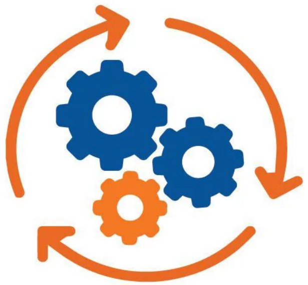
Chapter 10

Test Program Review and Assessment



Lecturer: Nguyen Thanh Quan (MEng)
Email: tg_nguyenthanhquan_cntt@tdtu.edu.vn

Content

1. **Test Program Lessons Learned - Corrective Actions and Improvement Activity**
2. **Test Program Return on Investment**

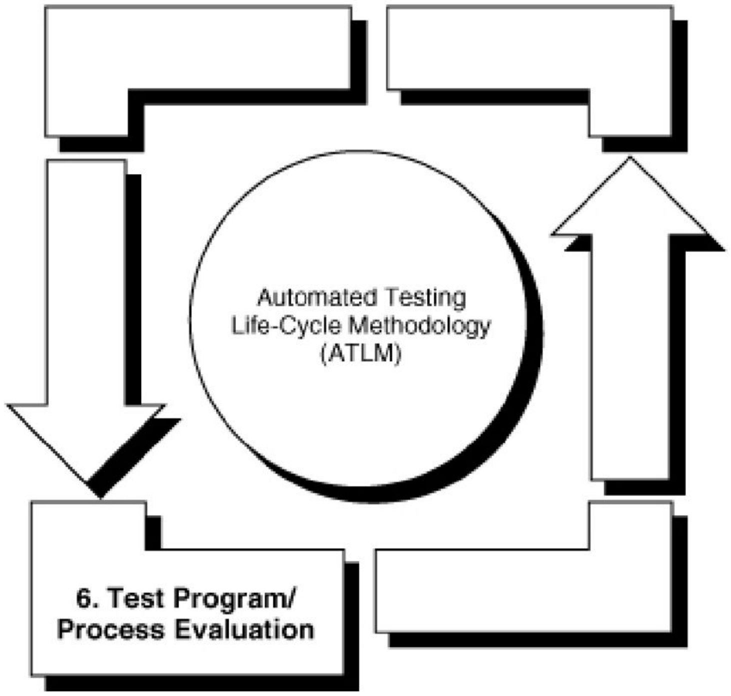
Introduction

Improvements in quality always and automatically result in reductions in schedules and costs, increases in productivity, increases in market share, and consequently increases in profits —W. Edwards Deming

- The test team needs to **review the performance** of the test program to determine where enhancements can be implemented during the next testing phase or on the next project.
- The test team **collects** various **test metrics,** including many during the test execution phase.
- the review of test metrics should **conclude with suggested adjustments** and **improvement recommendations.**

Introduction

- Should **document** the activities that it performed well
- Should **record lessons learned** throughout each phase of the test program life cycle.
- **Alter the detailed procedures** during the test program.
- Propose **corrective actions** once the project is complete.
- Should **adopt an ongoing iterative process** focusing on lessons learned.



Test Program Lessons Learned

Corrective Actions and Improvement Activity

- Although a quality assurance (QA) department is responsible for conducting to verify the processes and procedures => **the test team can conduct its own test program analysis.**
- Test personnel should **continuously review the QA audit findings** and follow suggested corrective actions.
- The metrics helps **pinpoint problems** that need to be addressed.
- Focus on **lessons learned** and issues pertaining to the development life cycle.

Test Program Lessons Learned

Corrective Actions and Improvement Activity

- **Too late** to implement corrective action if **lessons learned and metrics** evaluation sessions **take place at the end of a SDLC.**
- Document **lessons learned** at this late stage may benefit subsequent test efforts => **intranet** or requirement management **tool** database can be used.
- **Lessons learned** records should be viewed as improvement opportunities.

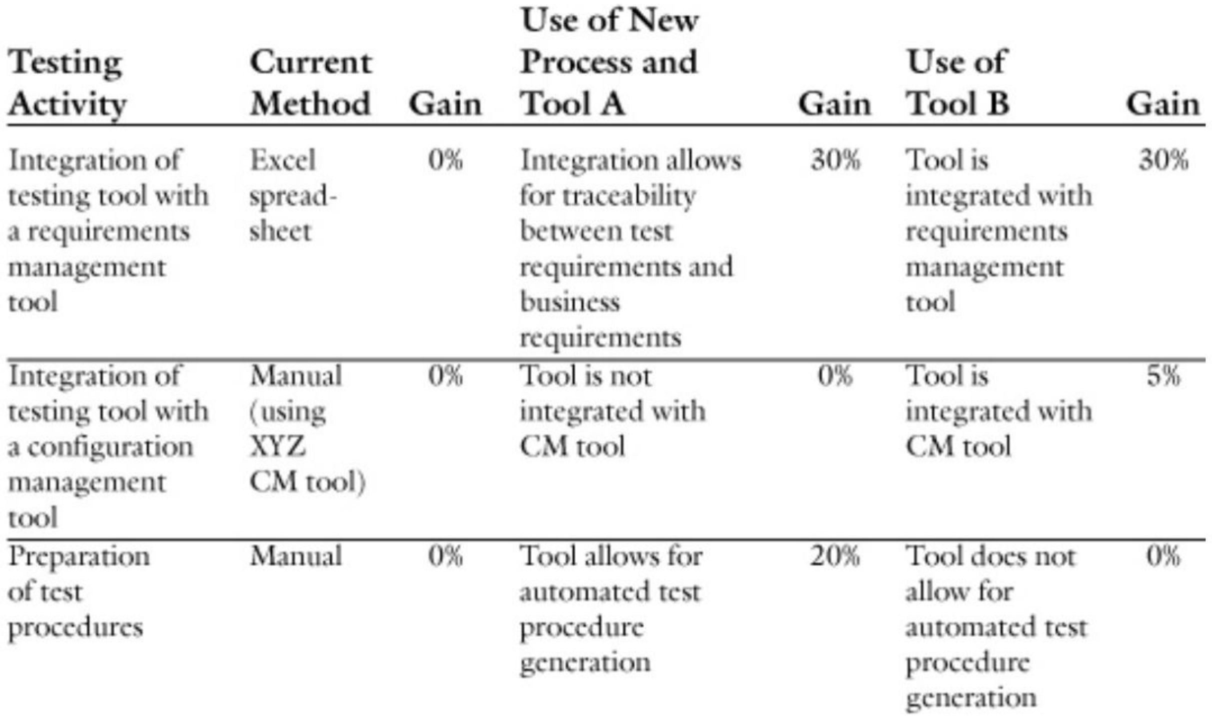
=> should not list the names of the individuals, should include corrective actions

- Sometimes a corrective action initially **sounds** like the perfect **solution** to a problem, **but** further analysis can show **no specific gain.**

=> be careful when coming up with improvements to testing lessons learned.

