Introducing Automated Environment Configuration Testing in an Industrial Setting

Caryna Pinheiro
MCIT Solutions Inc.
Calgary, Canada
caryna.pinheiro@gmail.com

Vahid Garousi*, Frank Maurer+, Jonathan Sillito+
*: Department of Electrical and Computer Engineering
+: Department of Computer Science
University of Calgary
{vgarousi,frank.maurer,sillito}@ucalgary.ca

***Abstract*— This paper presents an automated environment configuration testing strategy developed as part of an action research project to deal with issues of staging environment instability in a large organization. We demonstrate how a suite of automated environment configuration tests provided an unobtrusive way to verify the hospitability of staging environments, decreased the time wasted on manual troubleshooting of environmental issues, and consolidated software configuration management information. The test strategy was also greatly welcomed by upper-level management and is now being expanded to other parts of the organization.**

Keywords – Environment configuration, Software configuration management, staging, software development lifecycle, test automation, test strategy, action research.

I. INTRODUCTION

Modern IT ecosystems have a complex combination of hardware, software, middleware, components, applications, organizational culture, practices, and application lifecycles [1]. Staging environments are part of the release lifecycle of many complex software systems under development. They are used to assemble, test and review new versions of a software system before it is deployed in the field. During staging, software applications are tested in "production-like" server environments that integrate code from all existing applications in one organization [2, 3].

During deployment to staging environments, developers start to find out about the configurations needed in order to successfully run the deployed application. Dependencies to other software applications and infrastructural components are also identified. In this paper, we define software environment configurations as:

- Settings that must be applied to servers
- Application-related settings (e.g., access permission) stored in files or databases
- Third party shared services and components.

Deployment to staging environments – and later to the production environment – can be either manual or automated. In complex IT ecosystems, it is hard to successfully automate deployments to encompass all of the outer dependencies on external components, systems from other project teams, and infrastructural configurations [1].

In large organizations that have multiple concurrent software projects, it is common to have project teams with different release schedules sharing the same physical resources for their staging environments. It is also common to have a combination of teams with their own automated deployment scripts and manual deployments of legacy applications [1]. Such diversity in deployment approaches and timelines often lead to instabilities and rework effort in staging environments' configurations.

Strict vigilance of configuration changes may reduce the number and frequency of changes introduced into staging environment. But this often creates bottlenecks for the quick delivery of iterations and necessary fixes for project teams. There is also a problem of governance. The fine line between enterprise-wide configuration management of staging environments and the configuration needed by individual project teams is usually a grey one. In our experience based on several large-scale industrial projects, configuration management is often shrunken to a source control repository that includes documentation of how to set up a known baseline for an application system.

The challenges that we have faced in an industrial setting are the goals for the work reported in this paper: (1) How can we validate that the environment configurations, our software application depends on, have been correctly applied to staging environments? (2) When our application starts failing after other teams deploy to the same staging environment, how can we quickly validate that this is not due to missing or changed configurations? (3) How can we provide a pro-active way to consolidate configuration management information?

In this paper, we present an action research and development project conducted to address the above issues with staging environment instability. We demonstrate how a carefully-designed automated suite of environment configuration tests can provide an unobtrusive way to verify the hospitability of staging environments, consolidate configuration information and external dependencies, and also decrease the cost associated with manual troubleshooting of environmental issues.

The remainder of this article is structured as follows. We introduce our guiding research methodology in Section II. The problem domain is discussed in Section III. The existing literature is discussed in Section IV. Our test strategy is presented in Section V. Lessons learned are shared in Section VI. We conclude this paper in Section VII.

II. GUIDING RESEARCH METHODOLOGY

Action research has been increasingly used in the fields of information systems [4]. It looks at combining theory and practice to solve an existing issue. The work of Davison *et al.*

[5] provides a concise set of principles and guidelines for researchers looking for ways to combine theory and practice in an industrial setting using an action research method called Canonical Action Research (CAR). The CAR model has five encompassing principles: (1) the creation of a researcher-client agreement – signed in August of 2007; (2) the cyclical process model with five distinct phases: diagnosis (discussed in Section III), action planning, intervention (action taking), evaluation, and reflection (all discussed in Section V); (3) the adoption of theory; (4) the implementation of change through action; and (5) learning through reflection.

During action planning, we further investigated the issues related to environment instability. Informal question & answer sessions were set up with team members. The questions were structured to ask probing questions and concrete examples [6]. The testing techniques used during the intervention and evaluation steps are discussed on Section V.

