Benefits From Automated Testing at the Business Object Level – Case Study

Alison Edye
Benefits from automated testing at the business object level

Alison Edye

Solution 6
Level 21 Town Hall House
456 Kent Street Sydney
NSW 2000 Australia.

Tel +61 2 9278 0369
Fax +61 2 9278 0555
alison.edye@solution6.com
www.solution6.com

© Solution 6 Pty Ltd 2000
Key points

- Test scripts are developed in conjunction with component development
- Automated testing starts early in the lifecycle
- Manual testing can focus on user interface issues

...demonstration
Product under test

Solution 6 is a world leading e-Business solutions partner to Professional Advisors, Business and Government across the three core competencies of e-Commerce, Professional Services and Business Software.

- **CABS2000**
  - client/server practice management system
  - multi-tiered (COM, ODBC/API)
  - notebook to enterprise
  - global client base
Test goals

- Defect identification, quality control
- Stability, fast verification
- Achievable test coverage
  - GUI and business functionality, cross platform, sociability
- Effective use of automated tests
  - timely, robust, portable, maintainable, repeatable, reusable
Why automate?

Rapid change and development
more features, growing complexity

+ Customization
database, business rules, forms

+ Variation
OS, DB, mail transport, time zones, regional settings, direct, indirect, remote, web

= Need to test fast and frequently
Component based architectures

'industrialization' of software development
Product architecture

Presentation tier

Applications

Imports

Off-line transactions

Business Object

Business tier

SOE (Data API, security)

Data services tier

Access

SQL Server

Oracle
Business objects

- **Objects**
  - data and behaviour
  - interface: properties, methods and events

- **Business objects**
  - represent real ‘objects’ and concepts
    - eg client, staff member, expense item
  - have relationships and rules
  - *should* provide access to all application functionality

- **Encouraged by COM and CORBA**
Evolution of automated test system

are we there yet?

Integrated process

Object level - regression technology

Object level - data driven keyword generic interface

Object level - data driven keyword direct interface

GUI object harness - modified/capture replay

Product GUI - modified capture/replay

Testing

Maturity Model 5: “Automated tools totally support the running and re-running of test cases…”
GUI level test automation

- Necessary for testing GUI functionality
  - not needed for testing all business functionality
- Needs a working GUI
  - start later, after manual test (ie a retest)
  - maintenance can be a nightmare
- Complex test tools
  - object recognition, timing, unexpected windows, coding for robustness
  - takes focus from test objectives
Object level testing

■ What does a typical business object do?
  - process input - validate, apply business rules, return errors
  - query and update the database
  - call other objects

■ What should we test?
  - methods and properties using appropriate arguments, values and sequences
  - database values after updates
Object models

- Navigating the object hierarchy

**Diagram:**
- **Application**
  - **Workbooks**
  - **Workbook**
    - **Worksheets**
    - **Charts**
  - **Worksheet**
  - Collection item **Worksheets(“Sheet1”)**
Object level test approach

- **Comprehensive but selective tests**
  - don’t try to test all paths through all objects
  - select high value, well factored test cases
  - measure code coverage

- **Test in a system environment**

- **Test business scenarios**
  - define interfaces and use cases
  - understand object hierarchy
Test Tool Architecture

Business tier

- Business Object
- SOE (Data API, security)
- Access
- SQL Server
- Oracle
- File utilities
- Non-GUI applications

Data services tier

Shell file
Auto Script (Text Files)
Data definition (Text Files)

From Excel Work Book

Generic Object Interface Test Tool

Log
Test Plans
Reports
Current
Master

To Comparator Work Book

Non-GUI applications
File utilities

SOE (Data API, security)

SQL Server
Oracle

Access

Log
Test Plans
Reports
Current
Master
Generic object test tool

- Interprets data driven keyword scripts
- Creates instances of objects
- Calls methods and properties
- Performs direct database queries
- Checks for expected results and values
- Provides test case comparisons
- Provides control for unattended testing
Data driven keyword scripts

- Easier to learn, simpler than GUI testing
  - no windows or complex programming
- Not reliant on GUI tool experts
  - QA analyst - test design and data creation
  - Tester - script development, script execution
  - QA analyst/developer - test tool support
- Can focus on test cases, not tool
- Do need
  - management, support, champion and mentors
BONAME: CABSBilling.Bill

LOGMSG: *N Billing BO
LOGMSG: *D Billing Functionality raise New Fee Bill with 2 engagements

LOGMSG: *D Add Client details (FRD010)
PARDEF: sClientNo  FRD010
PARDEF: sBType    FEE
METHOD: ADDNEW    -1 0

LOGMSG: *D Add Engagement details for engagement FRD010010 and FRD010030
LOGMSG: *T ADDTAXBILLENGAGEMENT (Add 2 engagements)
LOGMSG: *R No errors expected

SETPROP: BillDate  2002/6/1
PARDEF: sEngNo     FRD010010
PARDEF: sLocTax    ~
PARDEF: sNatTax    T2
METHOD: ADDTAXBILLENGAGEMENT -1 0
Effective scripting

- Use scripts during development
  - auto TEST, not just RE-test, find defects
- Include checks for expected outcomes
  - better than captured baseline comparisons
  - add intellectual capital within script
- Create script libraries, data files
  - modular design, run one or run a sequence
- Keep scripts and baselines current
Object script use

- Unit, integration and system testing
- Test environment set up
- Test database generation
- Measurements - memory, timing
- Test management and control

Re-use test assets throughout lifecycle
Test development issues

- Current component specifications
- Fat applications, technical components
  - testers may need programming skills
- Test design adequacy
  - ‘false positives’ give a false sense of security
  - check database for updates and triggers
  - audits are valuable automated test tools
- Test database adequacy
  - comprehensive, clean, up-to-date
Making it work

- Training, test plan and code reviews
- Extend script and quality ownership
- Deliver scripts with components
- Component defect tracking and priority
- Developers run scripts after code changes
  - check coverage and update scripts
  - find defects before build, saves time overall
- Regular builds, regression runs
Regression tests

- Need to be very robust
  - object level tests don’t break easily

- Tight configuration management
  - structured, versioning, scripts and logs

- Simple, quick to run and check
  - flexible, unattended, auto result checks
  - ‘green pass’, result summaries
  - detailed logs for error identification
Object level regression tests

- Fast verification, frequent test iterations
  - *eg* ASAG: 42 suites, 16 business objects
    - executes ~150,000 lines of code
      (~11,000 functions) in under 3 hrs
    - takes one person ~2 hrs total to set up multiple environments and verify results
  - *eg* system test and database build ~ 2hrs
- Confidence in business function stability
- Manual testing can focus on GUI issues
Regular build cycle

- Database cut off
- Component cut off
- Build
- Release build to floor

- Post results
- Manual result check
- Database smoke test
- GUI smoke test
- Regression tests auto system test and database generation

Can be daily - usually 2 per week
Current approach

- Automate component and business functionality tests at the object level
  - early, fast, robust, reusable
  - portable, easy to use
- Use GUI level testing to verify UI navigation and functionality
  - GUI tools for smoke tests, sociability
- Use manual testing for complex, creative and demanding tests
Has object level testing helped us towards our goals?