Test Program Lessons Learned

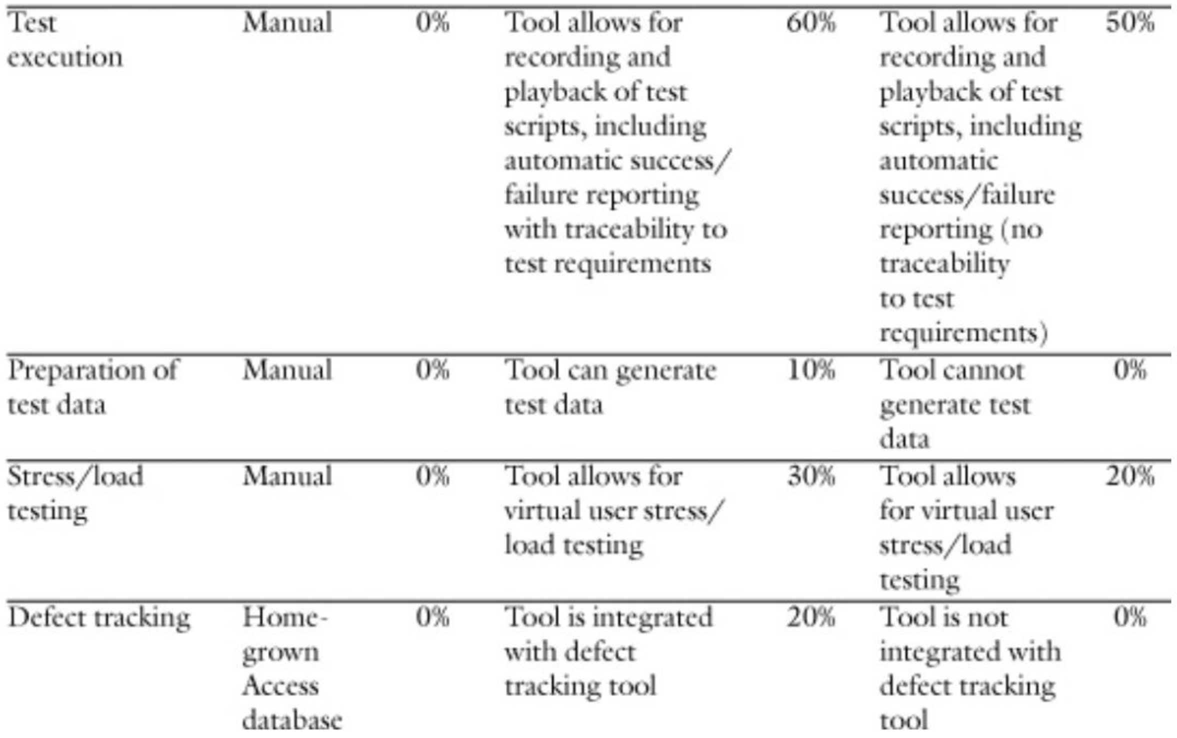
Test Program Improvements



Testing Activity	Current Method	Gain	Use of New Process and Tool A	Gain	Use of Tool B	Gain
Integration of testing tool with a requirements management tool	Excel spread-sheet	0%	Integration allows for traceability between test requirements and business requirements	30%	Tool is integrated with requirements management tool	30%
Integration of testing tool with a configuration management tool	Manual (using XYZ CM tool)	0%	Tool is not integrated with CM tool	0%	Tool is integrated with CM tool	5%
Preparation of test procedures	Manual	0%	Tool allows for automated test procedure generation	20%	Tool does not allow for automated test procedure generation	0%

Test Program Lessons Learned

Test Program Improvements



Test execution	Manual	0%	Tool allows for recording and playback of test scripts, including automatic success/failure reporting with traceability to test requirements	60%	Tool allows for recording and playback of test scripts, including automatic success/failure reporting (no traceability to test requirements)	50%
Preparation of test data	Manual	0%	Tool can generate test data	10%	Tool cannot generate test data	0%
Stress/load testing	Manual	0%	Tool allows for virtual user stress/load testing	30%	Tool allows for virtual user stress/load testing	20%
Defect tracking	Home-grown Access database	0%	Tool is integrated with defect tracking tool	20%	Tool is not integrated with defect tracking tool	0%

Test Program Lessons Learned

Corrective Actions and Improvement Activity

- The test team must **match the actual implementation** of the test program to the initially planned criteria.
- Some questions:
  - Was the **documented** test process **entirely followed?**
  - If not, which **parts** were **not implemented? Why?**
  - **What** was the **effect** of **not following the process?**
  - Have the test program **goals been met?**
  - Were **strategies** implemented **as planned? If not, why not?**
  - Were the **defect prevention** activities **successfully implemented?**
  - If not, which defect prevention activity step was **omitted and why?**

Test Program Lessons Learned

Corrective Actions and Improvement Activity

- **Some questions:**
  - Were the **risks** made **clear** and **documented in advance?**
  - Were **lessons learned activities** conducted throughout the life cycle and **corrective actions taken**, when appropriate?

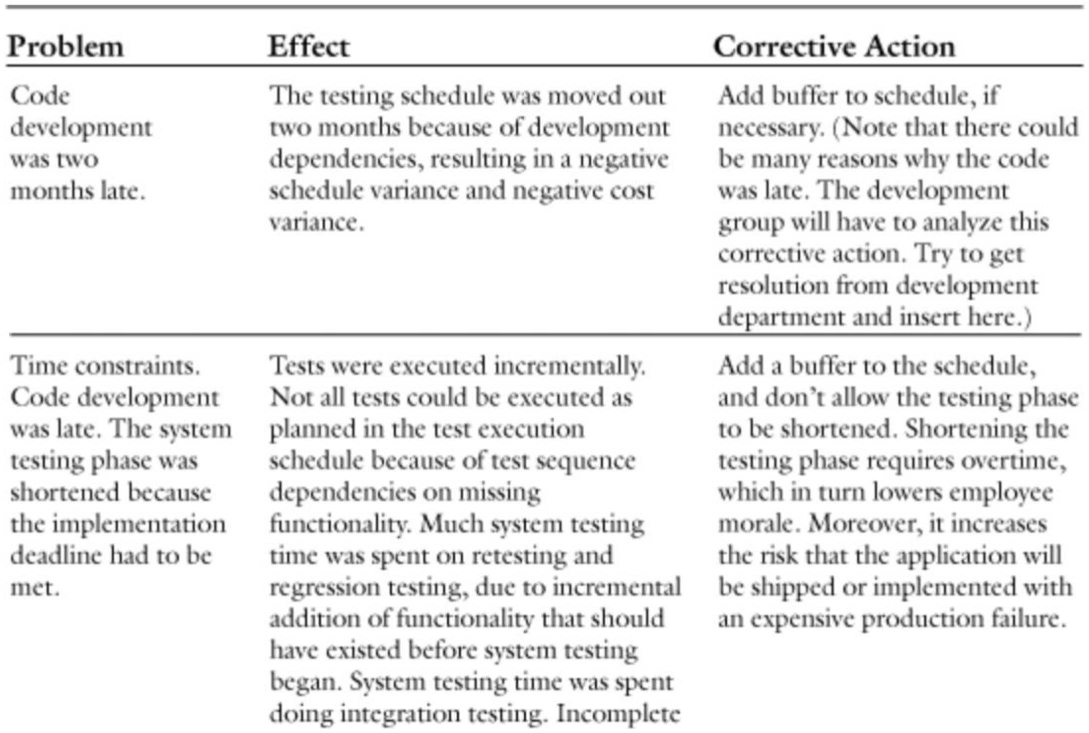
=> At the conclusion of a test program, test personnel should **evaluate** the **effectiveness of the defined processes.**

=> The test team should **find out** whether **the same mistakes** were **repeated** and confirm whether any suggested improvement opportunities were bypassed.

