

An alternative approach to software testing to enable SimShip for the localisation market, using Shadow™

Conference Papers

K Arthur, D Hannan, M Ward
Brandt Translations,
6 Faughart Terrace,
St. Mary's Road,
Dundalk, Co. Louth.
info@BrandtTranslations.com

Abstract - *We believe that our approach to automated software testing is novel. We can test several language instances of a product simultaneously, either through direct engineer interaction or by a record/playback script. The Shadow™ application can manage a situation where the user interface of the product under test is slightly different in layout, either due to localisation of the different versions, or due to the original language version running on different platforms. In our pilot studies, we examine the effect of separating out the functions of a test engineer into a product specialist and QA specialist. Our testing methodology outputs a set of screenshots of the products under test in each language. The screenshots can be used by a translator for linguistic/consistency QA or in product documentation. We performed a comparative analysis of the automation tool Winrunner with the Shadow™ testing process.*

Keywords: Automated testing, Quality Assurance testing, QA.

1 Introduction

The purpose of this paper is to describe a new automated testing tool called “Shadow™” developed by Brandt Translations and to illustrate how it works with the aid of case studies. Shadow™ is a software application that allows the user to control one or more operating systems (PCs, VMWare instances) simultaneously. The idea of one computer controlling another is not new and there are many products available for this purpose such as VNC (RealVNC Ltd 2002). However, the idea of one computer controlling many computers simultaneously appears to be novel.

SimShip is the process of shipping the localised product to customers at the same time as the original language product, or within a couple of weeks. SimShip can result in increased market share (Common Sense Advisory 2004) and possibly increased revenues. SimShip can result in unrecoverable costs due to factors such as (Langewis 2003); localised software not taking off in foreign markets, localised software being substandard, or localised software having been created inefficiently.

In this article we propose; to examine software testing, explain what Shadow™ is and how it is relevant to testing, and finally discuss some case studies. In the first section, we will introduce Shadow™ and describe in general terms what the product does. We will give an overview of the Shadow™ functionality and describe how it might be used to test original products and localised products.

2 Software testing and software quality

In this section we examine software testing and software quality in general terms. Towards the end of this section we will introduce Shadow™.

2.1 What is quality?

In this section we are interested in the definitions of quality as applied to software development. The Crosby definition of quality (Crosby 1980) is terse and defines quality as “conformance to requirements”. An alternative definition states that quality is “fitness for use” (Juran 1951). The value of these context free definitions is that they can be used in domains other than software (Mass 2004). As mentioned in all of the above references, the key driver of quality in the relevant context is the customer. Alternative, but consistent definitions of quality are given in (Mass 2004b). The ISO 9126 standard (Goodyear 2007, Wikipedia 2007) provides the definition of the characteristics of software quality. This standard does not provide a metric for software quality nor does it propose methods for quality measurement. The characteristics of quality are as follows (King 1996):

- Functionality is the set of attributes that bear on the existence of a set of functions and their specified properties. The functions are those that satisfy stated or implied needs.
- Reliability is the set of attributes that bear on the capability of software to maintain its level of performance under stated conditions for a stated period of time.

- Usability is the set of attributes that bear on the effort needed for use, and on the individual assessment of such use, by a stated or implied set of users.
- Efficiency is the set of attributes that bear on the relationship between the level of performance of the software and the amount of resources used, under stated conditions.
- Maintainability is the set of attributes that bear on the effort needed to make specified modifications.
- Portability is the set of attributes that bear on the ability of software to be transferred from one environment to another.

For our purposes, software quality will be defined as software that conforms to customer driven requirements and design specifications.

We believe testing should be viewed as an integral part of the software development process, rather than an activity performed after the core development. This idea is fundamental to software development approaches, such as “eXtreme Programming” (Succi and Marchesi 2001). The key phrase for this programming model is “Never write a line of code without a failing test” (Noyes 2006). Software testing should be viewed as a scientific process, wherein an application is placed under known conditions of setup, hardware and configuration, where it accepts some known input, and then a known and expected outcome should result. We expect that the quality of the test process and effort will have an impact on the software quality, just as the quality of the software is determined by the quality of the software development process (Kitt 1995).