III. PROBLEM DOMAIN

This section describes the diagnosis phase of our action research project.

A. *Context*

The IT department of our industrial partner (based in Calgary, Alberta, Canada) has over 190 professionals. This paper studies projects from the largest IT Program in this agency with approximately 50 IT professionals. This agency's IT focus is to develop, enhance and maintain contemporary systems to enable timely responses to requests from the oil & gas industry. In 2008, there were a total of 18 different concurrent software projects with budgets ranging from $0.5 to $1.5 million dollars.

The overall architecture for hosted applications is illustrated in Figure 1. Servers are clustered for redundancy. The organization's projects are developed on a backbone of four staging environments: development (DEV), test (TST), acceptance (ACT), and production PRD. The environments are described in detail next.

Construction iterations lead to completed development activities built and delivered to the DEV environment and code ready and tested in an integrated environment (TST), which are constituted of many virtual servers. A production release includes many iterations. Once a group of iterations leads to a candidate release, the completed work must be deployed and tested in real staging (ACT), and if approved by all stakeholders, it then becomes a production release (PRD).

B. *Staging Environment Instability*

All in-house development teams share the staging servers. Software applications have to be deployed on several staging environments during different milestones of the release lifecycle. Changes to server configurations made by one team have the potential to impact other teams. This situation certainly happened very frequently in the organization under study.

On many occasions, successfully deployed applications started to fail and teams were not made aware of what actually changed in the environment. There was no formal ownership of non-hardware environment configurations. As a result, teams had to constantly and manually verify server configurations to ensure that all different environments were set-up correctly for their applications.

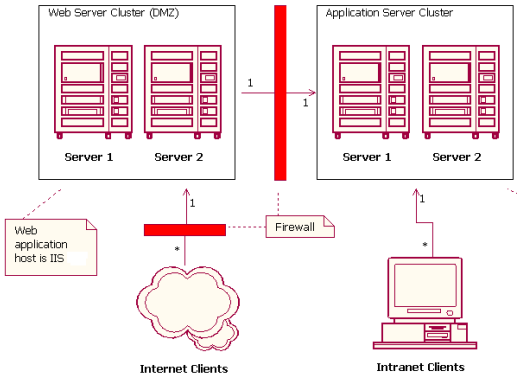

Figure 1. Overall architecture of hosted applications.

The entire situation was almost ad-hoc, caused last-minute surprises, and chaos in many occasions. "It worked for me yesterday, but I don't know why it doesn't work today!" This manual check of configurations was also very time consuming and frustrating. The company had looked for best practices out there to deal with the staging environment instability problem, however no good best practice was found. While using virtualized servers for each individual project would alleviate some of these instabilities, it would introduce an extra layer of complexity to test integrated systems. Existing problems with virtualization and n-tier applications using COM+ and other licensed third party tools [7] are also another barrier to many of the legacy applications in house. Virtualization also incurs a cost to performance, due to extra process boundaries to serialize data to and from.

In the presence of automated deployment scripts, one may suggest that re-deploying the application would be an easier way to solve these issues. However, automatic deployments still require core shared services (e.g., COM+, and Internet Information Services), and application pools to be shut down, introducing disruptions to other teams that also have tight schedules and cannot afford constant downtimes. While we highly believe and support automated deployments, in the context of multiple on-going software projects, they simply overwrite the existing configurations without revealing which ones were incorrectly configured or changed by other teams. This becomes an environment where "the last team to deploy wins." And it is our experience that automated deployment scripts to not encompass all configurations needed in order to successfully host an enterprise n-tier software application. Automated deployment scripts also require expertise and knowledge to understand. The three projects under study had created semi-automated build scripts, consisting of Microsoft Installer files (msi) or simple Visual Build scripts. The word "semi-automated" is used to reflect that these scripts do not pull the source code from any source control repository. Instead these scripts copied existing compiled assemblies and files to desired locations. They also did not set server

configurations, which were left to be done manually. It is fair to say that deployment to the staging environments was mostly manual, done by a person following a set of written step-by-step instructions. In many occasions, especially after deployments, parts of the application would simply not function due to missed or changed configurations in the servers.

At a later stage, a fulltime build automation expert was hired. The build scripts were enhanced to include core environment configurations, but the support for such configuration is limited in many commercially available tools (such as Visual Build and NAnt). Although further automation of the deployment process was helpful, it could not completely address the situation as many project teams did not have the budget or expertise to build automated deployments. The bottom line is that servers were still being changed manually.

