Agile Regression Testing Using Record & Playback

Gerard Meszaros
ClearStream Consulting
87 Connaught Dr NW
Calgary, AB
T2K 1V9 Canada
1-403-210-2967
gerard.meszaros@acm.org

ABSTRACT
There are times when it is not practical to hand-script automated tests for an existing system before one starts to modify it (whether to refactor it to permit automated testing or to add new functionality). In these circumstances, the use of "record & playback" testing may be a viable alternative to handwriting all the tests.

This paper describes experiences using this approach and summarizes key learnings applicable to other projects.

Categories and Subject Descriptors
D.2.5 [**Testing and Debugging**]: Testing Tools – *record & playback; robot user.*

General Terms
Verification.

Keywords
Automated Testing, Acceptance Test, Functional Test, JUnit, Patterns, Best Practices, User Interface, Robot User, Record, Playback, XML

1 INTRODUCTION

The Business Problem
Many of us "inherit" code from the original developers and need to fix or enhance it. This may be for the purpose of minor changes (the so-called "maintenance phase" of a system's lifecycle), or it may be to re-engineer the system in a large-scale way.

In either case, it is highly desirable to discover that your changes have had unexpected consequences (e.g., 'bugs") long before you deliver the code to your customers. During maintenance, there is a large potential for such defects to sneak in, especially when the maintenance developers are not intimately familiar with the code. And the less well structured the code, the higher the likelihood of introducing more defects.

OOPSLA 2003, Oct 26-30, 2003, Anaheim, California.
ACM 1-58113-751-6/03/0010

Regression Testing
The act of testing the software to ensure that it hasn't changed is called "regression testing". In most cases, it is prohibitively expensive to do proper manual regression testing of the software after every bug fix. As a result, many maintenance teams play a continuous game of *Russian Roulette* by delivering the software after running only a subset of the complete test suite.

Automated Regression Testing
Automated regression testing is the most cost-effective way of doing a full regression test run after each bug fix.

Many agile development methods advocate "test driven development" [1] (or "test first") in which you write automated unit tests before the code it tests. This results in having a safety net of automated tests available to regression test your application during maintenance and extensions. But what if your application wasn't built "test first"?

XP (eXtreme Programming) [2] advocates writing a new automated test to expose the bug before fixing it. This test becomes part of the regression test suite to ensure that the bug never comes back. After all, having the test would have prevented it!

Why Not Use XUnit?
XUnit purists would propose writing XUnit [10] tests to verify the system functionality before it is refactored or modified. But it is very difficult to write cost-effective XUnit tests to verify the system functionality if the system wasn't designed with testability in mind. Most systems need to be refactored for testability before XUnit tests can be written. But how can you ensure that the refactoring hasn't introduced bugs? This "Catch-22" was the motivation behind trying to use *Robot User* testing on several recent projects.

2 R&PB TEST AUTOMATION ISSUES
The "robot user" approach to test automation predates

XUnit-style testing by many decades. Test automation folklore is rich with horror stories of failed attempts to automate testing using "record & playback (R&PB).

The "robot user" approach to test automation had received enough bad publicity in past attempts at test automation that we found it to be a hard sell when we proposed it on a recent project. We had to convince our sponsors that "this time it would be different" because we understood the limitations of the approach and that we had a way to avoid the pitfalls.

The "Fragile Test" Problem

Tests automated using the "robot user" approach often fail for seemingly trivial reasons. It is important to understand the limitations of this approach to testing to avoid falling victim to the common pitfalls. These include *Behavior Sensitivity, Interface Sensitivity, Data Sensitivity* and *Context Sensitivity*. Many of these issues also apply to XUnit-based test automation (see [8].)

Behavior Sensitivity

Tests are intended to verify the behavior of the system. So if the behavior of the system is changed (e.g., the requirements are changed and the system is modified to meet the new requirements), we would naturally expect any tests that exercise the modified functionality to fail when they are replayed. One would hope that a small change in requirements would lead to a small number of tests failing. The problem is when other tests also fail because of the changes. This is typically because tests must get the system into a known starting state and this may require using the modified functionality.

