and compared to an image acquired during testing of a new build. *Object Properties*, *Region Image*, *Object Data*, and other comparisons are referred to as *test cases*, or *checkpoints*. A test case acts as a validation point for the application under test. Test cases are created to capture and record information about the current state of an object in the application under test. The test case then stores the information as a baseline of expected behavior. As subsequent builds of the application become available, the recorded test procedures are played back against the new builds to compare them with the established baseline. When a test procedure is played back, the testing tool repeats the recorded actions. Each test case recaptures the information and compares it to the baseline. If the information is the same, the test case passes. If not, the test case fails.

4.3. Testing the Blaise System and Blaise Instruments

It is important to distinguish between testing Blaise as a system/product and testing an instrument that was developed using Blaise. In the former case, testing is performed on a stable, mature application. There are typically few GUI changes from one Blaise version to the next and scripts written for one build should easily work on another. Testing instruments that were developed with Blaise is more problematic. Since there are frequent and intentional changes to the instrument during the development phase, previously recorded scripts quickly become out of date and unusable. However, as instruments become mature and stable, automated testing becomes more useful. As an aid for updating scripts, testing tools provide a feature that allows a script to be updated with new or changed information during play back. This makes minor updates more manageable.

The capability to recognize Delphi object properties requires that Statistics Netherlands compile files supplied by the testing tool manufacturer when preparing the Blaise system. This extends the automated testing ability from only comparing screen images to checking Delphi specific object properties. The extent to which a testing tool can recognize Delphi objects and properties is determined by the robustness and completeness of the files the testing tool manufacturer supplies to Statistics Netherlands. To support Delphi, the manufacturers must create up-to-date files when Delphi versions change. This commitment to Delphi support is imperative when evaluating a tool for Blaise testing.

Automated testing of Blaise as a system has been a focus at Westat. In this case, we have a stable system that, upon new releases, can be tested using various data models. This is a classic example of regression testing. Blaise tools such as Manipula and Hospital can also be tested. *Figure 7* shows a flow chart for the process of testing the Blaise DEP, where each **n** represents a different data model selected to optimize testing for different aspects or targets for each test. Example aspects include *normal interview*, *extreme navigation*, *parallel blocks*, *lookups*, etc. Note that there are two paths of testing: a baseline branch and a testing branch. The baseline branch stores the information as a baseline of expected behavior. The test branch plays back recorded procedures with new builds to compare their results against the established baseline.