Almost all system were being developed using object-oriented design and analysis methodologies. Subsystems were created to foster reusability. But such reusability creates dependencies between external objects, services, and database objects, amongst others. Project teams become consumers of components that are not deployed and configured by them. The delivery timelines of reused components may or may not be in-line with the consumers. It becomes important to validate that dependencies are addressed in staging environments.

IV. Literature Review

Lents and Bleizeffer [1] describe an ethnographical study of eight enterprises to understand the sources of IT environment complexity in the design of middleware software. They focused on organizations, user roles, and technologies involved in the life-cycle of applications. The authors found that software deployments differ depending on the environment being targeted. "*Deployment into unit test environments is usually informal while deployment into production is a highly rigorous process.*"

These authors [1] found that although deployment was typically done in an automated fashion, most enterprises used ad-hoc methods for deploying small changes. Deployment responsibilities also shift to the hands of external groups (such as change control) for higher staging environments. A point of complexity identified was the supporting software that surrounds software applications. Dependencies on databases, transaction and messaging servers, clients, and source control must be integrated well with applications. The authors did not provide suggestions on how to eliminate this bottleneck; instead they focused on design considerations to minimize complexity.

Ambler talks about the use of "sandboxes" as being an Agile best practice [8]. Sandboxes are integration staging environments that become more controlled as they move up in the release lifecycle. But still it is easy to automatically detect the issues when they occur using sandboxes.

Beck, the original author of eXtreme Programming (XP), advocates that software needs to be integrated and tested early and often [9]. Continuous integration paired with a suite of NUnit tests are used to achieve this goal. Eckstein [11] states that each Agile team must have someone capable and responsible for integration and configuration management. Suggestions to support configuration management include source control versioning and baselining with continuous integration tools. But as we mentioned in Section III, automation alone does not resolve the underlying issue in contexts of multiple n-tier integrated project teams with different schedules and deployment practices.

A taxonomy of existing SCM systems by Conradi and Westfechtel [5] focus on the inner processes of source control versioning models. To the best of our knowledge, we were unable to find any related literature that addresses the automated verification of staging environment configuration as a way to consolidate configurations.

V. Test Strategy: Environment Configuration Testing

A. Action Planning

To address our first goal - (1) how can we validate that the environment configurations our software application depends on have been correctly applied to staging environments - we designed a test strategy that ensures certain properties of the deployment environments. Our automated test strategy has the following aspects: (1) definition of a meaningful and effective test adequacy criterion, (2) test oracle, (3) choice of test tool, and (4) an error seeding (mutation) technique.

Test adequacy criterion: Well-known black-box or white-box test adequacy criteria (e.g., line or decision coverage) do not fit the non-functional nature of our problem domain (i.e., staging instabilities). Thus, a fault-based testing criterion proved to be the best fit. A set of most common faults were identified by the interviewed developers and the lead tester. As a result, the initial test suite needed to cover:

- Folder permissions
- Database (SQL Server) stored procedures permissions
- Availability of required services
- COM+ DLL registration and identification
- Internet Information Services (IIS) settings
- Network groups, machine users, and machine groups
- Database (SQL server) users and roles memberships.

Test Oracle: Before our involvement, all teams were required to maintain an up-to-date "Disaster Recovery Plan System Manual" (DRP). This document was one of the few documents kept relatively up-to-date in source control due to auditing requirements.

This manual provides a high-level explanation of how to configure each individual application. The DRP did not list some required configurations, e.g., required database permissions, and folder permissions. Outdated or missing configurations were often gathered from deployed server configurations. External dependencies, such as services, stored procedures, and database users were gathered through code inspections.

Chosen test automation tool: NUnit version 2.2.6 was chosen as the test tool. NUnit was already an approved tool

for testing in the company, which meant that a long bureaucratic approval process from the technical enterprise team could be avoided. It was important to follow the path of least resistance due to existing contextual antipathy of management towards test automation.

Error seeding (mutation): We planned to manually inject defects into the configurations (during mutation testing only) to assess the fault detection effectiveness of the test suite. For example, deleting a folder permission, or stopping a service. The approach and results are presented in Subsection C.