Interface Sensitivity

Commercial "robot user" test tools typically interact with the system via the user interface. Seemingly minor changes to the interface can cause tests to fail even though a human user would say the test should still pass. This is in large part what gave test automation tools a bad name in the past. This has also been an area of significant technological improvement in the commercial testing tools in recent years.

Data Sensitivity

All tests assume some starting point; these are often called the "pre-conditions" or "before picture" of the test. In information systems, this is defined in terms of data that is already in the system. If the starting point (i.e., the database contents) changes, the tests may fail unless great effort has been expended to make the tests insensitive to the data being used.

Context Sensitivity

The behavior of the system may be affected by the state of things outside the system. This could include the states of devices (e.g., printers, servers) other applications, or even the system clock (i.e., the time and/or date of test.) In some ways, this can be viewed as a form of *Behavior Sensitivity* except that the cause is not a change in the program logic but rather a change in the context in which the system is embedded. While the symptoms may be the same, the cause is completely different and needs to be addressed in different ways.

3 TEST AUTOMATION CHOICES

As part of our analysis of the choices available to us, we came up with a way of classifying the approaches to test automation. This helped us better understand why certain approaches worked better in some circumstances than others.

There is more than one way to automate tests. The approaches can be classified using a 3 dimensional grid. The three dimensions are:

- Granularity of the system under test (SUT). The SUT can be a single unit (module, class or even method), a component, or the entire system.
- Test Creation Approach. The two main options are "Record & Playback" (R&PB) and hand-scripted tests.
- Test Interface. The two main options are testing via the user interface or testing via an internal software interface or API.

In theory, there are 2x2x3 possible combinations but it is possible to understand the primary differences between

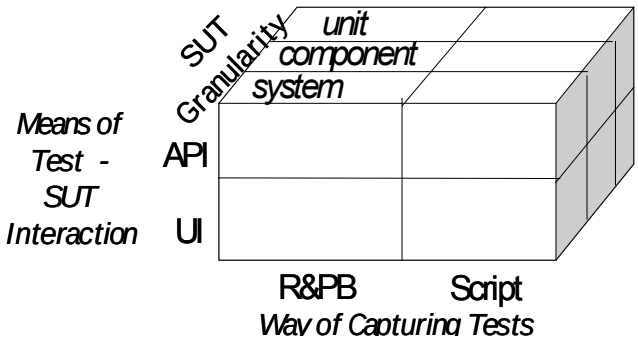

Figure 1. The 3 dimensions of test automation

the approaches by looking at the front face of the cube. Some of the four quadrants are applicable to all levels of granularity while others are primarily used for system testing.

Upper Right Quadrant—Modern XUnit

The upper right quadrant of the front face of the cube is "modern XUnit". It involves hand-scripting tests that exercise the system at all 3 levels of granularity (system, component or unit) via internal interfaces. A good example of this is unit tests automated using JUnit [6].

Bottom Right Quadrant—Scripted UI Tests
A variation on "modern XUnit" is "Scripted UI Tests" with the most common examples being the use of HttpUnit[5], JfcUnit or similar tools to hand-script tests using the user interface. (It is also possible to hand-script tests using commercial "Robot User" tools.) These approaches would all fit into the bottom right quadrant. Where the entire system is being tested, this would be at the system test level of granularity. They could also be used to test just the user interface component of the system (or possibly even some UI units such as custom widgets) but this would require stubbing out the actual system behind the UI.