- Defect identification, quality control
- Achievable test coverage
  - GUI and business functionality, cross platform, sociability
- Stability, fast verification
- Effective use of automated tests
  - timely, robust, portable, maintainable, repeatable, reusable
Lessons learned

- Use appropriately skilled people
- Review test design
- Build and verify test databases
- Extend ownership of automated tests
- Manage component level defects
- Allocate resources to keep tests ‘living’
- Do need
  - clear component interface specifications
  - management support and champion
Benefits

- Automated testing can start early
- Simple test interface and tool
- Low maintenance, increased coverage
- Fast verification and frequent iteration
- Identify defects before the build
- More time for creative testing

Definitely worth the effort!
Test drive...
Resources

Links to articles, books, samples, tools ...

www.cetus-links.org/oo_business_objects.html

www.rstcorp.com/resources

www.sqa-test.com

http://members.aol.com/sascanagl/Default.htm

Newsgroup: Computer Software Testing
Questions
Product architecture

Presentation tier

Applications

Imports

Off-line transactions

Business Objects

SOE (Data API, security)

Data services tier

Access

SQL Server

Oracle

Business tier
Test tool architecture

Shell File
Auto Script (Text Files)
Data Definitions (Text Files)

From Excel Work Book

Generic Object Interface Test tool

Business Objects
SOE (Data API, security)
Non-GUI applications

Access
SQL Server
Oracle

Log
Test Plans
Reports
Current
Master
To Comparator Work Book

File Utilities

Data services tier

Business tier
### Script syntax

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Argument(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BONAME</td>
<td>CABSBilling.Bill</td>
<td></td>
</tr>
<tr>
<td>LOGMSG</td>
<td>*N Billing BO</td>
<td>* Billing Functionality raise New Fee Bill with 2 engagements</td>
</tr>
<tr>
<td>LOGMSG</td>
<td>*D Add Client details (FRD010)</td>
<td></td>
</tr>
<tr>
<td>PARDEF</td>
<td>sClientNo</td>
<td>FRD010</td>
</tr>
<tr>
<td>PARDEF</td>
<td>sBType</td>
<td>FEE</td>
</tr>
<tr>
<td>METHOD</td>
<td>ADDNEW</td>
<td></td>
</tr>
<tr>
<td>LOGMSG</td>
<td>*D Add Engagement details for engagement FRD010010 and FRD010030</td>
<td></td>
</tr>
<tr>
<td>LOGMSG</td>
<td>*T ADDBILLENAGEMENT (Add 2 engagements)</td>
<td></td>
</tr>
<tr>
<td>LOGMSG</td>
<td>*R No errors</td>
<td></td>
</tr>
<tr>
<td>SETPROP</td>
<td>BillDate</td>
<td>2002/6/1</td>
</tr>
<tr>
<td>PARDEF</td>
<td>sEngNo</td>
<td>FRD010010</td>
</tr>
<tr>
<td>PARDEF</td>
<td>sLocTax</td>
<td>~</td>
</tr>
<tr>
<td>PARDEF</td>
<td>sNatTax</td>
<td>T2</td>
</tr>
<tr>
<td>METHOD</td>
<td>ADDBILLENAGEMENT</td>
<td>-1 0</td>
</tr>
</tbody>
</table>
Test tool architecture

Generic Object Interface Test Tool

Shell file
Auto Script (Text Files)
DATADEF (Text Files)

From Excel Work Book

Log
Test Plans
Reports
Current
Master

To Comparator Work Book

Business tier

Business Object

SOE (Data API, security)

Access
SQL Server
Oracle

Data services tier

File utilities
Non-GUI applications
BOAutoTest run-time

Control from simple GUI or command line

<table>
<thead>
<tr>
<th>Control from simple GUI or command line</th>
</tr>
</thead>
<tbody>
<tr>
<td>run summary</td>
</tr>
<tr>
<td>display log</td>
</tr>
<tr>
<td>status</td>
</tr>
</tbody>
</table>

---

### Run Summary

- **Shell path:** D:\EuroBOAT\AutoBSHShell.txt
- **Run errors:** 6
- **Total:** 6

### Log

```plaintext
<table>
<thead>
<tr>
<th>q. Results LUPQY009</th>
</tr>
</thead>
<tbody>
<tr>
<td>q. LOOKUP_ID NAME CODE_LENGTH DESCRIPT LENGTH CODE_CODE LABEL</td>
</tr>
<tr>
<td>ADDRIGN Address Region 10 40 Address Region Address Region</td>
</tr>
<tr>
<td>. EOF is: True Record Count is: 113</td>
</tr>
<tr>
<td>. DbClose</td>
</tr>
<tr>
<td>c. ILN 389</td>
</tr>
<tr>
<td>c. CABSDBOPEN Time Recording cabs cabs</td>
</tr>
<tr>
<td>c. GetQuery LOOKUP02 SELECT * FROM LOOKUP_DATA ORDER BY LOOKUP_ID ASC</td>
</tr>
<tr>
<td>c. SQL statement SELECT * FROM LOOKUP_DATA ORDER BY LOOKUP_ID ASC, CODE_CODE</td>
</tr>
<tr>
<td>. SQL OK</td>
</tr>
<tr>
<td>c. CreateRecordSet DB Handle: 5 Recordset Handle: 1</td>
</tr>
<tr>
<td>q. Results LUPQY010</td>
</tr>
<tr>
<td>q. LOOKUP_ID CODE CODE_DESCR CODE_DELETEABLE CHNG_C1</td>
</tr>
<tr>
<td>ADDRIGN BEL Belgium -1 1 ADDRIGN CHF</td>
</tr>
<tr>
<td>STOKXCHNG FSX French Stock Exchange -1 1 STOKXCHNG</td>
</tr>
<tr>
<td>. EOF is: True Record Count is: 900</td>
</tr>
<tr>
<td>. DbClose</td>
</tr>
<tr>
<td>END SCRIPT</td>
</tr>
<tr>
<td>END LOG</td>
</tr>
<tr>
<td>. Test Log saved D:\EuroBOAT\TestLogs\demo1.txt</td>
</tr>
<tr>
<td>END LOG</td>
</tr>
<tr>
<td>c. LOG ERRORS Shell path D:\EuroBOAT\AutoBSHShell.txt</td>
</tr>
<tr>
<td>c. Total 6 Sys 6 Env 0 Scr 0</td>
</tr>
<tr>
<td>END SHELL</td>
</tr>
</tbody>
</table>
```
Features