Errors or bugs are introduced into software in a large variety of ways during the development process. Bugs are typically of the following types:

- Logical errors – where the software runs, but produces unexpected results.
- Crash bugs – where the software fails in a catastrophic manner.
- Failed functionality – where the application fails to meet its specifications.
- Layout issues, widgets/text not displayed correctly.
- Linguistic issues – spelling, grammar.
- Localisation issues – date/time format, number format.

2.2 What is software testing?

A good definition of software testing is given in (Hower 1996), namely; software “testing involves operation of a system or application under controlled conditions and evaluating the results. The controlled conditions should include both normal and abnormal conditions. Testing should intentionally attempt to make things go wrong to determine if things happen when they shouldn't or things don't happen when they should.” Software testing is a process of quality control using a group of defined methods and evaluation criteria, together with guidelines for their use.

2.3 Why perform software testing?

Software testing is performed to find defects, to ensure that the code matches the specification and to estimate the reliability of the code. The output of testing is a defect list and the effectiveness of the testing process can be measured through the number of defects (Voas 2004). Examples for estimating the number of defects in a product are given in (Chulani and Boehm 1999; Longstreet 1995). As the number of defects is reduced, there is an enhanced confidence in the quality of the product.

Testing adds to the cost of production. When balanced against the idea that the most expensive defect to fix is the one found by the customer, it is highly desirable to capture defects as early as possible in the software development lifecycle. There are several types of software testing. For example, “Unit Testing” means testing a unit of code in isolation, but could also mean testing an individual software method or module. The unit test is an example of a test to output a finite result. Testing focuses primarily on the evaluation or assessment of quality and is realised through a number of core practices consisting of at least the following (O'Regan 2002);

- Validating, through concrete demonstration, the assumptions made in design and requirement specifications.
- Validating that the software product functions as it was designed to.
- Validating that the requirements have been implemented appropriately.

The cost of software bugs is large. In 2002 a US government agency suggests that software bugs cost \$59.5 billion annually (Tassey 2002). The report suggests that over half these costs are borne by the users and the remainder by the developers. It is clear that if the number of bugs can be reduced, large savings can be made.

2.4 What are the types of software testing?

There are two approaches to software testing, namely, manual testing and automated testing. In general, testing is where an engineer, or team of engineers, takes a product, writes a test script to navigate through the product while testing the user interface looks the way that it is supposed to and the product functions the way it is supposed to.

Manual testing uses human engineers to aid in this process. Automated testing requires that the test script is coded, and then executed using some application.

2.5 What are the characteristics of automated and manual testing?

Manual testing is vital to the development or localisation of any product. In (Masciana 2005) it is shown that; manual test scripts can be used to provide feedback to development teams in the form of a set of repeatable steps that lead to bugs or usability problems and manual test scripts can also form the basis for help or tutorial files for the application under test. Manual testing also does something automated testing can never do, namely, ad-hoc testing. Ad hoc testing is testing outside the test script, where the engineer will explore and improvise a test strategy. Ad hoc testing can be used by a skilful engineer to go beyond the standard tests given in a test script. If these tests prove valuable, they should be documented and repeated manually or by automation. In this way, automation complements manual testing.

Successful software development and localisation should use both manual and automated testing methodologies. The balance between each can be decided using budgetary and schedule constraints. Following (Ford 2005) the advantages of automated testing include repeatability, and a disadvantage of manual testing is that repetition can be tedious for the engineer. There are areas that automated testing cannot replace, such as the use of a skilled linguist examining a product for use of the appropriate translations. Shadow™ can be used to complement manual testing and be used as a test automation solution.

2.6 What is Shadow™?

Shadow™ is a software-testing tool for performing automated and manual tests on original or localised software products. Shadow™ allows the user to record and playback scripts that run and control multiple machines at the same time. It also allows the user to directly interact multiple machines simultaneously, making the manual test effort more efficient. Shadow™ allows the user to simultaneously test localised software applications running on:

- Different language operating systems.
- Original language products running in different configurations.