=> Test program analysis can identify problem areas, potential corrective actions, review the effectiveness of implemented corrective action changes.

Test Program Lessons Learned

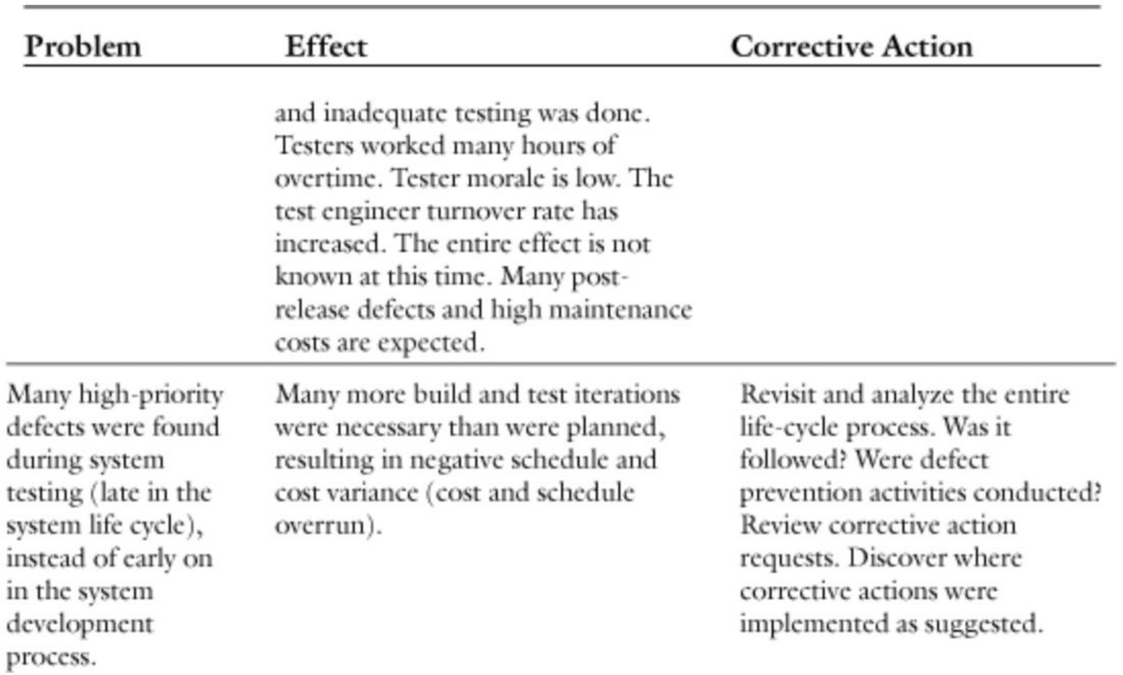
Examples of Schedule-Related Problems and Corrective Actions



Problem	Effect	Corrective Action
Code development was two months late.	The testing schedule was moved out two months because of development dependencies, resulting in a negative schedule variance and negative cost variance.	Add buffer to schedule, if necessary. (Note that there could be many reasons why the code was late. The development group will have to analyze this corrective action. Try to get resolution from development department and insert here.)
Time constraints. Code development was late. The system testing phase was shortened because the implementation deadline had to be met.	Tests were executed incrementally. Not all tests could be executed as planned in the test execution schedule because of test sequence dependencies on missing functionality. Much system testing time was spent on retesting and regression testing, due to incremental addition of functionality that should have existed before system testing began. System testing time was spent doing integration testing. Incomplete	Add a buffer to the schedule, and don't allow the testing phase to be shortened. Shortening the testing phase requires overtime, which in turn lowers employee morale. Moreover, it increases the risk that the application will be shipped or implemented with an expensive production failure.

Test Program Lessons Learned

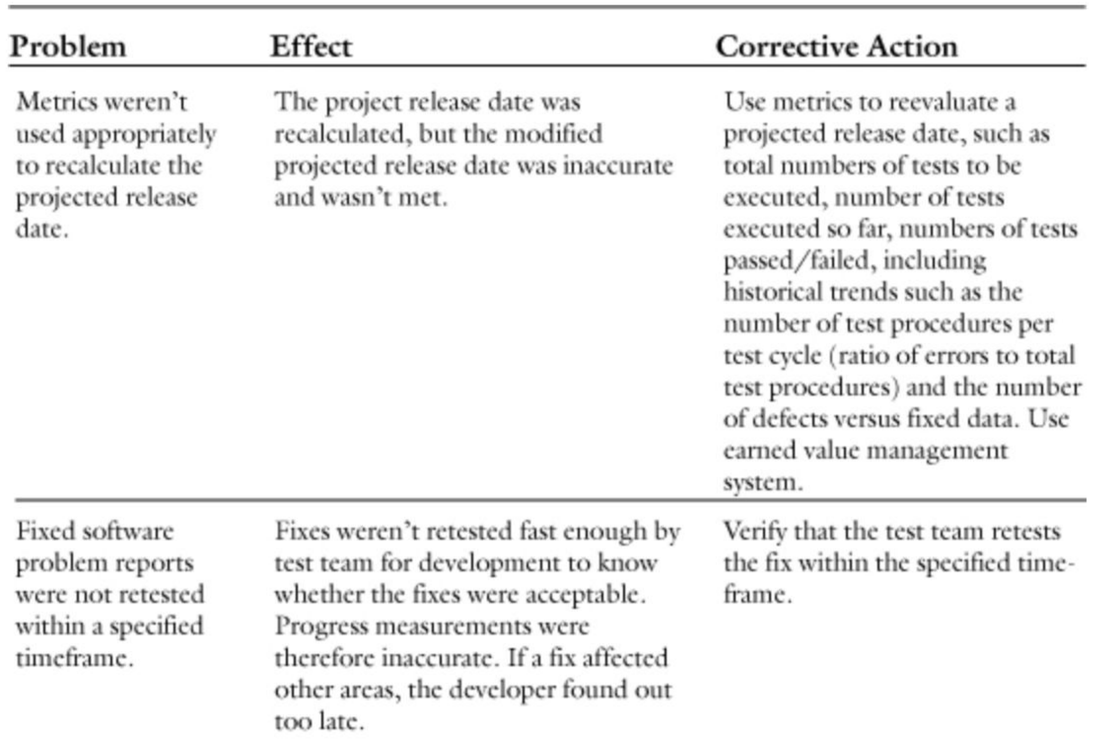
Examples of Schedule-Related Problems and Corrective Actions



Problem	Effect	Corrective Action
	and inadequate testing was done. Testers worked many hours of overtime. Tester morale is low. The test engineer turnover rate has increased. The entire effect is not known at this time. Many post-release defects and high maintenance costs are expected.	
Many high-priority defects were found during system testing (late in the system life cycle), instead of early on in the system development process.	Many more build and test iterations were necessary than were planned, resulting in negative schedule and cost variance (cost and schedule overrun).	Revisit and analyze the entire life-cycle process. Was it followed? Were defect prevention activities conducted? Review corrective action requests. Discover where corrective actions were implemented as suggested.