- **Familiar spreadsheet interface**
  - single repository, script and data editor, functions, macros, “smart” comparator

- **Run time**
  - automated tasks, interactive or background, debugging, small footprint, multiple instances, low spec machine, unattended, fast, status display

- **Can be used by non-programmers**
```plaintext
AutoBOShell

LOGDATE
LOGMSG *** Regression Auto Test Shell ***
ATFILEPATH C:\Temp\BOAT_LOGIN.txt
DISPLAYLOG OFF

LOGMSG *D *** POSITIVE TESTS ***
LOGMSG *D This BO updates to the CMPLU01, CMPLU02 table
LOGMSG *D Perform initial queries LOOKUP_HEADER table
REPEATDATA Lookup_Scripts\DD_LookupQuery.txt
ATFILEPATH Lookup_Scripts\LookupQuery.txt

LOGMSG Data Definition for LookupHeader Positive Testing

<table>
<thead>
<tr>
<th>DATADEF</th>
<th>VARNAMES</th>
<th>%CODE_LABEL</th>
<th>%CODE_LENGTH</th>
</tr>
</thead>
<tbody>
<tr>
<td>POS002</td>
<td>Add 2 Lookup AAA Label</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>POS003</td>
<td>Add 3 Lookup CCC Label</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>POS004</td>
<td>Add 4 Lookup DDD Label</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>END</td>
<td>VARNAemes</td>
<td>%CODE_LABEL</td>
<td>%CODE_LENGTH</td>
</tr>
</tbody>
</table>
```
BOAutoTest outputs

All use tab delimited text files:

- Extracted test plans
- Logs
  - verbose, full trace record for comparison and problem resolution
- Comparator
  - cell level comparison, errors, warnings, dates
- Test result summaries and reports
Case Study: Benefits from Automated Testing at the Business Object Level

Alison Edye
Solution6 Pty Ltd
Level 21 Town Hall House
456 Kent St Sydney
NSW 2000, AUSTRALIA
Phone: 61 2 9278 0369 Fax: 61 2 9278 0555
alison.edye@solution6.com

Multi-tiered, object oriented architectures offer a choice of testing interfaces. The graphical user interface (GUI) is usually the most complex and challenging for automated testing, and is often too unstable for effective use of commercial automation tools early in the development cycle. This case study describes how automating tests at the business object level can provide early returns, low maintenance and effective reuse of test assets throughout the product lifecycle.
1 Introduction

Test automation offers great potential benefits but can be difficult to implement successfully at the graphical user interface because of the inherent complexity of the interface. In multi-tiered architectures it is possible to automate business functionality testing at the business tier, leaving the visual and navigational aspects for presentation tier testing. Test automation can be introduced early in the development cycle by developing object level scripts in conjunction with component development, providing early returns, greater reuse of test assets, and more time for creative GUI testing.

This case study describes our learning experiences during the development of a test automation process. It demonstrates the features of the test infrastructure and scripting approach that enabled reuse of test assets from coding and unit testing through to automated regression testing. It includes experiences with GUI and non-GUI test automation, but concentrates on the processes we developed and the benefits we gained from automating tests at the business object level.

The following sections describe

- The product under test
- Stages in the evolution of our automated test system
- Testing business objects
- Test development and process issues
- Current approach
- The test tool and scripting language (Appendix)

2 Test goals

The main testing goal was to develop test procedures that would improve quality control, defect identification and product stability. Testing was focussed on user interface and business functionality, cross platform certification and sociability testing. We needed fast verification after changes so that issues were known before product release.

We hoped to use automated testing to provide fast verification of the growing feature set on multiple platforms. Most benefits would be gained if tests the tests could be repeated frequently and the test assets could be reused in different scenarios. The tests would be most effective if they were robust early in the development cycle and were portable and maintainable.

3 The product under test

3.1 The product

The product is a client/server Practice Management System that runs on different Windows operating systems and supports different relational databases. It can be configured with a wide range of system options and provides customer modification of forms and underlying business rules.

3.2 Product architecture

The product uses Microsoft’s ActiveX/COM technology in a logical multi-tiered component based architecture. The Component Object Model (COM) is Microsoft’s standard for object interaction. The architecture includes a graphical user interface (GUI) application layer, a business layer, a system operating environment layer which handles common functions such as security, local registry and database access, and a data layer. Many of the components are re-used for interactive GUI processing, remote off-line processing and batch imports, and will be used by the Web interface. In the development process the components are generally developed independently before GUI applications are available.
3.3 Business objects

In a software or application context, a business object represents how a business modelling concept is realised in a software design or program code. Most of the product business functionality is delivered by business objects together with database tables, triggers, procedures and views.

Objects have data (state information) and behaviour (a set of routines), which can be accessed by client programs via the properties, methods and events defined in its interface. All objects are defined by a class, which is essentially a template from which the object is created. When more than one object is created from a single class, each object is referred to as an instance of the class. ActiveX components are precompiled binary code (independent of the language in which they were created) that expose one or more object interfaces for use by client programs. Some of the product components are written in C++, but most of the product is developed in Visual Basic (VB).

Business objects represent real ‘objects’ and concepts such as clients, staff members and expense items. They interact with each other by calling methods which simulate real-world processes, and should provide access to all application functionality. The use of business objects is encouraged by both COM and CORBA (Common Object Request Broker Architecture). They both advocate separation of business rules from the presentation layer in such a way that the main application logic is contained in an object model that is largely independent of the user interface.

Object models show the relationships between the objects used in a program. Objects can be accessed directly or within the object structures that provide the application’s business functionality. The objects have methods, properties and collections which together provide the structure for most of the business applications. Most people are familiar with navigating object hierarchies in products like Microsoft Excel, in which workbooks contain collections of worksheets and have properties such as Name and methods such as Save.