Shadow™ does not have a difficult user interface or an arcane programming language. With Shadow™, a user can become productive in a short time. We have tried to address the issues raised above, but also address some of the concerns expressed in relation to automated testing (Miller 1995a). In (Miller 1995b) we can see a discussion of automated testing, and in particular a discussion of the synchronisation problem. This synchronisation issue relates to the replaying of mouse and keyboard actions faithfully to reproduce what the user entered. A failure to synchronise during playback causes the test to fail, indicating that the application has in some way changed, when in fact this is generally not the case. The methodology used to overcome this is “Object Oriented” testing. These tests are more robust, and rely on “under the hood” information about the application. In common with many other testing tools, there is an invasive requirement that the windowing library and other operating system data must be accessed. This ties the automated testing tool to a particular operating system or coding language such as C++ or Java. This is not the case with Shadow™.



Figure 1 Shadow™ setup

There are different types of automated testing modes. See for example (Miller 1995b), where the following types are listed;

True-Time: The first mode is true time capture/playback. Keyboard and mouse input are recorded with the exact timing employed by the user, and replayed in exactly the same way. This ensures tests are executed as if a human user were performing them.

Character Recognition: The test application searches for the text of interest. This means that the text does not have to have a fixed position from test to test. Text can also have changed in point size or font since the test was recorded.

Widget (Object-level): With Widget (for a definition see for example (HYPERDICTIONARY 2005)) or object-level playback, the X and Y coordinates on the screen are no longer significant as the application's widgets are activated directly by the testing software. The widget can be located anywhere on the screen.

The results of True-Time testing will indicate variances from the baseline cases. For example if a button moved to a different location in the window, this would be flagged as a bug. The tester will then determine if this is a “real” bug or a “non-issue”. The scripts for character recognition may be used through many versions of an application, where the user interface look and feel may change. The power of the object-oriented approach is that it lends itself to cross-platform testing. However, the conventional products and tools for automated testing use a significant amount of information from the windowing libraries and operating system. This ties some automated test software to a particular operating system.

Shadow™ tries to avoid the disadvantages of these modes, and combine the advantages in one product. Shadow™ uses the keyboard, mouse and timing information from the user, and employs intelligent technology to identify the location of the interactions in the screen. During the replay of scripts Shadow™ finds the appropriate location in the screens under test. This allows for widgets to have changed position and size between tests. We are at present adding the character recognition module to the Shadow™ suite of tools. This has particular relevance to localisation, where testing is not just a matter of finding the appropriate text, but where a mapping exists from one language to another.

For the Shadow™ application, we are concerned with elements of each mode of automated software testing. We have to capture mouse and keyboard interaction, and we have to be aware of the timing issues, to ensure that our playback is in synchronisation with the application behaviour. We will use character recognition, and the logical extension, namely, widget recognition. This will give us the power of the object oriented automated testing mode, without the requirement to query the API or windowing libraries of the operating system.

We believe that our approach takes something from each mode of automated testing. Our development philosophy has been that we avoid operating system dependencies, where possible. We want to avoid getting information about the application under test that is not available on all operating systems in the same manner. An example of a testing tool that uses “under the hood” information is “Rational Robot”. Rational Robot natively knows how to manipulate SQABasic GUI objects only. To test Java applications and applets, Java GUI components have to be mapped to the SQABasic GUI objects. Rational Robot can dynamically determine the class, functionalities, and properties of a Java GUI component loaded into the JVM, using a process called Reflection (Olsen 2007). Reflection is a process used to get information about an object and its publicly accessible methods and members (Litwak 2000). It uses the information generated from using the Reflection API to determine which SQABasic object the class maps to, and the proxy it can use to interact with the Java component.