Test Program Lessons Learned

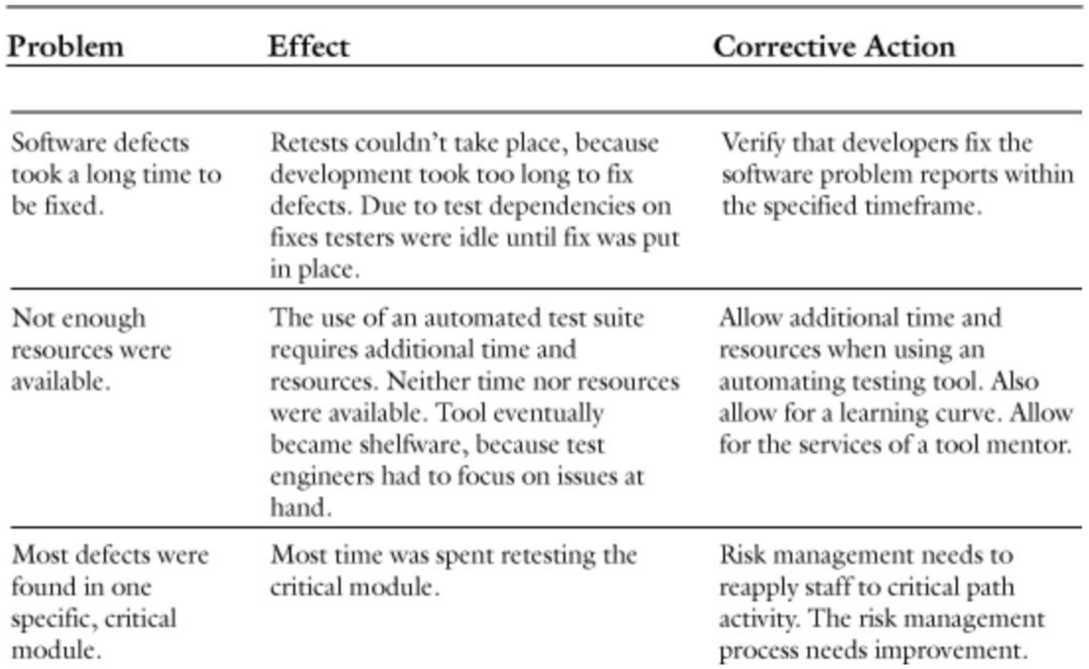
Examples of Schedule-Related Problems and Corrective Actions



Problem	Effect	Corrective Action
Metrics weren't used appropriately to recalculate the projected release date.	The project release date was recalculated, but the modified projected release date was inaccurate and wasn't met.	Use metrics to reevaluate a projected release date, such as total numbers of tests to be executed, number of tests executed so far, numbers of tests passed/failed, including historical trends such as the number of test procedures per test cycle (ratio of errors to total test procedures) and the number of defects versus fixed data. Use earned value management system.
Fixed software problem reports were not retested within a specified timeframe.	Fixes weren't retested fast enough by test team for development to know whether the fixes were acceptable. Progress measurements were therefore inaccurate. If a fix affected other areas, the developer found out too late.	Verify that the test team retests the fix within the specified time-frame.

Test Program Lessons Learned

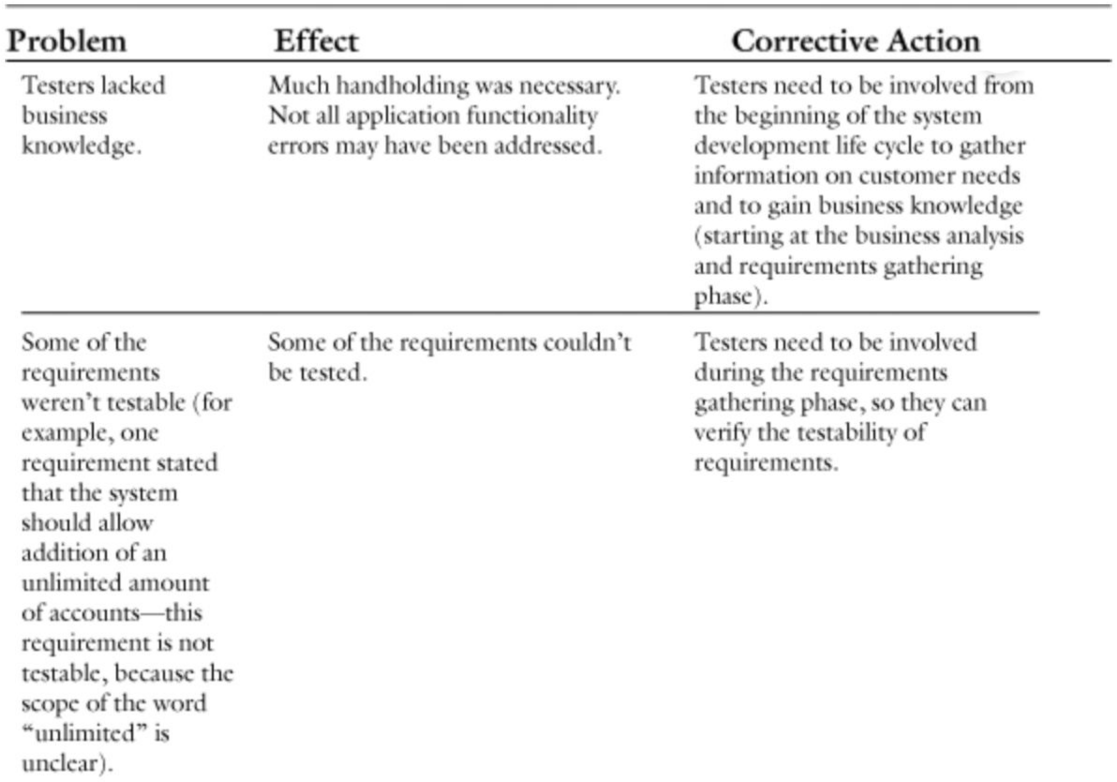
Examples of Schedule-Related Problems and Corrective Actions



Problem	Effect	Corrective Action
Software defects took a long time to be fixed.	Retests couldn't take place, because development took too long to fix defects. Due to test dependencies on fixes testers were idle until fix was put in place.	Verify that developers fix the software problem reports within the specified timeframe.
Not enough resources were available.	The use of an automated test suite requires additional time and resources. Neither time nor resources were available. Tool eventually became shelfware, because test engineers had to focus on issues at hand.	Allow additional time and resources when using an automating testing tool. Also allow for a learning curve. Allow for the services of a tool mentor.
Most defects were found in one specific, critical module.	Most time was spent retesting the critical module.	Risk management needs to reapply staff to critical path activity. The risk management process needs improvement.