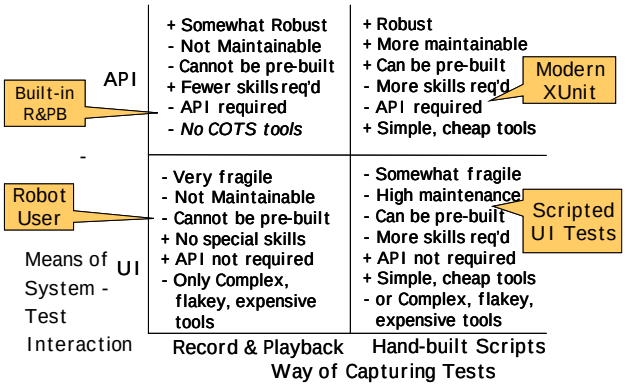

Figure 2. The 4 quadrants of test automation

Bottom Left Quadrant – Robot User
The bottom left quadrant is "Robot User". During "recording", the user interacts with the system manually via the User Interface (UI) while the tool records the interactions. During "playback", the tool interacts with the system via the UI to "replay" the original session. This is the approach employed by most commercial test automation tools. This approach is primarily focused on testing the entire system, but like "scripted UI Tests", could be applied to the UI components or units if the rest of the system can be stubbed out.

Top Left Quadrant – Internal R&PB
The top left quadrant involves creating a record & playback API somewhere behind the user interface. This is then used to record everything that affects the system state into a file that can later be used for input. This quadrant is not well populated with commercial tools but is a feasible option when building R&PB into the application itself.

4 CASE STUDIES
The conclusions of this paper are based on using "record and playback" testing on a series of projects. While all used R&PB testing as a key part of their test strategy, they used several different approaches to implement the R&PB testing.

Case 1 – Project "Billy"
This project involved the construction of a prototype for a billing system. The project team was whoever was available at the time (sitting "on the bench".) There were a number of architectural objectives that needed to be demonstrated. The application framework was built "test first" complete with both unit tests for each class and functional tests that verified the system functionality exposed via an application façade. The development approach was eXtreme Programming with some minor local adaptations.

At one point in the project, a potential customer asked to see a demo. As it is very difficult to demonstrate the architecture of a system, it was decided that a front end (user interface) was needed to show off some of the features of the architecture.

The web-based demo UI was built in a hurry. It used a *Transform View Architecture* [4] with a simple JSP front page from which all the functionality was accessed. All other screens were generated from XML using XSLT under the control of a generic servlet that looked up the URL, found the application method to invoke, invoked it to get the results as XML, and transformed the results into HTML with the XSLT corresponding to the URL. There were a few unit tests for a few of the classes but no tests that verified that the UI as a whole was functioning properly.

After the demo, there was some changeover in the project staff. The incoming developers had trouble understanding the structure of the UI software and decided to clean it up. But retesting the UI after each structural change was cumbersome and time consuming. The team, seeking to work smarter rather than harder, looked for a way to automate the retest.

Our first attempt was to use HttpUnit to script test for the web server. Submitting the URL's was pretty straightforward but interpreting the HTML that came back was challenging. The tests had to be coded to look for certain structures on the web page and examine certain table cell values to verify correctness of the results. It became clear very quickly that this approach would take much too long to do thorough regression testing.

Test Architecture
Since the refactored software was intended to produce exactly the same HTML from the input XML, we surmised that it should be possible to record the sequence of URLs posted and the corresponding HTML received in response so that we could quickly replay a previously recorded session. Because this was an in-house project being done on "bench time", we didn't explore the use of commercial "robot tester" tools. Instead, we found a few strategic places in the generic servlet where we could write the incoming URL, the XML returned by the

system, and the HTML produced by the XSLT transform into an XML file.

For playback we used HttpUnit to emulate the browser. It would read one <interaction> element from the playback file and submit the contents of the <request> element to the web server. It then took the HTML it got back and to compared it with the contents of the <expected-html> element from the recorded session. If they matched, it would read the next <interaction> element from the file. When it reached end of file without any comparisons failing, the test passed.