Navigating a business object hierarchy is similar to navigating the application’s GUI screens and controls.
4 Stages in the evolution of our automated test system

4.1 Automating testing at the product GUI

We first attempted automated testing at the user interface. Few of our QA staff and testers came from a technical background, so we chose a ‘user friendly’ commercial automated test execution tool. We purchased SQA TeamTest, used SQA Manager for defect tracking and began automated test trials with SQA Robot. The problems encountered will be familiar to most users of this type of tool: inability to recognise third party and custom controls, limitations of capture/replay, and the need for programming skills for modifying scripts. However, the deciding factor was that at this stage in the development cycle we did not have a stable GUI and the scripts required continual modification.

RESULT The scripts were not robust enough for repeated execution.
EFFECTIVE? No
DECISION Abandon tests.

4.2 Automating business object testing through a GUI test harness

4.2.1 Using the test harness manually

The business objects contain most of the business functionality and are developed before the GUI applications. A GUI test harness was developed to create an instance of an object. It provided a visual interface for viewing and setting properties, calling methods and viewing error messages, and navigating the object hierarchy. A hard coded interface was needed for each object. ‘Upstream testers’ in the development teams used the test harness for unit testing. They had to understand the business object model and hierarchy, but were able to test the object outside the development environment without programming skills.

RESULT The test harness worked but repeating tests was slow and tedious
• required a developer to modify the harness for new and changed object interfaces
EFFECTIVE? Only for limited testing
DECISION Automate tests through the harness GUI

4.2.2 Automating tests through the harness

Scripts were captured initially in SQA Robot, then modified to use common functions. The automated tests mimicked the manual process of the testers, with the GUI navigation and data and test verification points
embedded in the scripts. The results were checked to see if they matched expected outcomes, and saved as
master files for use by the Robot comparator in regression tests.

RESULT The process worked but there was more focus on GUI scripting than on object testing
- labour intensive scripting process
- slow execution time
- large test logs for GUI verification
- needed to capture different master files for different databases (Oracle and SQL server returned data in different formats)
- scripts complex and difficult for investigating defects
- high maintenance

EFFECTIVE? Partially
DECISION Extend the coverage to more business objects and try to improve the process

4.3 Bypassing the GUI

4.3.1 Providing a direct interface to the objects using simple keyword script files

Our next approach used Robot’s scripting language, SQA Basic, for direct interaction with COM objects.
The interface elements of the test harness were transferred to Robot. Most of the testers had considerable
business experience but limited programming skills, so simple keyword text files were used for specifying
the script actions and data. SQA Basic functions were written to interpret the keyword script and execute
the actions and verification points in the full Robot environment. The interpreter could be extended easily
to add new functionality, and checks for expected results were included so that tests could verify
functionality when run for the first time. Excel was used for script editing and management and SQA Shell
scripts controlled script execution.

RESULT Repeatable unit tests
- faster scripting, lower maintenance
- scripts ran ten times faster than through the test harness
- simple keyword interface for non-technical testers
- finding original defects, not just retesting
- scripts ran in the background and could be used in component debug mode
- each object needed a hard coded SQA basic interface to methods and properties
- needed separate master files for different databases

EFFECTIVE? Yes for unit testing
DECISION Convert earlier scripts, build new test suites and expand the test infrastructure

4.3.2 Expanding the test infrastructure

A common database interface was added so that the same scripts and master files could be used with
different databases. Testers did not need as much training to use the keyword language, and the scrips were
easier to code, modify and maintain. The tests scripts ran on different platforms and passed with the same
master files. We had a history of detailed test plans that became rapidly out of date and no longer reflected
the current functionality of scripts, so we added a facility to extract comments from the test scripts to
produce up-to-date test plan details. When the unit testing was complete, the scripts could be used for
regression testing. Test suite development, hand over processes, and naming conventions were put in place
and SourceSafe was used for tool and script configuration management and version control.

Each different object still needed a hard coded SQA Basic interface. In SQA Basic and VB 5 the only way
to address a method or property on an object was to hard code the method or property name into the
statement. There was no mechanism for passing the name of a method or property as an argument as you
can do with VB 6 using the ‘CallByName’ statement.

RESULT Repeatable object level regression tests
- tests on different databases could use the same scripts and master files
- familiar spreadsheet interface for script editing and management
• automatic extraction of test plans from scripts
• configuration management
• finding original defects, not just retesting
• still needed a hard coded interface for each object

EFFECTIVE? Yes for regression tests
DECISION Build more test suites

4.4 Creating a generic object interface test tool

4.4.1 Adding a generic object interface

The developers wrote an ActiveXHelper component in C++ to provide functionality similar to ‘CallByName’, and used it to replace the object specific code in the GUI test harness with a generic interface. Unfortunately we were unable use the ActiveXHelper effectively with SQA Basic because of difficulties we encountered passing objects as arguments. The Robot suppliers were very supportive, but could not solve our problems at the time. The SQA Basic code was converted to Visual Basic 5 and ran successfully as a VB application using the ActiveXHelper.

New and changed object interfaces could now be tested without any code modifications, and the new generic test tool could run the same keyword scripts as the SQA Basic interpreter. It could also access an object’s interface definition and provide the names, type and arguments for all exposed methods and properties.

RESULT Generic object interface for upstream script development and unit testing
• uses same keyword scripts
• provides an object interface definition for script writing and smoke tests
• VB tool development environment
• simple, portable, small footprint, simple installation, no licence fees
• no comparator for regression testing

EFFECTIVE? Yes for script development and unit testing, not yet for regression tests
DECISION Use the generic interface tool for script development and unit testing, and Robot for regression tests

4.4.2 Expanding the capability of the generic tool

A simple but effective comparator was developed in Excel so that an output log could be saved as master files and compared with the current log. Initially, when the outcomes from test scripts did not match the expected outcomes, the developers suspected the test tool rather than their code. Sometimes the test had to be reproduced in a small VB test application without using the generic interface to establish its validity. As the process matured the tool and the testers gained more respect from developers. Although the initial tool development was uncontrolled, we now needed to apply appropriate quality control. When changes were made, the tool itself had to undergo rigid testing for new functionality and backwards compatibility, and the product regression scripts were used to regression test the tool.

File handling capabilities and an application command line interface were added for handling imports, offline transactions and existing special test harnesses.

The most powerful addition was to add variables, repetition and data definition files. This enabled scripts to be reused in different sequences, with different data sets and in different business scenarios. The data file could include comments, expected outcomes and different scripts in each row, and tests could be created quickly and modified easily. The data files could be created in the same workbook as the scripts.