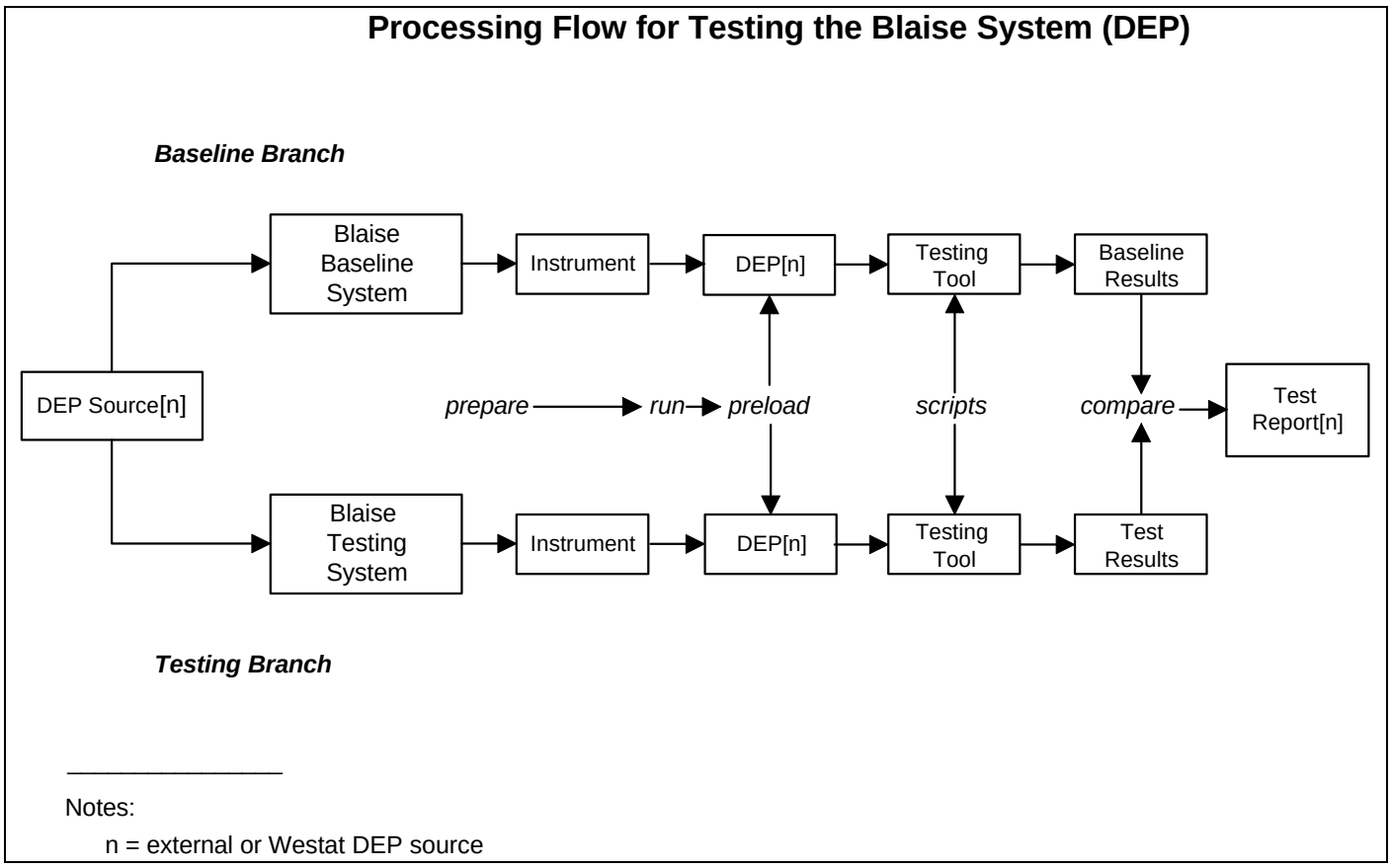

Figure 7: Processing flowchart for testing the Blaise DEP

Figure 8 is a Requirements Tracing Matrix for testing the Data Entry Program (DEP). Westat has created similar matrices for testing the Blaise Control Centre, Hospital, Cameleon, Manipula, CATI, and other Blaise system components. Each row represents an aspect of the DEP being tested. A column exists for each data model being used for testing. Within each cell is the name of the script that meets the required aspect test for each data model. To avoid unnecessary and redundant testing, there are empty cells indicating scripts that are not required.

Instrument Names	**Diary**	**Eu9707s1**	**DACFINFA**	**HLFS**
Aspects				
Normal interview	DIAR0001	EU970001	DACF0001	HLFS0001
Interview with edit failures and other navigation	DIAR0002	EU970001		HLFS0001
Extreme navigation	DIAR0003			
Edit invoking, edit navigation	DIAR0001	EU970001		HLFS0001
Parallel blocks, parallel tabs	DIAR0001			
Language switching	DIAR0004	EU970002		
Functions keys	DIAR0001			
Lookups		EU970003		HLFS0002
Browse records and other data file access		EU970001		
Coding – hierarchical		EU970001	DACF0001	HLFS0001
Coding – trigram	DIAR0005	EU970004		HLFS0001
Coding – alphabetical				HLFS0001
WinHelp		EU970002		
Multimedia		EU970001		
Audit trails	DIAR0006		DACF0001A	
Stress		EU970004		
Display features	DIAR0001			
Data storage		EU970001		

Figure 8: Requirements Tracing Matrix: DEP

The complexity of testing such a large system becomes immediately apparent. There are hundreds of scripts to run in various combinations to get complete regression testing coverage. Testing tool programming languages provide a way to run these script combinations. This means that we can play back a row, column, or any combination of scripts as one test, rather then executing each script individually. This provides great flexibility in focusing the testing on suspected weak areas. Scripts can run overnight, with obvious savings in cost and time. The testing tool automatically records test results.

4.4. Final Comments on Automated Testing

There is a substantial learning curve and commitment associated with learning and implementing automated testing tool software. In particular, while recording scripts can be relatively simple, to implement a robust automated testing system, the scripting language must be learned. As with any high level language, this takes some time and experience. When testing the Blaise system, for example, the Requirements Tracing Matrices indicate the numerous scripts and scenarios that have to be addressed. A “front end” programming task, written in testing tool script language, could provide the user with an interface to select which scripts to run. Tasks such as pre-loading data and launching instruments need to be programmed. Furthermore, testing tool systems come with management software that must be implemented to successfully plan, track, and report the testing process. Finally, as distributed and Web-based systems become the rule, automated testing tools can simulate hundreds of users for load testing—a topic we can address in the future. Given the growing complexity of surveying instruments and scarcity of resources for testing, time spent learning and implementing automated testing is time well spent.

References

Humphrey, Watts S. 1990. *Managing the Software Process*. New York: Addison-Wesley Publishing Company

Kaner, C., Falk, J., Nguyen, H. 1993. *Testing Computer Software*, 2nd ed. New York: Van Nostrand Reinhold

Dustin, E., Rashka J., Paul J. 1999. *Automated Software Testing.* Reading, Massachusetts: Addison-Wesley Publishing Company

Microsoft Corporation. 1996. *User's Guide, Microsoft Visual SourceSafe*, Version 5.0, Microsoft Corporation

Mercury Interactive. 1999. *WinRunner Tutorial,* Version 6.0, Sunnyvale, California: Mercury Interactive Corporation