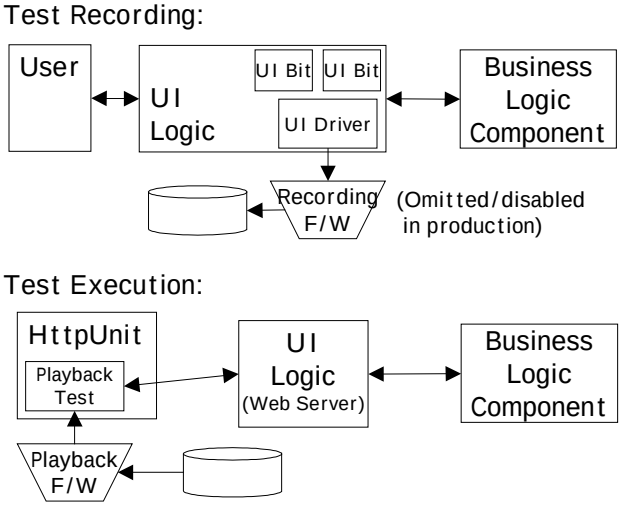

Figure 3. Test Architecture for Project Billy

Because we expected the refactored code to produce exactly the same HTML, we could use JUnit's assertEquals(String expected, String actual) to do the comparison. However, whenever we had a failure, we had to scroll through the expected and actual strings manually in side-by-side object inspectors to find out what had changed. So we wrote a *custom assertion* [8], assertEqualsWithCursor(…), that would determine the reason for failure and report on what was different as part of the JUnit error log.

Why It Works

This approach worked very well because our billing system ran entirely in memory. All the data for each run was loaded from files as during the demo. This avoided the *data sensitivity* problem. To avoid *context sensitivity,* the system clock used by the application was controlled from the demo dashboard (which was part of what was recorded) so it was always the same and we did not have to worry about date-related variability in the test results.

We accepted a certain amount of *UI sensitivity* as the cost of doing business. When we made changes to the application that resulted in the HTML changing, we would manually run through the tests with recording

```
<CommandLog>
    <Exchange>
        <Request>action=generateInvoices</Request>
        <FinalResult>/demodashboard.jsp</FinalResult>
    </Exchange>
    <Exchange>
        <Request>action=getAllInvoices </Request>
        <IntermediateResult>
            <?xml version="1.0" encoding="UTF-8"?>
                <ArrayList>
                    <invoice number="24"> ...</invoice>
                </ArrayList>
        </IntermediateResult>
        <FinalResult>
            <html>
                <head><title>DisplayInvoices</title></head>
                <body bgcolor="#FFFFFF">   ...   </body>
            </html>
        </FinalResult>
    </Exchange>
</CommandLog>
```

Figure 4. Recorded Script for "Billy" (excerpt)

turned on. Once we verified the results were correct, we replaced the original playback file (which included the expected results) with the newly recorded one. As long as we ensured that we did not change the visible functionality, refactoring benefited from the safety net of our regression tests.

Return on Investment

It took approximately 1 person day (8 hours) to build the recording capability into the system and the data-driven test that read the recording file.

It took about 15 minutes to retest all the demo functionality reasonably rigorously visiting each page and exercising each function, verifying that the correct page was reached, but not inspecting the calculated data except for a cursory "eyeballing". The tests were run several times per hour for several weeks of development. This would equate to approximately 160 (2*8*5*2) verification cycles. Done manually, this testing would have taken about 40 hours (assuming the developers could have been convinced to execute them manually this many times.) More than likely, the tests would have been run a lot fewer times and the cost would have been hidden as a "quality problem". ROI: approximately 5:1.

Case 2 – Project "Safety"

This project involved porting and re-engineering a rail traffic control system. The original system was built and deployed on OS/370 and was later ported to DOS and then OS/2. The announcement of end of support for OS/2 forced the system's owner to take action. Initially, the intent was to purchase a replacement system from a third-

party, but it soon became apparent that this would be a very expensive and time-consuming approach because of the extensive customization the system would require to comply with additional (beyond industry standards) business rules embedded in the current system.

An alternative approach of re-engineering the existing system was chosen. Based on the previous releases of the system, it was recognized that retesting the changes made to the system would be very resource intensive. The sheer number of test conditions that had to be verified made hand-written test automation a non-starter.