The Shadow™ process concentrates on getting information about the application under test that is available to the human user. This information partly consists of analysis of visual cues and therefore analysis of screenshots of the application. We have to write operating system dependent drivers for capturing mouse and keyboard interactions, but the bulk of the Shadow™ application code will not have to be changed. The specific problems we address with Shadow™ are:

- Making automated software testing easier to use with less programming.
- Separating the roles of Test Engineer into the complementary roles of “Product Specialist” and “Quality Assurance Specialist”. See the glossary of terms for proposed specifications of the two roles.
- Making software testing more like the actions of a human user.
- Making automated testing easier – reducing the barrier to automation.
 - o Reduction of the cumbersome nature of script creation (Dustin 1999).
 - o Reduction of the training time necessary for the tool to be useful.
- Increasing compatibility with 3rd party application components (widgets), see (Dustin 1999).
- Accelerating the manual testing process through Shadow™'s unique user interface.
- Recording screenshot data by default. In other automated testing applications this has to be implemented programmatically.

- Script maintenance reduction. This is accomplished by examining the developer's release documentation to identify those user interface areas that have changed in the new build. We can then compare screenshots from the newly run tests with previously recorded screenshots in those areas that have been changed.

2.7 Advantages and disadvantages of automated testing

In this section we would like to summarise the differences between manual and automated testing, see for example (Ford 2005).

Advantages of Automated testing

- If a set of tests has to be run repeatedly.
- Run automation against code that frequently changes to catch regression errors quickly.
- A computer does not get tired or bored.
- Aids in testing a large test matrix (different languages on different OS platforms).
- Automated tests can be run at the same time on different systems, whereas the manual tests would have to be run sequentially.
- High productivity – 24 x 7.
- Consistency in producing results.

Disadvantages of Automated testing

- It costs more to automate:
 - o Writing or configuring the automated framework costs more initially than running the test manually.
 - o The scripts have to be maintained as the software changes.
- It takes longer to write, test and document automated tests.
- Test automation is software development (Skrivanek 2005).
- Visual references or cues cannot be automated, for example, if you can't tell the font colour via code or the automation tool, then it should be a manual test.
- Test cases are fixed and not varied, and there is no ad hoc testing.
- Existing automated tools can only test applications on an operating system that provides a software interface to which they can plug into. In some cases such as multimedia software products, software games and mobile devices this software interface is not provided.

Advantages of Manual testing

- Infrequently required test cases are cheaper to run manually than to automate.
- Manual testing allows the tester to perform more ad-hoc testing.
- Manual testers are capable of making variations in test cases to more fully explore the application under test.
- Tests product usability.

Disadvantages of Manual testing

- Running tests manually can be very time consuming (installing and maintaining multiple operating systems).
- Each time there is a new build, the tester must rerun all required tests – could become mundane and tiresome.
- It can be difficult to get a group of different engineers to perform a set of test instructions manually in a reproducible manner.

Automation has a high initial investment cost, but a low ongoing maintenance cost. Manual testing on the other hand, is a fixed cost often incrementing on a daily basis. There are additional criteria that can be used to assess how the manual plus automated testing mix is decided, such as; whether or not automation is possible, the skill set or training of the engineers, the budget, the schedule, or the return on investment.

3 Shadow™

Automated testing tools currently come in three general categories with different operating modes, namely (Skrivanek 2005):

- Simple capture – record and playback.
- Object Oriented Automation (API calls “under the hood”).
- Image-based component discovery.

Simple capture utilities: Using a “simple” capture utility an engineer can create a test script containing an exact sequence of keystrokes, mouse movements and or mouse commands. However, with the slightest change to the GUI of the software under test the recorded script becomes invalid requiring the engineer to possibly re-record the script.

Object Oriented Automation: Makes API calls to the operating system to identify information about the control being interacted with. Once the automation tool has the handle for the object, it can then manipulate the control. It is the reaction of the code to a function that makes an API call that is being tested, not the actual user interface. This is an important distinction.

Image-based component discovery: In this mode of automated testing, images are taken of regions around the mouse at the time of some mouse interaction. This image is then stored. At a later time the image is used to identify where the

mouse action should be taken either during playback of a script or where the action should be taken on some other system. This is the method used by Brandt Translations in the Shadow™ application. The characteristics of image based component discovery are similar to those of Object Oriented automation, without the need for any of the “under the hood” information.