Test Program Lessons Learned

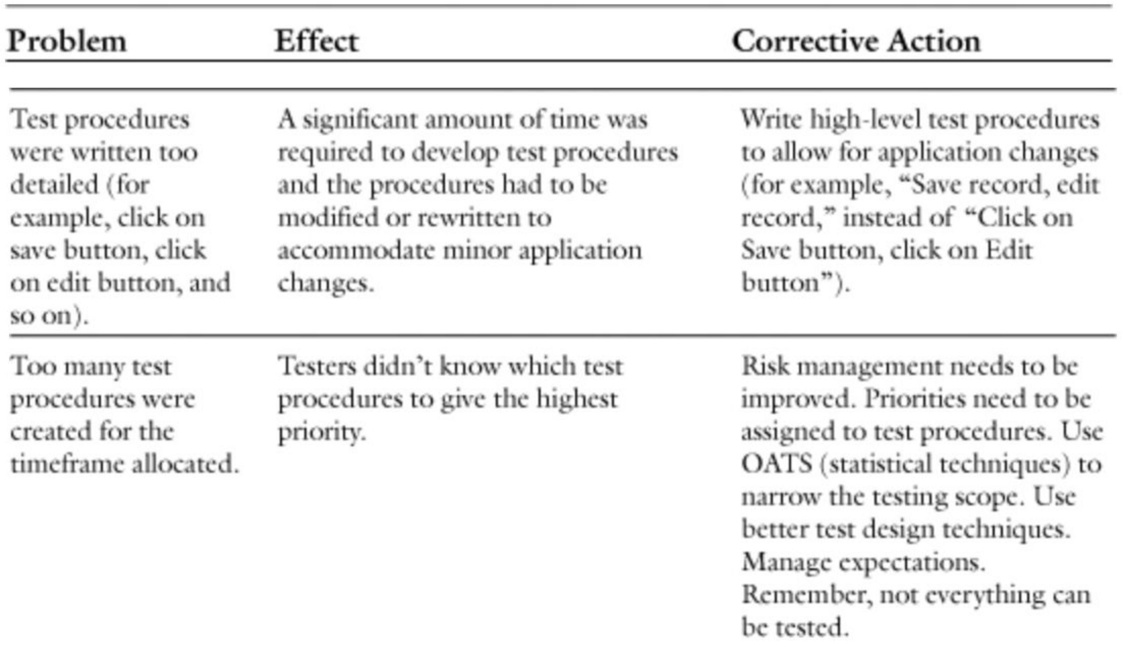
Examples of Test Program/Process-Related Problems and Corrective Actions



Problem	Effect	Corrective Action
Testers lacked business knowledge.	Much handholding was necessary. Not all application functionality errors may have been addressed.	Testers need to be involved from the beginning of the system development life cycle to gather information on customer needs and to gain business knowledge (starting at the business analysis and requirements gathering phase).
Some of the requirements weren't testable (for example, one requirement stated that the system should allow addition of an unlimited amount of accounts—this requirement is not testable, because the scope of the word "unlimited" is unclear).	Some of the requirements couldn't be tested.	Testers need to be involved during the requirements gathering phase, so they can verify the testability of requirements.

Test Program Lessons Learned

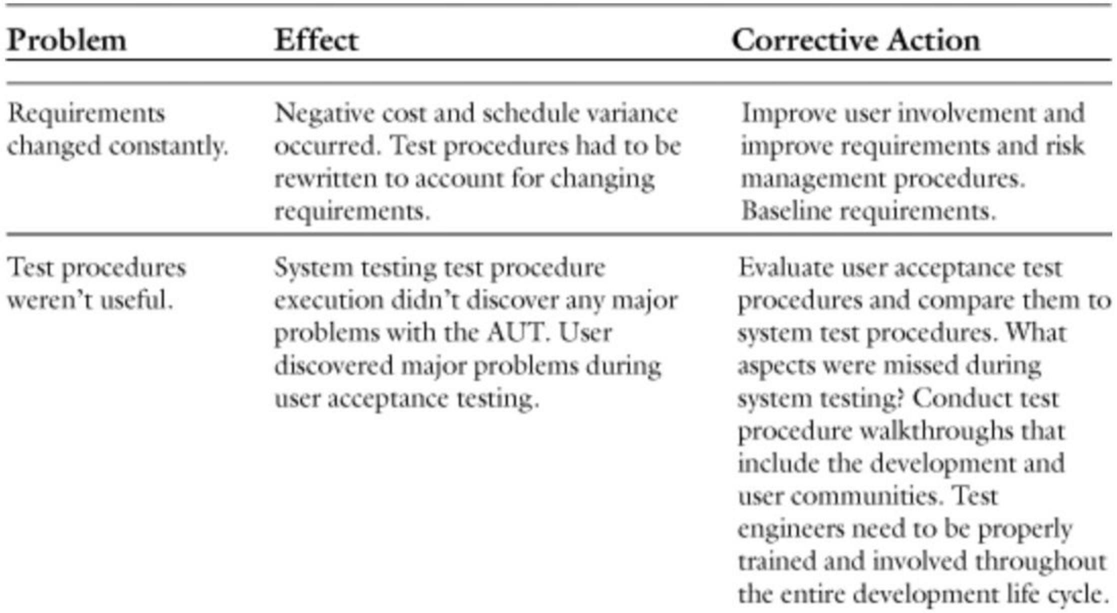
Examples of Test Program/Process-Related Problems and Corrective Actions



Problem	Effect	Corrective Action
Test procedures were written too detailed (for example, click on save button, click on edit button, and so on).	A significant amount of time was required to develop test procedures and the procedures had to be modified or rewritten to accommodate minor application changes.	Write high-level test procedures to allow for application changes (for example, "Save record, edit record," instead of "Click on Save button, click on Edit button").
Too many test procedures were created for the timeframe allocated.	Testers didn't know which test procedures to give the highest priority.	Risk management needs to be improved. Priorities need to be assigned to test procedures. Use OATS (statistical techniques) to narrow the testing scope. Use better test design techniques. Manage expectations. Remember, not everything can be tested.