The project used a traditional "architecture-centric" development process with incremental development that demonstrated a working version of the system every 4-8 weeks with ever increasing functionality. The architecture was defined largely up front and detailed design was done at the beginning of each increment of functionality.

Test Strategy

Because the re-engineered system was expected to work the same way as the original system ("ideally, the user's won't know it's a different system"), we proposed using it as a "gold standard" against which the new system would be compared. We would record tests by exercising the old system and recording which screens and fields were visited, what choices the system offered to the user and all the user inputs. Then we could play the tests back against the new system and verify that it behaved the same way.

Unfortunately, the existing system used a text-based window user interface. (Dialog boxes were drawn on the screen using the character symbols for vertical bars, horizontal lines and corners.) There were no commercial record & playback (R&PB) tools available that supported this technology let alone on both OS/2 and Windows 2000. This forced us to build our own R&PB capability into the system.

Test Architecture

The system was built before it became commonplace to separate the business logic from the user interface code. As such, there was no "application façade" (internal API) that could be hooked for the test tool. The code was organized as a large number of field processing modules. The business logic was mostly scattered throughout these modules interspersed with the UI code.

The R&PB tool was built into the user interface of the system by placing R&PB hooks wherever the application asks the user for input and recording the information into an XML file. Much of recording could be done by hooking utility functions called from many places in the code, but we sometimes had to add an extra parameter so the utility would know the context from which the field name could be generated. In other cases, we had to place hooks into the processing code for the fields themselves.

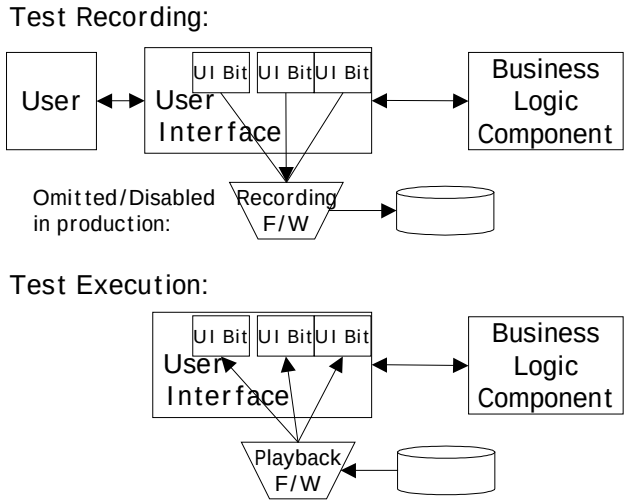

Figure 5. Test Architecture for Project "Safety"

For playback, we replaced the main menu driver of the system with a loop that read the commands from the playback file. Whenever the system visited a field, our hooks did a callback into the R&PB framework to retrieve the recorded "user input" and to compare the system outputs with the expected outputs. The results of the comparison were captured in an annotated version of the playback file that include the status of each field (either OK, Missing, or Surplus). The resulting XML file was formatted using a style sheet to resemble the output from a FIT test [3].

We built a plug-in for TestDirector [7] using the "Open Test Architecture" so that our re-engineered system could be launched from within a TestDirector test suite for completely unattended execution of test playback. The plug-in would copy the playback from TestDirector's repository to the local file system, launch our application, retrieve the results file and other log files and put them back into the TestDirector repository. Based on the outcome, it would mark the tests as either passed or failed.

Why It Works

Behavior Sensitivity was avoided because the system

```
if (playback_is_on()) {
    choice = get_choice_for_playback(dialog_id, choices_list);
} else    {
    choice = context->menu_choice;
}

display_dialog(choices_list, choice, row, col, title, key);

if (recording_is_on())  {
    record_choice(dialog_id, choice_list, choice, key);
}
```

Figure 6. Sample R&PB Hook for "Safety"

functionality is essentially frozen for the duration of the re-engineering project. Some minor changes are being made to the system, but care is being taken to ensure that the changes will not affect R&PB testing.