Shadow™ can operate as a “simple capture” utility in its control of multiple operating systems simultaneously. This can be useful for some application testing, but it is sensitive to graphical user interface changes and the position of the application on the screen. Tools other than Shadow™ that implement object oriented automation can deal with layout changes in the application under test and are insensitive to screen location, but they make API calls to the operating system. Shadow™ uses image-based component discovery to find the appropriate location on the screen. In this way it comes close to object orientation, but without API calls. In “Exact match” mode, Shadow™ becomes more than a “simple capture” utility, and the layout of the screen is less relevant to the execution of scripts.

3.1 Shadow™ setup

Shadow™ is a piece of software that can control several machines simultaneously. It can do so in several different ways, such as:

- Shadow™ can make a group of machines perform exactly the same actions at the same time.
- Shadow™ can make a group of machines perform “nearly” the same action at the same time.
- Shadow™ can record and playback scripts (“Mimic” and “Exact Match” modes).

4 Case studies

In this section we will look at case studies involving the use of Shadow™. We will examine three studies where Shadow™ was used, and the purpose for which it was used.

<i>Client</i>	<i>Profile</i>
A	Multinational software publisher.
B	Supplier of technical authoring, documentation and localisation services
C	Brandt Translations

Table 1 List of case studies

4.1 Client A

Client A produces enterprise resource planning (ERP) software for managing and analysing corporate spend. Their clients include “Fortune 100” multinationals. Brandt provides translation and engineering services to this client.

4.1.1 Task specification

Client A requires that the translated and localised user interface of its products undergo linguistic testing and functional testing. For this test case, the client used a combination of methods to perform the linguistic and functional testing, that is using Shadow™ and WinRunner. The engineer performed the following tasks:

- Wrote test scripts.
- Updated test scripts.
- Set up the hardware and software.
- Execute the test script on the machines; using both Shadow™ and WinRunner.
- LQA – performed by linguists using the screenshots.
- Localisation functional QA using Shadow™ and WinRunner.

4.1.2 Shadow™ usage

In this section we examine how the output of Shadow™ is used in linguistic testing and how Shadow™ itself is used in functional testing.

Linguistic testing

Software is translated in a tool that does not show the translator the context of the strings. No matter how familiar the translator is with the product, and how much experience they have on previous versions of it, there is no substitute for having the translator seeing the translated strings appearing in a running build in their proper context. The translator can then make the appropriate changes, if necessary, to the software strings. There are at least two methodologies that can be used in this situation such as having the translator go through the running build with a test script to bring up every screen, or providing the translator with the screens in the form of screenshots (or MHT files in this case study). We use the second method in this case study. Some advantages of giving the translator only screens to review include that the translators do not have to have spend time negotiating their way through the product to find the areas in which

an update occurs and there are cost savings in setup time. This is especially useful for small and frequent updates to a product.

Functional testing

The engineer used Shadow™ with three machines for functional testing, each machine running a different language operating system. The output of this was a list of bugs documented by the engineer. In this process the engineer manually went through the test script on one machine, with the others automatically following in Shadow™, noting bugs as they progressed. As with the linguistic testing, the process was then repeated with a different language set.

4.1.3 Results

Table 2 and Table 3 below show the time spent using both Shadow™ and WinRunner to perform the same tasks for a QA cycle. The engineer has detailed the time taken for the average amount of days spent on each task. The tables show the complete list of tasks.

<i>40 screenshots</i>	<i>Shadow™</i>	<i>WinRunner</i>	<i>Total</i>	
<i>Task</i>	<i>Days</i>	<i>Days</i>	<i>Days</i>	<i>Comment</i>
Write LQA script			3 – 4	Tool independent
Update LQA script			1 – 2	Tool independent
Write TSL script		1 – 2		WinRunner only
Execution using tool for screenshots	2	1		Both Shadow™ and WinRunner
LQA by translators			1 – 2	Tool independent
Functional QA			1 – 2	Tool independent
Total days	8 – 12	8 – 13		