RESULT Reusable test assets, regression capability
• data abstracted from the detailed script logic
• can create a library of reusable scripts for reuse with different data sets
• easy for non technical people to design and create test data once the library scripts were available
• flexible data definition format for variables, including script file names and expected results
• Excel functions can help to automate data generation and expected outcomes
• test analysts can concentrate on creative test data design
• much faster test development and maintenance
• reduce the scope of the manual test requirements to something more attainable

EFFECTIVE? Yes for unit tests and individual regression tests
DECISION Expand use of generic tool, keep using Robot regression test

4.5 Another look at GUI testing

We developed a suite of simple GUI level smoke tests using Robot to test basic user interface functionality. We conducted a pilot study using the same keyword approach to simplify GUI level scripting and eventually produced a working prototype. The product’s form customisation technology generates controls at run time from definitions held in the database, and forms can differ for different users and morph as a result of entered data. This is a really challenging environment for effective use of GUI test tools. We used run time form and control mapping to make the scripts more robust, but the whole scripting environment was much more complex and sensitive to change than the object level test environment.

RESULT Prototype data driven keyword GUI script engine with run time form and control mapping
• needed a GUI test tool expert with technical product knowledge to develop and maintain the keyword script interpreter and functions
• test environment was more complex and sensitive than the object level

EFFECTIVE? May be, if given sufficient resources
DECISION Wait until perceived benefits justified more resources

4.6 Enhancing regression capability

More regression capability was added to the object testing tool to provide for unattended running of sequences of test suites and automatic result summaries. When the generic VB test tool infrastructure matured Robot object level regression tests were discontinued because of the unnecessary duplication of effort.

RESULT Unattended execution of multiple regression suites
• production of summary reports from the run execution and the comparator
• logs sufficiently detailed to identify the cause of most problems without needing to re-run the script
• tests suites configured so that they can be run independently using the same master files for comparison as in the full regression test

EFFECTIVE? Yes
DECISION Develop procedures for adding automated testing into the build cycle

4.7 Automated system tests and database generation

The existing tests were used for unit testing and regression testing of business objects. They used valid and invalid data, some of which was held in a test database. Tests accessed a dedicated database that was automatically refreshed during test runs. Most testing was performed on a test data that had been created by direct entry to the database, not by the product itself. The data did not have integrity, was hard to maintain to match increased product functionality, and it was difficult to obtain meaningful results from audits after test runs. Verifying the database updates manually was difficult because of the complexity of the business rules. We needed a clean database so that we test and use the audit applications to verify the data.

We embarked on a project to create a fully automated system test that would start with an empty database and enter a comprehensive set of data using realistic business scenarios over a number of periods, and run audits to verify the results. Experienced staff prepared comprehensive test plans, but the implementation
was scheduled as part of the initial training for industrial trainees and new graduates. Training to use a GUI test tool was beyond the scope of the project, so a core team of three was set up to use object level scripting. The scripts were based on existing scripts with new data, but were integrated into a system test structure. The team successfully created the basis for our current object level system test and database build capability, finding many defects in the process. Since then we have frequently used new staff to help extend and maintain the system tests, and they have found it a challenging but valuable learning environment. Several of them had no previous programming or scripting experience. The team needed a lot of support to overcome technical and process difficulties and chase defect fixes. Business experience is needed to verify the results.

RESULT Automated object level system test and database generation

- can set up test data and environments saving weeks of manual test effort
- can verify data and component functionality by running system audits and integrity checks

EFFECTIVE? Yes for unit tests, individual regression tests, object level system test, test database generation

DECISION Use in parallel with Robot regression tests

4.8 Integrating automated testing into the development process

Regression tests are not effective unless they are robust, up to date and have adequate coverage. When a new build passes the regression tests, there can be a false sense of security and an assumption that there are no defects. The regression tests are only repeating what has previously been tested, and may have always had poor coverage or lost coverage as the component has been enhanced without updating the scripts. People will not use the scripts if they are not fast, easy to run and reliable, so resources must be allocated and processes introduced to ensure that they are kept alive.

We introduced tight configuration management for the scripts and logs, and used NuMega TrueCoverage to show which lines of code were and being executed by the scripts. The test analysts and developers reviewed the results and enhanced the scripts accordingly. In our current process, when a developer fixes a defect, the developer runs the corresponding scripts and ensures that they pass the regression test. The developer also checks the script coverage, and enhances the script if necessary. The scripts and component are subject to similar checks and review before being checked back into SourceSafe and delivered to the build with the new version of the component. This can take more time up front but it saves time overall.

This process frequently identifies defects that can be fixed on the spot and avoid the costly process of defect raising, tracking, resolution and independent verification that is necessary if the defect escapes into the build. It prevents defects from entering the build and the formal regression tests in the build cycle now rarely identify new defects. The developers accept a greater ownership of quality, and are more quality aware when they know that their fixes are subject to review and regression testing. Most initial scripting for unit testing is still done by testers working with developers in conjunction with new component development. Some developers write their own scripts to facilitate their environment set up and debugging.

RESULT Extended ownership of automated tests and quality

- scripts kept current and useful
- more defects detected before they reach the build

EFFECTIVE? Yes

DECISION Continue to improve processes and test coverage

4.9 Regular build cycle

The object level regression tests are an integral part of our regular build cycle. We normally schedule one or two weekly builds depending on the rate of change but can do more if required.

We have a mid-morning cut off for the database components so that they can be checked before they go into the build. Object level regression scripts are run against the different databases to confirm that all components are present and that only known changes are present in the structure and deliverable contents. This is followed by the main component cut off and the build. A suite of GUI level smoke tests is run to
confirm that each application will load and perform basic add, modify and delete functions before releasing the build to the floor in the mid afternoon.

The remaining regression tests and system tests are set up in the test laboratory on a variety of test platforms to run overnight and are checked the following morning before the results are posted on the network. The databases generated by the system tests can be automatically saved and restored during test runs, and current databases are made available to the floor after the build.

It would not be feasible to test all these functions manually in a frequent build cycle.

For example, one of the main object regression runs executes 42 suites testing 16 business objects. It executes approximately 150,000 lines of product code (11,000 functions) in under three hours, and takes one person a total of 2 hours to set up on multiple environments and verify the results. The system test and database generation can run in two hours on a fast machine.

RESULT Fast verification of business functionality after builds
- up-to-date test databases available
- increased confidence and stability
- manual testers can focus on GUI testing

EFFECTIVE? Yes
DECISION Use regularly

5 Object level testing

5.1 What business objects do

Typically, business objects perform the same functions as the GUI application level for a particular subsystem, such as Billing. A business object accepts input, validates the input values according to a set of business rules, and returns validation errors and exception. It processes database queries and updates and returns data to be displayed on a form or formatted for a report. It calls other objects and builds collections of objects that usually mimic the structure of the data displayed at the GUI. Some business objects create files and communicate with external applications such as mail and Office.