Interface Sensitivity was avoided by building the R&PB capability directly into the application. Cosmetic changes will not affect the accuracy of R&PB testing because the R&PB hooks are behind the logic that formats the screen and parses the inputs.

Data Sensitivity was avoided by carefully controlling the data that the system is tested with. All tests start with a known starting point in the database. The testing harness records the version of the data along with the test and automatically launches the system with the same version of data when the test is replayed.

Return on Investment
The R&PB capability was built at a cost of approximately half a million dollars over the course of 8 months. From past experience, it was expected that the number of manual test cycles required to validate the software would be about 5 with a duration of 3 months and 5 resources. This would cost about $600K (5*3*5*$8K). The planned best-case expectation for the project with R&PB testing is 1 test cycle with a second cycle planned as a contingency. (Test planning costs are omitted because they would be similar for either approach.) This is a reduction of about 4 test cycles at $120K each (total 480K) and an elapsed time of 1 year. Additional savings come from being able to scale down the project team to a maintenance team a year earlier. So there is a net cost reduction within the current project in addition to delivering to the business one year earlier and removing a great deal of personal stress for the key decision makers. Direct ROI: 5:6. Indirect ROI: Priceless.

```
<interaction-log>
  <commands>
    <command seqno="2" id="Supply Create">
      <field name="engineno" type="input">
        <used-value>5566</used-value>
        <expected></expected>
        <actual status="ok"/>
      </field>
      <field name="direction" type="selection">
        <used-value>SOUTH</used-value>
        <expected>
          <value>SOUTH</value>
          <value>NORTH</value>
        </expected>
        <actual>
          <value status="ok">SOUTH</value>
          <value status="ok">NORTH</value>
        </actual>
      </field>
    </command>
  </commands>
</interaction-log>
```

Figure 7. Results XML for "Safety" (excerpt)

Status
This project is in the later stages of development. The test cycles are expected to start in early 2004.

Other Considerations
Initially, the cost of building the test tools and recording the tests was justified solely for the duration of the re-engineering project. We expected it to pay for itself within the first release of functionality. We also anticipated that an XUnit-based approach would be phased in to replace the R&PB testing once the system was refactored to be testable.

Now that the R&PB capability is mostly built, we feel that if may be possible to continue to use the R&PB tests for many years to come. The track data versioning allows the existing tests to be used with newer versions of the system as long as the rules don't change. We have also devised ways to rejuvenate tests when the rules do change that allow much of the initial investment to be retained. New tests will still need to be planned and recorded for new rules, but most existing tests' expected results can be regenerated from the actual results once any test failures are verified by the business users as being expected based on the rules change.

Case 3 – Project "Inform"
This project involves the construction of a web site that provides the public with information about government and non-profit services. Data stewards enter information about their organizations and services into the system through the administration interface. Public health nurses and general public users use the public interface to find the services.

The project used an agile development approach (eXtreme Programming with some minor local adaptations) and released a working system to the stakeholders every 3 weeks.

Test Strategy
This project built JUnit automated tests for all business functionality right from the start. As a result, very few bugs were being found in the business logic portion of the system. But the user interface logic was problematic. The Struts [9] code used to analyze the user inputs and build the response HTML was having bugs appear in each new release. As a result, we decided to use a commercial R&PB tool, Astra QuickTest [7] to verify the Struts code. Since we already had functional tests for all functionality, the R&PB tests were focused on visiting each page and verifying that all buttons and tabs took the user to the right place and that the screens were formatted correctly.

In effect, we were using the R&PB test tool to do component testing on our UI.

Test Architecture
During recording, the commercial R&PB tool is started by the user who then interacts with the system via the web browser while the tool records the interactions During playback, the R&PB tool emulates a user interacting with the browser that communicates with the web server being tested. The R&PB tool does not interact directly with the system under test.