B. Action Taking

In this paper, we refer to test fixtures as parts of the code needed in order to run the four phases of a test case in the NUnit framework (i.e., set up, exercise, verify, and tear down). Test cases were grouped into ten test fixtures based on the type of configuration they were trying to validate: (1) folder permissions, (2) database stored procedures permissions, (3) services availability, (4) COM+, (5) universal data links, (6) web configuration files, (7) IIS settings, (8) network and machine user membership, (9) global assembly cache, and (10) database security. These test fixtures were coded with generic *private* functions that could "exercise" a predefined behavior. For example, check if a folder has been granted a determined set of permissions. This generic function takes as input a folder path, and the permission set.

Test cases executing a call to these generic functions were passing in specific test inputs and performing assertions to "verify" that the return values match the expected values. Test inputs and expected return values were declaratively specified in XML files loaded during the test "set up."

Configurations are environment specific. They refer to different server paths, different domain accounts, and different namespaces for components and services. Before running, a test case reads an environment ID. Test inputs and expected values (oracle) are grouped by environment ID in each test-fixture-specific XML file. A "helper" class was created to assist in reading these declarative test inputs (making extensive use of the XPath, XML Path Language). This provided an easy way to point the test suite to the desirable environment.

As an example, a folder used for downloading attachments in a web application is given a tag (File name=TAG). This folder has a *path* in the Development staging environment (DEV) and it needs *modify permissions* granted to a network user called *userDEV*. The folder path acts as input while the permission and user pair are the expected values (oracle) for the test case execution. A folder that serves the same purpose in the Production environment (PROD) is given the same tag (File name=TAG). In PROD, this folder has a different *path* and it needs *modify permissions* granted to a different network user called *userPROD*.

Without the declarative XML inputs and expected values, we would need to code one test cases per staging environment (in our context at least five) to test that such download folder (TAG) has the correct permissions. Functionally speaking, the code executed to check the folder permission is the same for different test cases (a download folder, a temporary folder). We simply need a different set of test input/oracle for the different folders and different staging environments. This declarative XML approach also allowed us to create a living catalog of the most important environment configurations needed to run our applications – which addressed our third research goal – (3) a way to consolidate configuration management information.

Figure 2 shows an example of the XML input (sensitive data has been either replaced by "xxx"). The *expectedValue* node contains the expected result for the test case execution and is used for assertions in NUnit.

```
<?xml version="1.0" encoding="utf-8"?>
<TestConfiguration>
<Name>Verify File Permission</Name>
<Description><![CDATA[Verifies that the file
permission matches the expected value.]]>
</Description>
<Environment name="DEV" >
    <Server name="xxx">
    <BuildPackage name="xxx">
        <File name="TAG">
        <Path>\\xxx</Path>
        <expectedValue>
            <Type>FilePermission</Type>
            <Value>Modify</Value>
            <AccountName>userDEV</AccountName>
            <Check>True</Check>
        </expectedValue>
        </File>
    </BuildPackage>

    </Server>
</Environment>
<Environment name="PROD" >
    <Server name="xxx">
    <BuildPackage name="xxx">
        <File name="TAG">
        <Path>\\xxx</Path>
        <expectedValue>
            <Type>FilePermission</Type>
            <Value>Modify</Value>
            <AccountName>userPROD</AccountName>
            <Check>True</Check>
        </expectedValue>
        </File>
```

Figure 2 - Example of the XML test case inputs and expected output.

C. Preliminary Evaluation

After the test suites were implemented, we manually changed (mutated) the environment and checked if our tests would find all changes. Table II illustrates a subset of the mutants introduced and the results of the mutation testing. Our test cases found (killed) all the mutants (seeded faults) for every execution.

Table II. Example of mutants

Category	Mutant Generation (Error seeding)
Folder Permissions	Using windows explorer, we removed or changed permissions.
SQL Server stored procedures permissions	Using Microsoft SQL Server Management Studio, we removed permissions.
Availability of required services	Using Microsoft Computer Management Console, we stopped, disabled, or paused the services.
COM+ DLL registration and identification	Using Microsoft Component Services, we deleted or disabled the components. Also we changed the identities being tested.

After our mutation evaluation, we executed our test suite against the development staging environment (the most instable of the staging environments). The test suite successfully found at least half dozen missing permissions for stored procedures and file system folders in the first day of use. Such discovery would have taken many man-hours to diagnosed if done manually, and caused many hours of down time to the systems that needed these permissions. In

summary, the test suite proved to be effective in detecting both seeded and real environment configuration issues.