Table 2 Results of using Shadow™ Vs WinRunner for 40 screenshots

<i>400 screenshots</i>	<i>Shadow™</i>	<i>WinRunner</i>	<i>Total</i>	
<i>Task</i>	<i>Days</i>	<i>Days</i>	<i>Days</i>	<i>Comment</i>
Write LQA script			20 – 25	
Update LQA script			10 – 15	
Write TSL script		25 – 30		WinRunner only
Execution using tool for screenshots	16	8		
LQA by translators			5 – 6	
Functional QA			9 – 10	
Total days	60 – 72	77 – 94		

Table 3 Results of using Shadow™ Vs WinRunner for 400 screenshots

The engineer notes that WinRunner can fail if the Internet connection is slow and fails to bring up a widget in a “reasonable” time. A key piece of functionality of Shadow™ is “Wait for feature”. This means that Shadow™ waits a configurable amount of time for a feature to appear on the screen before declaring a failure.

4.1.4 Conclusions

From the tables above we can see that Shadow™ and WinRunner take approximately the same time to setup and run a test cycle in which there are a small number of screenshots required. Where a larger number of screenshots are required, the time taken to run Shadow™ is less than the time taken to write the code and execute it in WinRunner. The engineer notes in his report that WinRunner requires the build of software under test to be specially prepared in order to function with it, whereas Shadow™ does not. Each resource on the page has to have an AWL name (AWL is a client proprietary web language). The AWL name identifies the widget without reference to the text on it, so that the same button in French and German will have the same AWL name, but different text. This is an “under the hood” requirement of WinRunner that Shadow™ does not have. In summary, Shadow™ was used as a QA tool in this study. Shadow™ was more efficient than WinRunner on the QA of a larger product. Shadow™ did not require special preparation of the product build before its use and as a result it could be used “out of the box”.

4.2 Client B

Client B is a provider of documentation, consultancy and recruitment solutions.

4.2.1 Task specification

Client B wanted Brandt to prepare a document containing screenshots of their software that could be sent to translators for review. There were 203 screenshots to be taken of the software opened in a software engineering tool. This tool is a visual localisation environment allowing engineers to view the localisable resources of software. The final deliverable to Client B was one PDF per language, each containing the 203 screenshots of the English and localised software side by side.

4.2.2 Shadow™ usage

Shadow™ was installed on 5 VMWare machines each running Windows XP Professional. The appropriate tools were also installed. Each localised software file was opened in the visual editor on one virtual machine, and then the English, giving a total of 5 connected clients to Shadow™. Usually, Shadow™ takes a screenshot of the whole screen, or if configured to do so, just the window in focus. In this project, the window in focus was the visual editor application and the required dialog was part of a screenshot that would need to be cropped. The Shadow™ code was modified, using an extension of the existing “exact match” technology so that a screenshot could be adaptively cropped to leave only a feature of interest. This project is unique among the test cases, as it showed how the software development team was integral to completing the project using Shadow™ by making appropriate modifications to the application.

4.2.3 Results

The engineer took about 1.5 hours in total to setup and screen shoot 4 languages. The integration of the screenshots into the Word document and production of the PDF are tasks independent of Shadow™ and would be the same length irrespective of how the screenshots were taken.

4.2.4 Conclusions

Shadow™ was used as a QA tool for this project, where the output was a set of screenshots for linguistic QA performed by the translation team. This project involved use of a software application that requires significant processing resources of the host operating system. We found that the process using Shadow™ was faster than the manual process, directly as a result of the ability to perform tasks in parallel.

4.3 Brandt Translations

Brandt Translations uses Shadow™ for the purposes of testing and as an automation tool to perform tasks that need to be repeated frequently. Brandt Translations has found that there are tasks in the production of multimedia tours in Adobe Captivate® that are repetitious and prone to human error, such as; audio integration, text integration, and font assignment. We have written short scripts, called macros, which can be activated using different keystrokes that perform these tasks. Let us look at each one in turn.

4.3.1 Task specification

Adobe Captivate® is a tool for rapid authoring of multimedia tutorials. These tutorials contain screenshots, text, animations and audio that all have to be localised. The localised elements have to be integrated into each slide in the tour. Tours can vary in size from 30 slides to in excess of 100 slides, and are localised in a number of European and Asian languages. Brandt has run this project several times over the past eighteen months and improved the process to make it more efficient, with the result that the output is more consistent and of a higher quality.