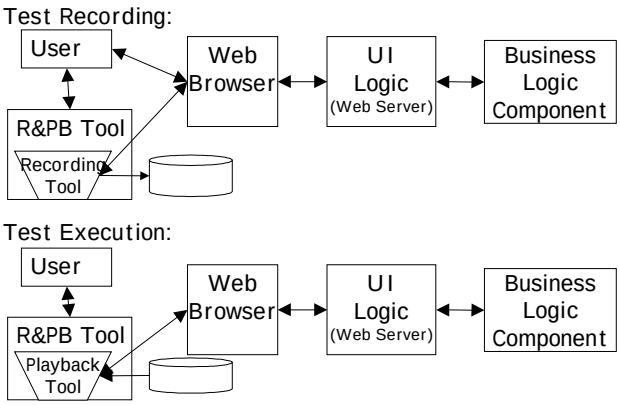

Figure 8. Test Architecture for "Inform"

Why It Works
Behavior Sensitivity was somewhat avoided by focusing on the User Interface logic. The tests were designed specifically to verify the screen flows, not the business logic (which is tested via the JUnit functional tests.) They avoided looking at the data on the screens to make them less sensitive to changes in the database.

Interface Sensitivity was accepted as the cost of automating these tests. It is accepted that we may need to re-record some of the tests whenever the user interface changes, but this is still better than rerunning all the tests manually all the times the interface hasn't changed. The time to record a test is only 10-20 minutes per script longer than to run the test manually. Most of the additional effort is tweaking the sensitivity of the test checkpoints and removing any automatically recorded checkpoints that cause the test to fail when it is rerun. (The buttons often invoke JavaScript that invokes hidden buttons. QuickTest [7] records both button presses but only the visible button test can be played back so the hidden button press must be removed manually.)

Data Sensitivity was avoided by carefully controlling the data that the system is tested with. All tests start with a known starting point in the database.

Context Sensitivity was avoided because the system has no interactions with other systems and very little (if any) of the behavior is time/date sensitive.

```
Browser("Inf").Page("Inf").WebButton("Login").Click
Browser("Inf").Page("Inf_2").Check CheckPoint("Inf_2")
Browser("Inf").Page("Inf_2").Link("Taxonomy Linking").Click
Browser("Inf").Page("Inf_3").Check CheckPoint("Inf_3")
Browser("Inf").Page("Inf_3").Link("Maintain Taxonomy").Click
Browser("Inf").Page("Inf_4").Check CheckPoint("Inf_4")
Browser("Inf").Page("Inf_4").Link("Add").Click
wait 4
Browser("Inf_2").Page("Inf").Check CheckPoint("Inf_5")
Browser("Inf_2").Page("Inf").WebEdit("childCodeSuffix").Set "A"
Browser("Inf_2").Page("Inf").WebEdit("tax.desc").Set "Top Level"
Browser("Inf_2").Page("Inf").WebEdit("tax.defn").Set "Top Level"
Browser("Inf_2").Page("Inf").WebButton("Save").Click
wait 4
Browser("Inf").Page("Inf_5").Check CheckPoint("Inf_5_2")
Browser("Inf").Page("Inf_5").WebList("selTaxCode").Select "Top"
wait 4
Browser("Inf_2").Page("Inf_2").Check CheckPoint("Inf_2_2")
Browser("Inf_2").Page("Inf_2").WebEdit("child").Set "B"
```

Figure 9. Recorded Script for "Inform" (excerpt)

Return on Investment
At printing deadline, it is still too early in the project to project actual dollar savings, initial indications are that the use of the R&PB tool for regression testing will save considerable effort due to the detection of UI logic changes much earlier in the development cycle.

Status
This system is expected to be in production in October 2003.

5 RECOMMENDATIONS

Record and Playback testing should be considered when:

- You need to refactor a legacy system to make it amenable to XUnit-style hand-scripted tests and you feel it is to risky to do so without having regression tests.
- You cannot afford the time or cost of hand-scripting tests
- You do not have the programming skills required to hand-script the tests.

Record and Playback testing should be avoided when:

- You cannot fix the behavior of the system by freezing/snapshot the data on which the system will operate.
- The behavior of the system is expected to change significantly between when the tests can be recorded and when they will be played back.
- If you want to use the automated tests as a specification and there is no existing system that can be used for recording the tests.