Test Program Lessons Learned

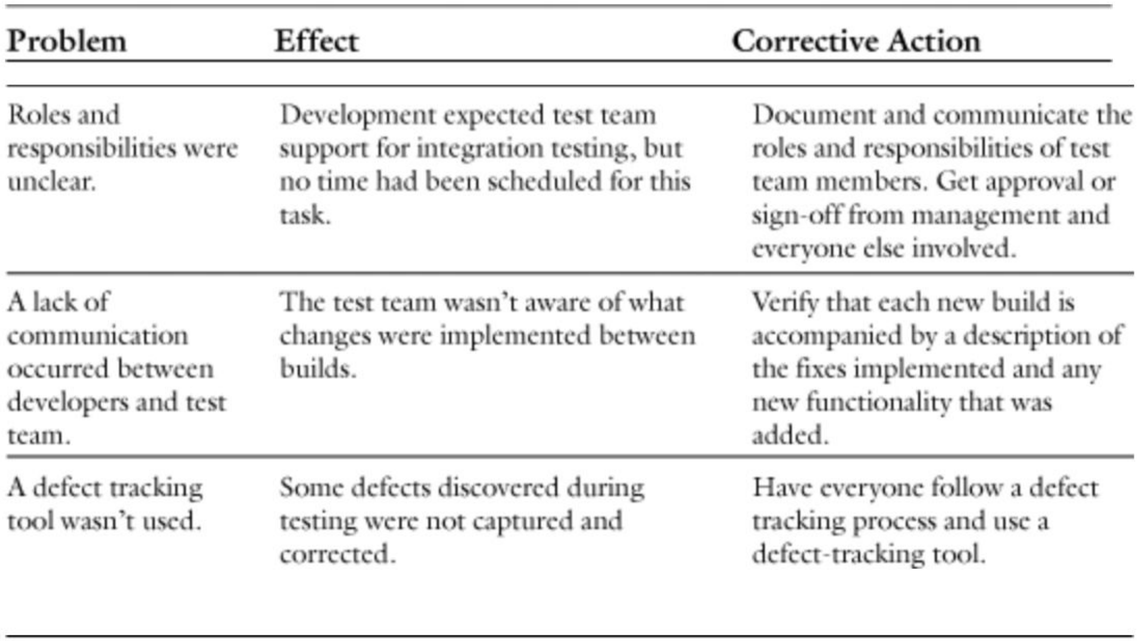
Examples of Test Program/Process-Related Problems and Corrective Actions



Problem	Effect	Corrective Action
Requirements changed constantly.	Negative cost and schedule variance occurred. Test procedures had to be rewritten to account for changing requirements.	Improve user involvement and improve requirements and risk management procedures. Baseline requirements.
Test procedures weren't useful.	System testing test procedure execution didn't discover any major problems with the AUT. User discovered major problems during user acceptance testing.	Evaluate user acceptance test procedures and compare them to system test procedures. What aspects were missed during system testing? Conduct test procedure walkthroughs that include the development and user communities. Test engineers need to be properly trained and involved throughout the entire development life cycle.

Test Program Lessons Learned

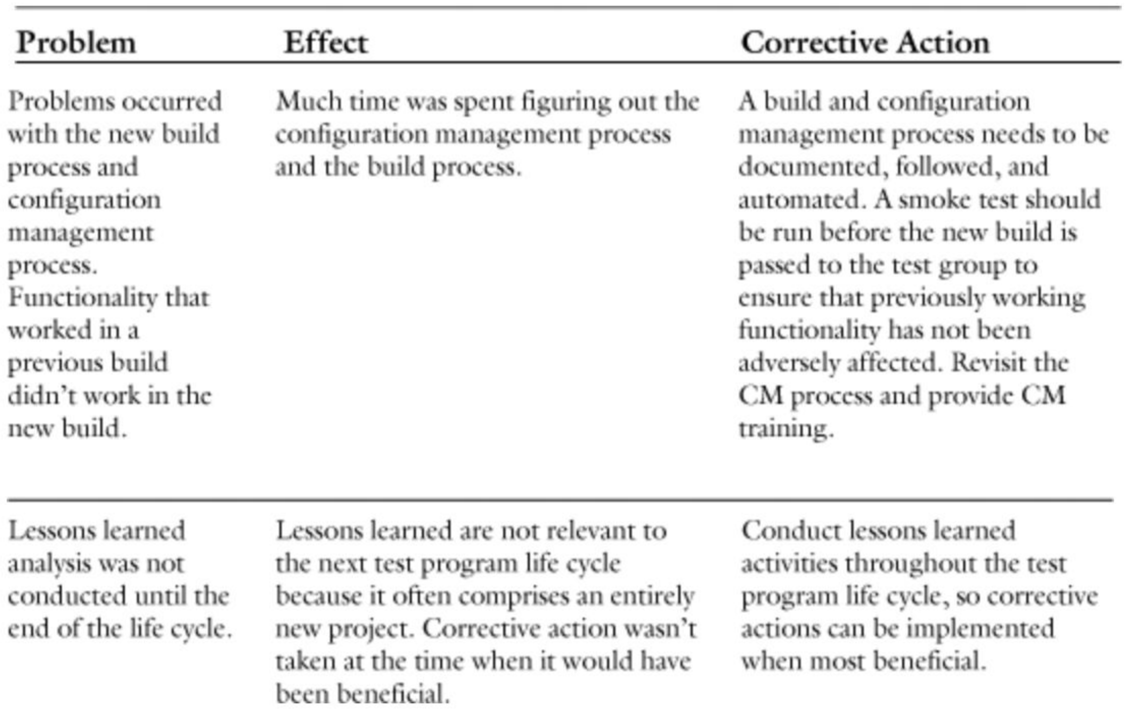
Examples of Test Program/Process-Related Problems and Corrective Actions



Problem	Effect	Corrective Action
Roles and responsibilities were unclear.	Development expected test team support for integration testing, but no time had been scheduled for this task.	Document and communicate the roles and responsibilities of test team members. Get approval or sign-off from management and everyone else involved.
A lack of communication occurred between developers and test team.	The test team wasn't aware of what changes were implemented between builds.	Verify that each new build is accompanied by a description of the fixes implemented and any new functionality that was added.
A defect tracking tool wasn't used.	Some defects discovered during testing were not captured and corrected.	Have everyone follow a defect tracking process and use a defect-tracking tool.