D. Periodic Execution of the Automated Tests

It is important that tests are run frequently so that errors are caught in a timely fashion. It is also important that test execution be made visible to appropriate stakeholders. The lack of frequency and visibility is seems as one of the main root causes for test automation failures [10, 11]. Beck suggests that someone in the team must be responsible for executing tests frequently and for publishing the results to all team members [9]. In order to address our second goal - (2) if our application starts failing after other teams deploy to the same staging environment, how can we quickly validate that this is not due to missing or changed configurations – and to address the above concerns, we decided to automate the execution of the test suites on a scheduled basis, to automatically publish the results in the intranet and to pro-actively send emails to interested parties after each test execution.

NUnit provides console commands to execute test suites and to export test run results into XML files. A batch file was created to call the environment configuration test suite. Microsoft Task Scheduler was used to execute this batch file at 6:00 AM.

To provide an easier, more readable presentation of the results, an XSL style sheet was generated to display the results in an "Environment Forecast" webpage.

A "sunny" forecast indicates that all tests are passing. A "mostly sunny" with occasional clouds forecast indicates that less than 25% are failing. A "cloudy' forest indicates that 25-50% are failing. Finally, a "stormy" weather is displayed if more than 50% of the tests are failing. To provide further information, data logged during tests (such as the parameters being tested) is exported to a text file. A link to this detailed log file is displayed at the bottom of the "Environment Forecast" web page. Once the test run was completed, e-mails were sent with a link to the results webpage to interested individuals (project lead, testing lead, and developers).

At first, we only had access to run the tests against the development and testing staging environments. We executed these tests for a period of 6 weeks. The test results provided improvements to the following areas:

- Database security: many stored procedures had the incorrect level of permission. This was brought to the attention of the Enterprise Architecture team which was asked to revisit the security models and tighten up permissions.
- Network Folder security: many folders had more permissions than necessary in the development and test environments, which resulted in certain problems to only appear in higher staging environments, where security was more strictly enforced.
- Visibility into changes in database administrators: stored procedures permissions seemed to "go missing" every once in a while, and system roles get regrouped without teams being properly notified.

The long term affect of the configuration tests is discussed in the following section.

E. Technical Challenges And Impact

After seeing the benefits of the testing suites on the lower level staging environments, we inquired about scheduling them to run against higher staging environments. The team responsible for granting such permissions (enterprise testing) was at first resistant to give any special permissions to run these tests in higher staging environments.

Because NUnit was used as the test harness tool, the enterprise testing team first assumed the tests were developer unit tests, thus unfit to be on-going in more secured environments. To clarify this misconception, and to get their support going forward, we booked meetings with the enterprise team, inviting the management group as well to present the test suite and automation strategy. Members of that team were also included in the daily test-run e-mail notifications.

After the presentation, managers and the enterprise testing team became interested to see the source code. Access to the source code was granted to them. Soon after the demos, this enterprise team assigned a developer to validate the test suite in order to give the permission to run it in higher staging environments. During this review, the developer abstracted the XML declarative input approach and adapted our test suite to be able to run tests from multiple projects. We gained approval to run the tests in higher staging environments while the enterprise architectural team enforced that all other in-house projects specify their environment configurations in XML to also run in this test suite. These tests are now scheduled to run three times a day in all-staging environments, including Production. They can also be run on-demand by a group of users that have been granted special permissions, including the first author of this article.

A verbatim quote from an e-mail sent by the test lead states that "***before the configuration tests:*** *One or more testing environments would be unstable each day causing delays in development and testing. During a period of 3 months, environment outages were tracked and over 50 hours of development and testing time were lost and business confidence in the environments/systems was very low. In addition, identifying and addressing the cause of the outage required 2 to 5 people to get involved to search for and address the issue. The primary root cause was very difficult to determine.* ***After configuration test****: The suite initially focused on monitoring and "flagging" environment configuration changes. The tests were executed on a daily basis and on command. The impact of these tests is significant. Problem root cause identification is immediately available and resolution activities are focused (1 - 2 people) and usually are corrected in minutes. It was identified that 97% of all instability problems were associated with environment configuration and database issues/changes (not system code/functionality). Test environment downtime has been reduced to 0 - 10 minutes per week. Most importantly, overall organization confidence in the test environments has*

significantly increased - to the point where development and test results are trusted and used for benchmarking purposes."

VI. Lessons-Learned

NUnit proved to be an effective, flexible and easy to use testing harness for the execution of tests other than traditional unit tests.