5.2 How should they be tested?

Objects can be tested by calling methods, checking the returned values and error conditions, and getting and setting property values. The detail design documents should include use case scenarios that describe the range of normal values and the normal sequences of operations on the object. Core components from the System Operating Environment generally require knowledge of the internal code to ensure that critical paths are adequately tested.

Business object tests need to check that the business rules are correctly applied, and that the correct values and error messages are returned. They should query the database to check updates and triggers. In other words, the tests use a different interface but should check the same business functionality would through GUI level testing. Negative test scenarios should be included to make sure that the component will not crash under situations which may occur in a live environment. Large scale testing for scalability, performance and reliability is outside the scope of this case study.

The basis for object testing methodology is the need for a precise definition of its interface. The interface specification should be included detailed design documents, but the interface is not always implemented as specified. A useful first test for an object is to display its methods and properties together with arguments and types. Following that the methods can be called with varying arguments and sequences to see whether they perform as expected. When a component depends on the services of other components, it may need to be tested together with that component or with a stub that returns the expected results. A components tested with stubs may not encounter the kind of conditions that exist in an actual system environment, and may not exhibit incorrect behaviour that would show up in a live environment. The components may be able to be tested in a sequence such that they use the services of other components that have already been tested, or alternatively all components in a system may need to be tested together. Because of the large number of
possible paths that could be taken to reach a low level element, it is not realistic to test all possible paths in
the system.

5.3 Object level test approach

Our approach is to try to make the tests comprehensive, but selective. We do not attempt to test all paths
through all objects, but try to select test cases that return high value. Like product components, the tests
should be well factored – they should cover a range of situations once, and not the same one over and over.
Coverage can be checked manually by code reviews or automatically using code coverage measuring tools,
but needs to be done by experienced test analysts or developers.

We test in a system environment using business scenarios and use cases. We can use the same test
scenarios and data to test at the GUI level. Object level scripters need to have an accurate knowledge of the
object interfaces, the object hierarchy and the valid calling sequences. This should be clearly defined for
the application development, but isn’t always. In a rapid development environment testers need to work
closely with developers.

5.4 Object level test tools

5.4.1 Features

The basic requirement for an object level test tool is to be able to call objects directly. It is desirable to have
a generic interface (ability to pass method and property names as arguments) so that programming is not
required for each object. The tool also needs at least some of the features that are found in GUI level
automation tools (eg features for test creation, management and control; test verification points and logs,
comparator). There is no need to develop the test infrastructure if a suitable commercial tool can meet the
requirements.

The features of the test tool that contributed to simpler scripting and effective reuse of test assets were
included in the previous section, and are restated below:

- Provides general test management and control
- Provides script editing and management
- Uses simple keyword scripts – easy to learn and maintain
- Uses variables, data files and repetition
- Provides automatic extraction of test plans
- Scripts can be reused with different data
- Creates instances of objects
- Calls methods and properties
- Allows navigation of the object hierarchy
- Performs direct database queries with common results for different databases
- Includes run-time checks for expected results and values
- Provides test case comparisons and common master files for different platforms
- Provides management and control for unattended testing of sequences of test suites
- Writes detailed logs and summary reports
- Provides interfaces to other applications, test harnesses and file handling utilities

More information about our test tool is included in the appendix for those who are interested in different
approaches to scripting and data abstraction.

5.5 Skills required for object level scripting

Testers need a good understanding of the business process, business object model and hierarchy, and object
interfaces. They do not have to handle unexpected GUI events or complex controls, and in our experience
have found object level scripting using keywords easier to learn than GUI testing. Many of our scripters
have had no previous programming experience. Complex objects may need a more technical person to
write the scripts, but a business oriented tester can develop the test plans and test data to drive them. When
using data driven keyword scripting at the object level there is no need to have GUI tool experts for script
development or execution. Testers can focus on the test cases, not the complexities of the test tool.
Scripting is a development process and requires the same management, support and code review processes that are used with developers to ensure that they deliver good quality test assets.

5.6 **Hints for writing object level test scripts**
- Check for changes in object interfaces.
- Check object hierarchy navigation.
- Check for appropriate clean-up, reuse and object release. Test for memory leakage.
- Test for appropriate error handling. Components should raise errors and not display message boxes.
- Plan for iterative development. Start early and develop tests in conjunction with the component.
- Include sufficient database queries and object state values in the output to be able to confirm test results.
- Where appropriate, check critical expected values within the script, including returns from method calls, exposed property values and critical database updates. These tests can be used to verify functionality and find defects. Checking the outcomes against expected values within the script is more valuable than relying on comparisons with master files. The run-time checks put the intellectual capital within the script, show errors immediately and don’t rely on careful checking when replacing master files.
- Design test suites to run independently, or sequentially in an unattended test environment using the same master files.
- Create modular test suites so that scripts can be reused.
- Create script libraries and use data files to increase useability of test assets.

6 **Current approach**

6.1 **Test components and business functionality at the object level**
We now have a situation where the same test environment and scripts can be used for the development stage for debugging and upstream unit testing, regression testing, object level system testing and building test databases. It is used for testing Active-X components and database components, and can use the same master logs for different operating systems and database platforms. We run an extensive object level system test after each build which builds a test database, runs audit tests and provides an up to date database for testing and training. It can also be used to build demonstration databases.

We have found that using this approach we can start automatic testing very early in the product lifecycle and reuse the development test assets for regression testing. The same skills and script assets can be used for fully automated object level tests, or for setting up the environment and test database for manual tests. These assets contribute to most of the test activities on the right side of the Extended/Modified V-Model, which shows the test-related activities that need to be performed once the code is completed. In fact, we can start using the tests while the code is under development to set up a test environment, identify problems and assist in debugging. The Extended/Modified V-Model diagram is taken from Developing a ‘Testing Maturity Model: Part I’ (see references). Figure 3 highlights the activities which include reuse of object level scripting test assets such as test plans, scripts and skills.

6.2 **Use GUI level testing to verify UI navigation and functionality**
We use automated GUI level tests for application smoke tests and some sociability tests. These tests are simpler, faster and easier to maintain then full GUI level business functionality tests.