Audio integration

This involves importing a single WAV file per slide. The WAV file name is numbered in sequence, for example “0001.wav”. To perform this task manually, the engineer runs the risk of importing the wrong file into a slide. For an engineer who is not familiar with the language they will not be able to test that the audio is appropriate to the slide. This has to be repeated up to 100 times without error, for up to eight languages. That is a total of 800 cut-and-pastes.

Text Integration

There is localised text on every slide in the tour. On most slides, the text is unique. The engineer has a MS Word document with the text ordered sequentially for every slide. Once again, this is a straightforward process, but it is prone to error. This has to be repeated up to 100 times without error, for up to eight languages. That is a total of 800 cut-and-pastes. One possible error is that the text sequence goes out of line with the slide sequence. Once this happens, the process will have to be repeated from the place where the error occurred.

Font assignment

The localised text appearing on every slide has to be a certain font for European languages, the SimSun font for Simplified Chinese and MS Mincho for Japanese. Each slide has to have the font individually set. The engineer must go to every slide, bring up the text properties and set the correct font. Again, this has to be repeated for each slide and then for each language.

4.3.2 Shadow™ usage

Shadow™ is used in the following way. Three virtual operating systems are set up in VMWare, in our case MS Windows XP Professional. Shadow™ is set up to view the 3 operating systems, generally referred to as (virtual) machines. Each machine runs Adobe Captivate®, MS Word and possibly MS Notepad. Each virtual machine is set up with the correct script (French, German or Japanese as required). The macro is set to run the correct number of times, usually equal to the number of slides, and it is then it is started. Each run of Shadow™ can process one tour in three languages in parallel. While this is running, the engineer can usually go and perform another task.

4.3.3 Results

Shadow™ was vital for this project, as some of these tasks are repetitive and subject to human error. Table 4 shows the times taken for Shadow™ to run the individual tasks versus the manual time for the same task. Note that Shadow™ can run at least three tours in parallel so there is a further efficiency present.

<i>Task</i>	<i>Automation per 30 slide tour – minutes</i>	<i>Manual time per 30 slide tour – minutes</i>
Audio integration	10	25
Text integration	15	25
Font assignment	10	20

Table 4 Results for Shadow™ automation Vs manual implementation

Whereas one Shadow™ run will perform audio integration (or another task) for three tours in parallel, one engineer performs the manual equivalent process in a serial fashion.

4.3.4 Conclusions

Shadow™ was used as an automation tool for this project and it was essential to the effectiveness of the engineering team. For some tasks, the time it takes to perform the task manually is not much different from the time the automation takes. In this case the automation is more efficient when it is possible to perform the tasks in parallel. It might not be worth investing time in automating some once off tasks. For this project, Shadow™ was invaluable, because of the number of tasks that are repeated.

5 Conclusions

In this paper we have examined some of the advantages and disadvantages of the different modes of testing software applications. We have come to the conclusion that a mix of manual and automated testing is essential to the success of a project. One type of testing will not replace the other. The ratio of the mix between automated and manual testing can be dictated by the nature of the project, the budget and the schedule. Shadow™ can help make automated testing more efficient by separating the QA from specialist product knowledge and hardware setup. As part of the localisation process, Shadow™ can be used to take screenshots of the running software that can be given to linguists for review. Shadow™ can also be used by the engineer, with specialist product knowledge, to walk through the different language versions of a product at the same time.

References

- RealVNC Ltd (2002) RealVNC remote control software [online], available at: <http://www.realvnc.com/> [accessed 28 August 2007]
- Common Sense Advisory (2004) Common Sense Advisory explains how companies will become “world enterprises” [online], available at: http://commonsenseadvisory.com/news/pr_view.php?pre_id=8 [accessed 05 September 2007]
- Langewis C, (2003) Localization and ROI: Increasing Value by Eliminating Pink Ink [online], available at: <http://www.ableinnovations.com/pdf/pinkink-I.pdf> [accessed 05 September 2007]
- Goodyear A (2007) ISO 9126 Software Quality Characteristics [online], available at: <http://www.sqa.net/iso9126.html> [accessed 30 August 2007]
- Wikipedia (2007) ISO 9126 [online], available at: http://en.wikipedia.org/wiki/ISO_9126 [accessed 30 August 2007]