By running environment configuration tests on an on-going basis, the root causes of environment instability and configuration problem areas started to disappear. Conflicting configurations were found during test executions. Dependencies on external components, such as services and databases were validated without manual intervention. Teams had to work together to ensure suitable resolutions to conflicts. Manual deployments still caused issues, but these issues were found in a timely manner, making individuals involved in manual deployments more cautions of changes to shared configurations.

The environment configuration necessary to run applications is no longer hidden in long documents in source control, or in deployment scripts. They are visible and readily available in a suite of automated tests. To provide quantitative insights to the usefulness of the automated build verification testing (BVT), some before- and after-BVT measures are provided below.

Before BVT	After BVT
• Over 50 hours of downtime in 3 months • Very low confidence • Outage required 2 to 5 people • Primary root cause was very difficult to determine.	• 0 - 10 minutes of downtime per week • 97% of all instability problems were associated with environment configuration • Focused (1 - 2 people) and corrected in minutes • Problem root cause identification is immediately available.

Last but not least, based on our experience, we make the following recommendations for testing practitioners wanting to introduce configuration testing practices:

Expect resistance. The truth is that automated testing requires special permissions for higher level staging environments. So admit it, get it working on a local environment where you do have enough security privileges and plan to do some convincing for getting appropriate privileges on higher staging environments. Don't take this resistance personally. But instead, nicely present the benefits of your tests to those being skeptical. They will see that it will actually help them do their job better too.

Keep maintainability in mind. This is one of the causes of why automated tests get abandoned by many teams [9]. Make it easy to add new test cases to your suite without having to change or redeploy code.

Keep it simple and use free tools. Create a simple testing strategy that leans on existing free test tools, creating generic test drivers that use declarative descriptions as test case inputs for future re-use.

Automate as much as possible. In addition to test automation, automate the test execution as well preferably using existing OS tools, such as Microsoft Task Scheduler and expose the test execution results in an easy to access location, such as a simple website.

VII. Conclusion

The introduction of a suite of automated environment configuration tests that verify the environment on demand has helped us identify many deployment dependencies. It has also assisted several of our partner teams resolve configuration issues in staging environments in a timely manner. It also abstracted environment configuration management into a live and evolving test suite that shows failure and it is easily maintainable. The creation of a simple test strategy that leaned on existing free test tools, with generic test drivers that use declarative descriptions as test case inputs enabled future re-use. This reuse escalated the use of this test strategy to an enterprise testing framework, proving the need and usefulness of the techniques we implemented.

Acknowledgements

The authors would like to thank Jim King and Bryan Schultz for their support.

References

[1] J. L. Lentz and T. M. Bleizeffer, "IT ecosystems: evolved complexity and unintelligent design," *Proc. of 2007 Symposium on Computer Human Interaction for the Management of information Technology*, 2007.

[2] Disruptive Library Technology Jester, "Traditional Development/Integration/Staging/Production Practice for Software Development," *On-line: http://dltj.org/article/software-development-practice*, Downloaded on Sept. 2009.

[3] MSDN, "What is the Staging Environment?," *On-line: http://msdn.microsoft.com/en-us/library/ms942990.aspx*, Downloaded on Sept. 2009.

[4] H. D. Frederiksen and L. Mathiassen, "A Contextual Approach to Improving Software Metric Practices," *IEEE Transactions on Engineering Management*, vol. 55, no. 4, pp. 602–616, 2008.

[5] R. M. Davison, M. G. Martinsons, and N. Koch, "Principles of Canonical Action Research," *Information Systems Journal*, vol. 14, no. 1, pp. 65–89, 2004.

[6] The question man, *On-line: http://www.ajr.org/article_printable.asp?id=676*, Downloaded: July 2009.

[7] Microsoft App-V team blog, "On-line: http://blogs.technet.com/softgrid/archive/2007/09/27/list-of-applications-that-can-be-virtualized.aspx," Downloaded: March 21, 2010.

[8] S. Ambler, "Development Sandboxes: An Agile "Best Practice" " *On-line http://www.agiledata.org/essays/sandboxes.html*, Downloaded on Sept. 2009.

[9] K. Beck, *Extreme Programming*: Addison-Wesley 2004.

[10] S. Berner, R. Weber, and R. K. Keller, "Observations and lessons learned from automated testing," *Proc of the Int Conference on Software Engineering*, pp. 571–579, 2005.

[11] M. Fewster and D. Graham, *Software Test Automation*: ACM Press, 1999.