6.3 **Use manual testing for complex, creative and demanding tests**
Some tests are difficult to automate and better suited to manual testing. Manual testers can test more complex scenarios and be more original and creative in their responses, and usually find more defects than re running automated tests. If the easier repetitive testing is automated there is more time for more creative and demanding manual testing.
7 Test development and process issues

We found that the following issues are important when developing business level tests:

- The specifications for components and object interfaces must be concise and up-to-date for planning and scripting object level tests. When testers are developing tests as the components are being developed, the developers need to realize the importance of keeping the documentation up-to-date.

- If poorly designed interfaces include business functionality in the application, the application becomes ‘fat’ and scripting business scenarios at the object level is more difficult. Testers may need programming skills to emulate the application, and time is wasted reproducing functionality which is not being tested. ‘Fat’ applications should be discouraged by design reviews before they are coded.

- Some technical components may be complex and require stringent testing. These may need a special harness, or a technical tester to write the scripts. When extra functionality is required, it may be able to be added to the test tool and made available for scripting other objects.

- One of the dangers with regression tests is that people think that if the component passes the tests then it has no defects. A ‘false positive’ gives a false sense of security. Test designs, scripts and master files need to be reviewed to ensure that the coverage is adequate and that the masters are correct. The database needs to be queried and the data carefully checked for accuracy of content. Testers need to be aware of the actions of triggers and make sure they check all the necessary tables. Audits can be very useful for checking that business transactions have performed correctly. It is necessary to have a comprehensive, up-to-date test database for adequate data verification.
If a defect is raised from component level testing and cannot be reproduced at the application level it may be treated as a low severity defect. This needs careful investigation to check whether there may be situations where it could occur in a live environment, and a decision has to be made whether to change the component or the test script. The defect may need to be given a high priority if it affects the regression capability.

Those with computing qualifications and experience were frequently able to produce effective scripts and contribute greatly without close supervision. However, when we neglected to provide adequate training and management for the less experienced staff they were much less productive. Scripting at this level is a development task, and needs committed management, well designed test plans and procedures and frequent code reviews to achieve effective results.

8 Lessons learned

- The main lesson we have learned is that it takes time, resources and commitment to develop a mature automated test infrastructure.
- The test process needs to be as simple as possible so that testers can concentrate on testing the product rather than the test scripts
- Use appropriately skilled people and manage script development with the same processes used for product development
- Provide clear component interface specifications
- Review test designs for coverage, robustness and reusability
- Build and verify comprehensive test databases
- Extend ownership of automated tests to development and support areas
- Allocate resources to keep tests ‘living’

9 Conclusion

In our development environment we have gained the following benefits from implementing an automated object level test infrastructure.

- Automated testing can start early in the development cycle
- The object level interface is less complex than the user interface and it is easier to build robust automated tests at this level
- The simpler interface allows testers to focus on the test cases rather than the tool
- The maintenance is low, so there is time available for increasing test coverage
- The automated system provides fast verification of product changes and frequent test iterations
- Defects can be identified by developers before they are put into the build, saving the time and overheads associated with defect tracking and repair
- The automated tests cover basic functionality and allow testers more time for manual testing
- Manual testing can concentrate on user interface issues and more demanding test scenarios
10 Appendix  Test Tool and Scripting Language

The tool and scripting language were not designed in advance but evolved to meet demands. They are simplistic but work reliably in our environment and are enhanced easily to meet new requirements. The features are included here for those who are interested in different approaches to scripting and data abstraction.

10.1 Architecture

BOAutoTest is a VB executable but the basic functionality was originally written in SQA Basic and could be implemented by most commercial test tools.

BOAutoTest replaces the presentation tier with a direct interface to the business tier. It can also call applications via a command line interface and wait until they complete execution, and includes file utilities. It uses a generic object interface (similar to VB 6 CallByName).

BOAutoTest is loosely coupled to an Excel script editor and repository and an Excel Comparator. All input and output files are saved as tab delimited text files independently from the Excel files, and BOAutoTest reads and write only text files. BOAutoTest can be started with an Excel, or run independently. When Excel calls BOAutoTest, it passes the name and path of the Shell file. When a script requests a master file comparison, BOAutoTest copies the specified current and master files into a comparator directory and starts the Excel Comparator. The Comparator can be opened manually at any time to compare the files in the comparator directory. BOAutoTest can call other applications, including other instances of BOAutoTest, and these can be called concurrently or sequentially. This provides a mechanism for concatenating regression suites for unattended operation, and for controlling basic load generation and scalability tests.

Figure A1  Test Tool Architecture
10.2 Familiar spreadsheet interface

Excel provides a familiar interface for script editing and test case comparison with a powerful function and macro capability. The interface is not intimidating for non-programmers, and the calculating ability can be used to help format and create input data, such as expected outcomes. The workbook structure is convenient as a single repository for holding associated scripts, data and documentation, but they do not need to be in the same workbook. VBA macros are used to save text files, for occasional pre-processing of files, for starting BOAutoTest and for the comparator.

10.3 BOAutoTest comparator

The comparator is simple but functional in the BOAutoTest environment. It compares complete output logs - suites can be divided into modules that each have their own log. The comparator is designed specifically for BOAutoTest log formats that identify log messages, script modules and lines which may be changeable. It reads the current and master logs and does a line-by-line, cell-by-cell content comparison, identifying differences with coloured cells and keeping a result tally. It has sufficient re-synchronising capability to handle normal variations such as extra or missing lines. If two corresponding lines are different and are marked to indicate that may change, the difference is identified and counted as a warning. If two different cells have an Excel date type, the difference is identified and counted as a date. On completion, the results are displayed at the top of the page, showing a green pass if there are no errors. In an unattended run the comparator writes a summary to the result log and exits. Macros provide synchronised navigation to highlighted errors, and the log contains sufficient information to identify the likely cause of the problem.

Comparators do not identify defects, they pick up differences between two files. If the master file contains errors, then these errors will not be identified in the current log. The integrity of the test is dependent on the how much care is taken to verify the accuracy of the results before master files are saved. Comparing a complete log is easier to manage than having many independent test case master files, but is more vulnerable to corruption of master files by careless recapturing. Scripts should check that values match expected outcomes while the script is executing and record an error in the log so that the intellectual capital is held in the script, not just the master file. Where this is not appropriate, the expected outcomes can be included as comments so that they can be checked before the master file is replaced.

10.4 Script syntax

The script syntax includes keywords, variables and arguments separated by tabs. Lines which are blank in column 1 are ignored. Variables are identified by leading ‘%%’, can be used anywhere, and must be defined before use. The script interpreter parses the lines one at a time for processing, and substitutes variable values for names where appropriate.

Execution is determined by keywords, which can occur in any column. The keyword in the leading column determines the interpretation of the values in the following columns. The remaining columns may contain further keywords, keyword argument values or variables.