King, M (1996) The ISO 9126 Standard [online], available at: <http://www.issco.unige.ch/ewg95/node1.html> [accessed 30 August 2007]

Rick Hower (1996) Software QA and Testing Resource Center [online], available at: http://www.softwareqatest.com/qatfaq1.html#FAQ1_1 [accessed 28 August 2007]

Chulani S and Boehm B (1999) Modeling Software Defect Introduction and Removal: COQUALMO (CONstructive QUALity MOdel) [online], available at: <http://sunset.usc.edu/publications/TECHRPTS/1999/usccse99-510/usccse99-510.pdf> [accessed 28 August 2007]

Longstreet, David (1995) Test Cases and Defects [online], available at: <http://www.softwaremetrics.com/Articles/defects.htm> [accessed 28 August 2007]

Voas J (2004) A Few Assertions on Information Hiding [online], available at: <http://www.cigital.com/papers/download/qualitytime1.pdf> [accessed 28 August 2007]

O'Regan, G (2002) "A Practical Approach to Software Quality", New York: Springer-Verlag

Crosby, P. (1980) "Quality is free. The art of making quality certain", Penguin Books

Juran, J. (1951) "Quality control handbook", New York: McGraw Hill

Maas, K.F. (2004) Introduction to Quality [online], available at: <http://www.kfmaas.de/qintroed.html> [accessed 28 August 2007]

Maas, K.F. (2004b) Five Ways of Looking at Quality Definitions [online], available at: <http://www.kfmaas.de/qualidef.html> [accessed 28 August 2007]

Succi, G. and Marchesi M. (2001) "Extreme programming examined" Addison-Wesley

Noyes, D. (2006) Never Write a Line Of Code Without a Failing Test [online], available at: <http://c2.com/cgi/wiki?NeverWriteaLineOfCodeWithoutaFailingTest> [accessed 28 August 2007]

Kitt, E. (1995) "Software testing in the real world", Addison-Wesley

Tassej, G. (2002) NIST: The Economic Impacts of Inadequate Infrastructure for Software Testing [online], available at: <http://www.nist.gov/director/prog-ofc/report02-3.pdf> [accessed 28 August 2007]

Masciana R (2005) A simple tool to manage manual testing: Introducing IBM Rational Manual Tester [online], available at: <http://www.ibm.com/developerworks/rational/library/04/r-3232/> [accessed 05 September 2007]

Ford S (2005) Automation Testing versus Manual Testing Guidelines [online], available at: <http://testingsoftware.blogspot.com/2005/10/automation-testing-versus-manual.html> [accessed 06 September 2007]

Miller E.F. (1995a) APPLICATION NOTE: Software Test Tools Considered Harmful? [online], available at: <http://www.soft.com/AppNotes/harm.html> [accessed 05 September 2007]

Miller E.F. (1995b) APPLICATION NOTE: Multiple Synchronization Modes [online], available at: <http://www.soft.com/AppNotes/capmodes.html> [accessed 05 September 2007]

HYPERDICTIONARY (2005) Meaning of WIDGET [online], available at: <http://www.hyperdictionary.com/search.aspx?Dict=&define=widget> [accessed 06 September 2007]

Olsen R., (2007) Reflections on Java Reflection [online], available at: <http://www.onjava.com/pub/a/onjava/2007/03/15/reflections-on-java-reflection.html> [accessed 06 September 2007]

Litwak, K (2000) Pure Java 2, Basingstoke, UK, SAMS

Dustin, E. 1999, Lessons in Test Automation [online], available at: <http://www.stickyminds.com/sitewide.asp?ObjectId=1802&ObjectType=ART&Function=edetail> [accessed 06 September 2007]

Skrivanek, J. (2005) TestingGUIApplications [online], available at: <http://wiki.java.net/bin/view/Javapeda/TestingGUIApplications> [accessed 06 September 2007]