Test Program Lessons Learned

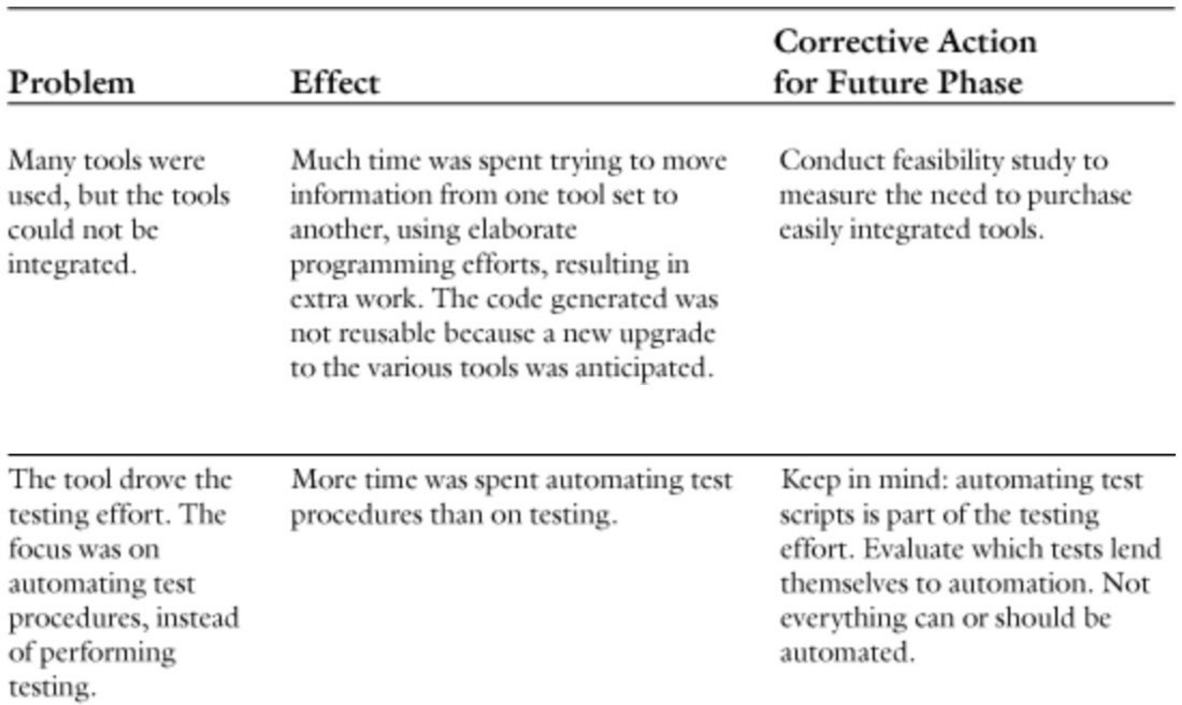
Examples of Test Program/Process-Related Problems and Corrective Actions



Problem	Effect	Corrective Action
Problems occurred with the new build process and configuration management process. Functionality that worked in a previous build didn't work in the new build.	Much time was spent figuring out the configuration management process and the build process.	A build and configuration management process needs to be documented, followed, and automated. A smoke test should be run before the new build is passed to the test group to ensure that previously working functionality has not been adversely affected. Revisit the CM process and provide CM training.
Lessons learned analysis was not conducted until the end of the life cycle.	Lessons learned are not relevant to the next test program life cycle because it often comprises an entirely new project. Corrective action wasn't taken at the time when it would have been beneficial.	Conduct lessons learned activities throughout the test program life cycle, so corrective actions can be implemented when most beneficial.

Test Program Lessons Learned

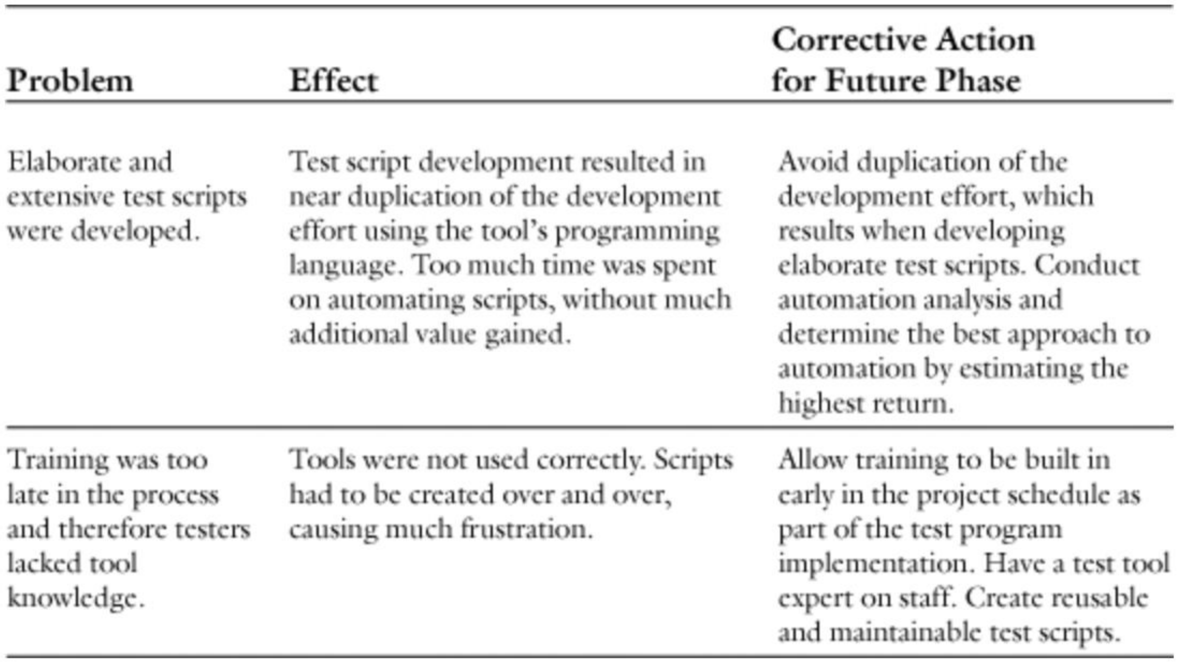
Examples of Test Program/Process-Related Problems and Corrective Actions



Problem	Effect	Corrective Action for Future Phase
Many tools were used, but the tools could not be integrated.	Much time was spent trying to move information from one tool set to another, using elaborate programming efforts, resulting in extra work. The code generated was not reusable because a new upgrade to the various tools was anticipated.	Conduct feasibility study to measure the need to purchase easily integrated tools.
The tool drove the testing effort. The focus was on automating test procedures, instead of performing testing.	More time was spent automating test procedures than on testing.	Keep in mind: automating test scripts is part of the testing effort. Evaluate which tests lend themselves to automation. Not everything can or should be automated.