The main script is called a shell script. It uses the same syntax and shares some of the same keywords as the auto test scripts, but includes extra keywords for controlling the test execution.

Script execution and iteration is controlled by the shell scripts. Auto test scripts may be repeated with the same data, or use variable data defined in data definition files. Data definition files have the same syntax as script files, but use only the log message and data definition keywords. Variable names are defined at the top of each column. Each row contains a comment and the variable values. They can hold any variables, including auto test script names to control script execution sequence. This allows different scripts to be used in different rows of data and provides a simple definition of flexible business scenarios. The variables can hold input data, and expected results and expected values.

There are no explicit conditional statements. Generally, scripts predict expected results and raise errors when they differ from expectation rather than branching execution - comparators do not handle unexpected branching at all well. To provide flexibility for data input to reusable auto scripts, most data keywords define a missing argument as a skip instruction, and will record this in the log without executing.
Scripts use meaningful file names, variable names and log message comments to enhance readability of scripts and logs. Log messages may include a test plan descriptor identified by ‘*’ in the first character (eg ‘*R’ Expected result) which is used for formatting automatically extracted test plans.

10.5 Running BOAutoTest

BOAutoTest can be started automatically from a command line, an Excel macro or by double clicking. It has a simple GUI interface that provides for interactive control and displays the execution status and optional display log. BOAutoTest will run shell scripts, extract test plans, open the current log, exit or abort. Errors are shown in the log by type (eg result error) and the number of errors in each category is displayed in the status area. The total number of errors is recorded at the end of each log and in a summary report in regression runs. If the script includes a log comparison, the comparator is called automatically.

BOAutoTest can run unattended on a low specification machine, and tests are often left to run overnight.

10.6 Configuration management

Automated test runs are subdivided into logical test suites which contain all the relevant resources for running the suite independently. Each suite is held in a directory and subdirectories with a defined structure and naming convention. The directory holds the Excel workbooks, and directories for scripts, log files and masters. The whole configuration is maintained in SourceSafe under version control and can be installed on any machine with a single ‘get’. New masters are checked in after authorised changes, and the history of masters can be used to back track to earlier test results if necessary.

---

**Figure A2  Shell script extract** (no variables are used in this sample)

AutoBOShell

LOGDATE
LOGMSG *** Regression Auto Test Shell ***
ATFILEPATH C:\Temp\BOAT_LOGIN.txt
DISPLAYLOG OFF
LOGMSG *D *** POSITIVE TESTS ***
LOGMSG *D This BO updates to the CMPLU01, CMPLU02 table
LOGMSG *D Perform initial queries LOOKUP_HEADER table
REPEATDATA Lookup_Scripts\DD_LookupQuery.txt
ATFILEPATH Lookup_Scripts\LookupQuery.txt

---

**Figure A3  Data definition file extract** (truncated)

LOGMSG Data definition for Error Validation Testing

DATADEF VARNAMES                %CODE_LABEL %CODE_LENGTH
LOGMSG *T NEG001 - Add a duplicate LookupHeader
DATADEF Error Messages: (LOOK4) AAA Label 10
LOGMSG *T NEG001 - Add a record with no details
DATADEF Error Messages: (LOOK31, LOOK5, LOOK6)
LOGMSG *T NEG001 - Add a record with invalid details
DATADEF Error Messages: (LOOK28, LOOK7) dfdsfd 400
END
Figure A4  Auto script extract (no variables are used in this sample)

<table>
<thead>
<tr>
<th>Column 1</th>
<th>Column 2</th>
<th>Column 3</th>
<th>Column 4</th>
<th>Column 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>BONAME</td>
<td>CABSBilling.Bill</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOGMSG</td>
<td>*N Billing BO</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOGMSG</td>
<td>*D Billing Functionality raise New Fee Bill with 2 engagements</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOGMSG</td>
<td>*D Add Client details (FRD010)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PARDEF</td>
<td>sClientNo FRD010</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PARDEF</td>
<td>sBType FEE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>METHOD</td>
<td>ADDNEW -1 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOGMSG</td>
<td>*D Add Engagement details for engagement FRD010010 and FRD010030</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOGMSG</td>
<td>*T ADDTAXBILLENGAGEMENT (Add 2 engagements)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOGMSG</td>
<td>*R No errors expected</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SETPROP</td>
<td>BillDate 2002/6/1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PARDEF</td>
<td>sEngNo FRD010010</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PARDEF</td>
<td>sLocTax ~</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PARDEF</td>
<td>sNatTax T2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>METHOD</td>
<td>ADDTAXBILLENGAGEMENT -1 0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure A5  BOAutoTest run time

```
# BOAT 2.01.4

Shell path: D:\EuroBOAT\AutoBOShell.txt
Run errors: 6  Total: 6

Results LUPQY009
- EOF is True  Record Count is: 113
  DbClose
- ILN389
  CABSDBOPEN Time Recording cabs cabs:
  GetQuery LUPQY002 SELECT * FROM LUPQY_DATA ORDER BY LUPQY_ID ASC,
  SQL OK
  CreateRecordSet DB Handle: 5 Recordset Handle: 1
- Results LUPQY010
  LUPQY_ID CODE CODE_DESCR CODE_DELETEABLE CHNG CD
  ADDRGN BEL Belgium -1 1 ADDRGN CHC
  STCKXCHG FSX French Stock Exchange 1 0 STCKXCHG
  EOF is True  Record Count is: 900
  DbClose
  END SCRIPT
END LOG
- Test Log saved D:\EuroBOAT\TestLogs\Demo1.txt
END LOG
- LOG ERRORS Shell path D:\EuroBOAT\AutoBOShell.txt
  Total: 6  Sys: 6  Env: 0  Scr: 0

END SHELL
```
References


BURNSTEIN, I., SUWANNASART T. and CARLSON C. R. Developing a Testing Maturity Model: Part II”


CASANAVE, C. Business-Object Architectures and Standards, Data Access Corporation

Benefits From Automated Testing at the Business Object Level – Case Study
Alison Edye

Alison Edye works in the architecture team of a large international software company that delivers e-Commerce, Professional Services and Business Software. She is the main architect of the automated testing technology used with their Practice Management product, and researches and develops appropriate testing approaches for new projects and technologies.
Alison graduated in Computer Science in the late sixties and worked for nearly ten years in the telecommunications industry implementing distributed real time communications systems in Australia and overseas before transferring to mainframe computing as an IT manager. She lectured at university in Computer Science and taught high school Computing Studies before returning to the software development industry to work in a Microsoft development environment.