Critical Success Factors
Given that you have decided to give *robot user* testing

tools a second chance, what features do you need to look for in the testing tool? And what techniques do you need to apply to system development and test automation to be successful?

Designing the system for context independence.
If the behavior of the system depends on any outside factors such as the current date or time, you must be able to control the system context so that the same context is used for each test run. For example, configure the current system date at the start of each test run.

Tool Provides Means to Initialize System
Tests must be able to start up the system with the known starting point. For example, wipe out the database and reloading it with a new copy of the standard data.

Functionality Stability
R&PB testing can only be used to good effect when a significant portion of the applications functionality is expected to be unaffected by the next release. Any tests that encounter modified functionality must be rerecorded as the functionality is verified manually.

User Interface Insensitivity
It must be possible to record tests in a way that cosmetic changes to the UI do not cause tests to fail. Many of the commercial test tools allow you to set the sensitivity of the checkpoints they record. This is just one of the ways tests of the business logic can be made less sensitive to UI changes. But these controls are what makes commercial tools more complicated and therefore harder to learn.

UI-Insensitive Business Logic Tests
All tests that verify business logic should be recorded in a way that minimizes *UI Sensitivity*.

Business Logic and Data Insensitive UI Tests
A separate set of tests (either manual or automated) should be used to verify the UI has not changed. These tests should not care about the business logic or the data in the database.

A useful trick is to record tests with different sensitivity settings; these can then be used to do "defect triangulation" by narrowing down where the defect is located. For example, if UI sensitive tests fail while UI insensitive tests pass, the change must be in the UI.

Limited Lifetime
Recognize that *robot user* tests will have a limited lifetime. They will not survive certain kinds of changes to the user interface or the business logic inside the system. Make sure your strategy for managing the tests allows you to identify the those tests that will be affected and which will need to be either discarded, rerecorded or superceded by newly scripted tests. One good way of doing this is to cross-reference the tests with the requirements by using a test management tool such as Test Director [7].

6 CONCLUSION

Sometimes, R&PB testing is your only viable option given various project constraints. E.g., when dealing with a legacy system that does not have automated tests, Record & Playback style testing is a cost effective way to create regression tests that can be used to verify that design changes to the system do not introduce defects.

R&PB testing tools and techniques have matured significantly over the years and can now avoid many of the potential pitfalls when used properly.

When commercial R&PB test automation tools are unavailable, too costly, or too undependable, it is feasible to build the R&PB capability right into the system under test.

7 ACKNOWLEDGEMENTS

We would like to thank the many clients who gave us the opportunities to gain the experiences described in this paper. Also, Neil Kosman of Mercury Interactive who was always available to dig for information whenever we needed help using the Open Test Architecture. And Jennitta Andrea, Ralph Bohnet and Shaun Smith, who all provided invaluable feedback on various drafts.

8 REFERENCES

[1] Beck, Kent. Test Driven Development: By Example, Addison-Wesley, Boston MA, 2002

[2] Beck, Kent. Extreme Programming Explained, Addison-Wesley, Boston MA, 1999

[3] Cunningham, Ward. FIT: *Functional Integrated Test.* http://fit.c2.com.

[4] Fowler, Martin. Patterns of Enterprise Application Architectures. Addison-Wesley, Boston MA, 2002.

[5] HttpUnit and JfcUnit user interface testing frameworks: http://JUnit.org

[6] JUnit testing framework: http://JUnit.org

[7] Mercury Interactive Software. TestDirector test management & QuickTest test automation software. http://www-svca.mercuryinteractive.com/products/

[8] Meszaros, G. et al. "Test Automation Manifesto" in Proceedings of XP Universe 2003 (New Orleans, LA, August 2003)

[9] Struts web-based user interface framework http://jakarta.apache.org/struts/index.html

[10] XUnit family of testing frameworks: http://www.xprogramming.com/software.ht