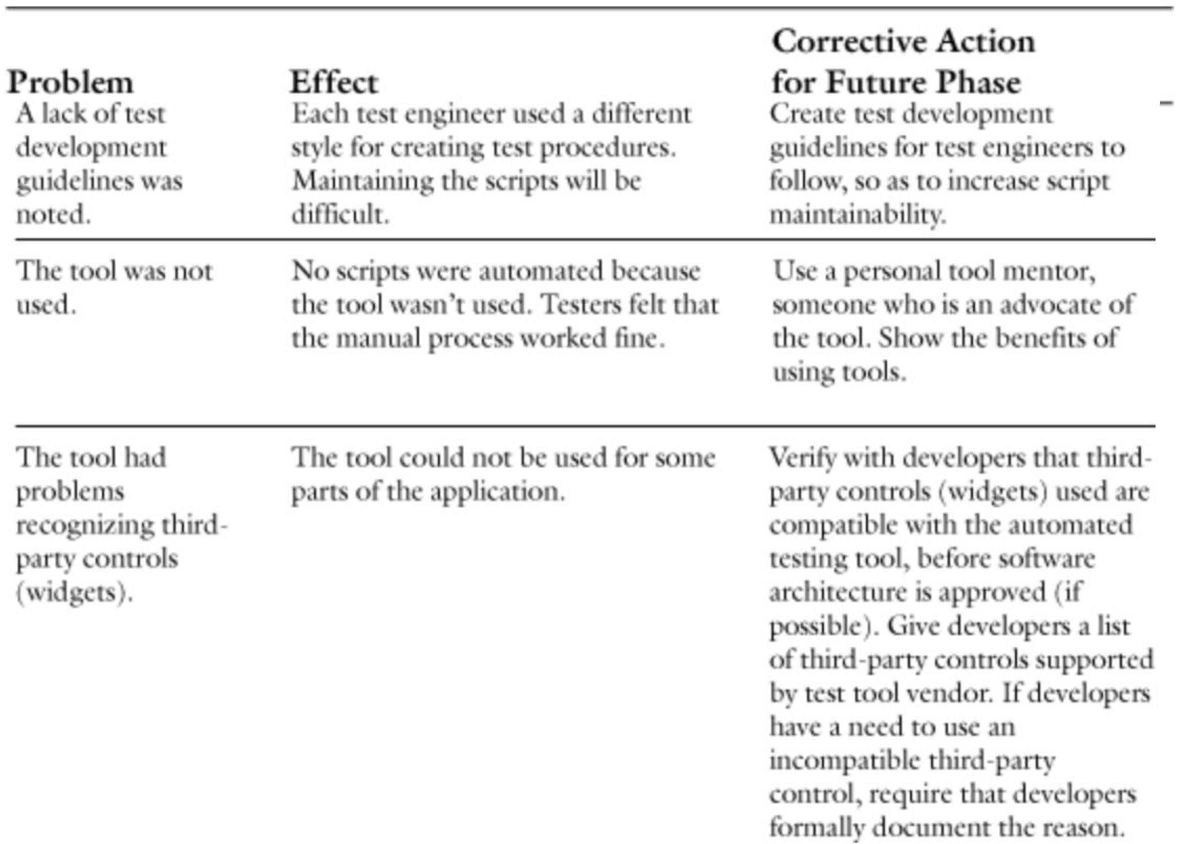
Test Program Lessons Learned

Examples of Test Program/Process-Related Problems and Corrective Actions



Problem	Effect	Corrective Action for Future Phase
Elaborate and extensive test scripts were developed.	Test script development resulted in near duplication of the development effort using the tool's programming language. Too much time was spent on automating scripts, without much additional value gained.	Avoid duplication of the development effort, which results when developing elaborate test scripts. Conduct automation analysis and determine the best approach to automation by estimating the highest return.
Training was too late in the process and therefore testers lacked tool knowledge.	Tools were not used correctly. Scripts had to be created over and over, causing much frustration.	Allow training to be built in early in the project schedule as part of the test program implementation. Have a test tool expert on staff. Create reusable and maintainable test scripts.

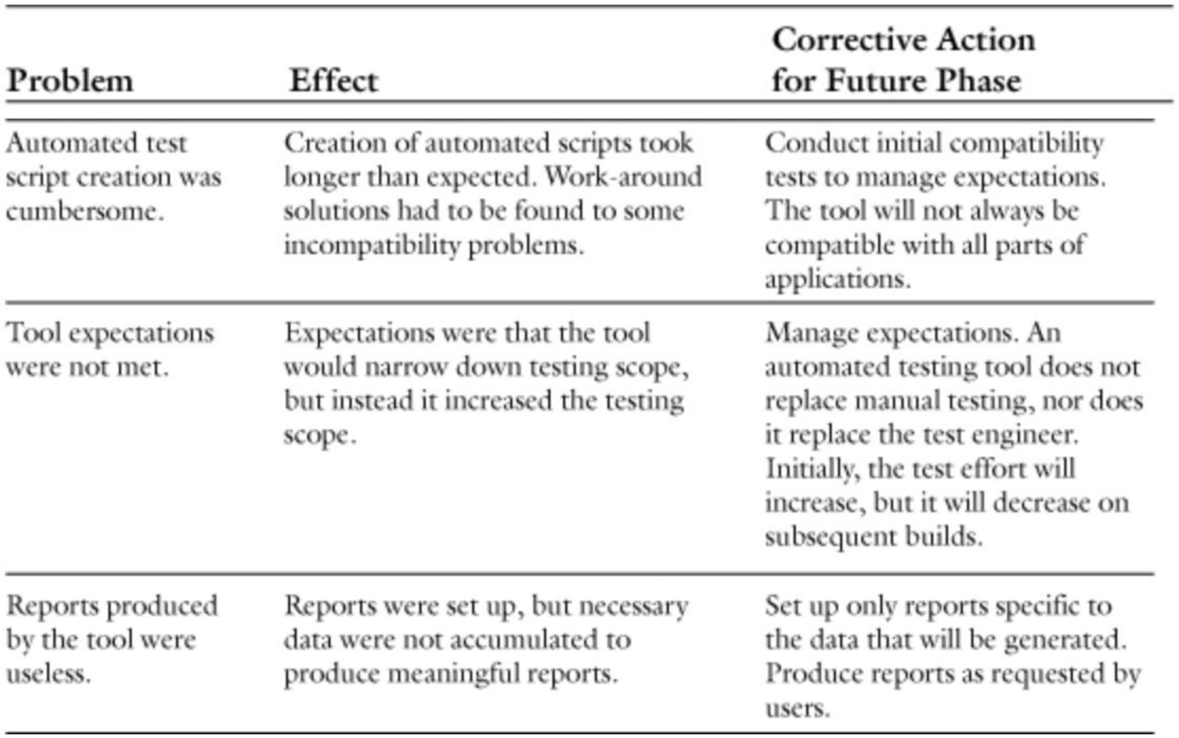
Test Program Lessons Learned



Problem	Effect	Corrective Action for Future Phase
A lack of test development guidelines was noted.	Each test engineer used a different style for creating test procedures. Maintaining the scripts will be difficult.	Create test development guidelines for test engineers to follow, so as to increase script maintainability.
The tool was not used.	No scripts were automated because the tool wasn't used. Testers felt that the manual process worked fine.	Use a personal tool mentor, someone who is an advocate of the tool. Show the benefits of using tools.
The tool had problems recognizing third-party controls (widgets).	The tool could not be used for some parts of the application.	Verify with developers that third-party controls (widgets) used are compatible with the automated testing tool, before software architecture is approved (if possible). Give developers a list of third-party controls supported by test tool vendor. If developers have a need to use an incompatible third-party control, require that developers formally document the reason.

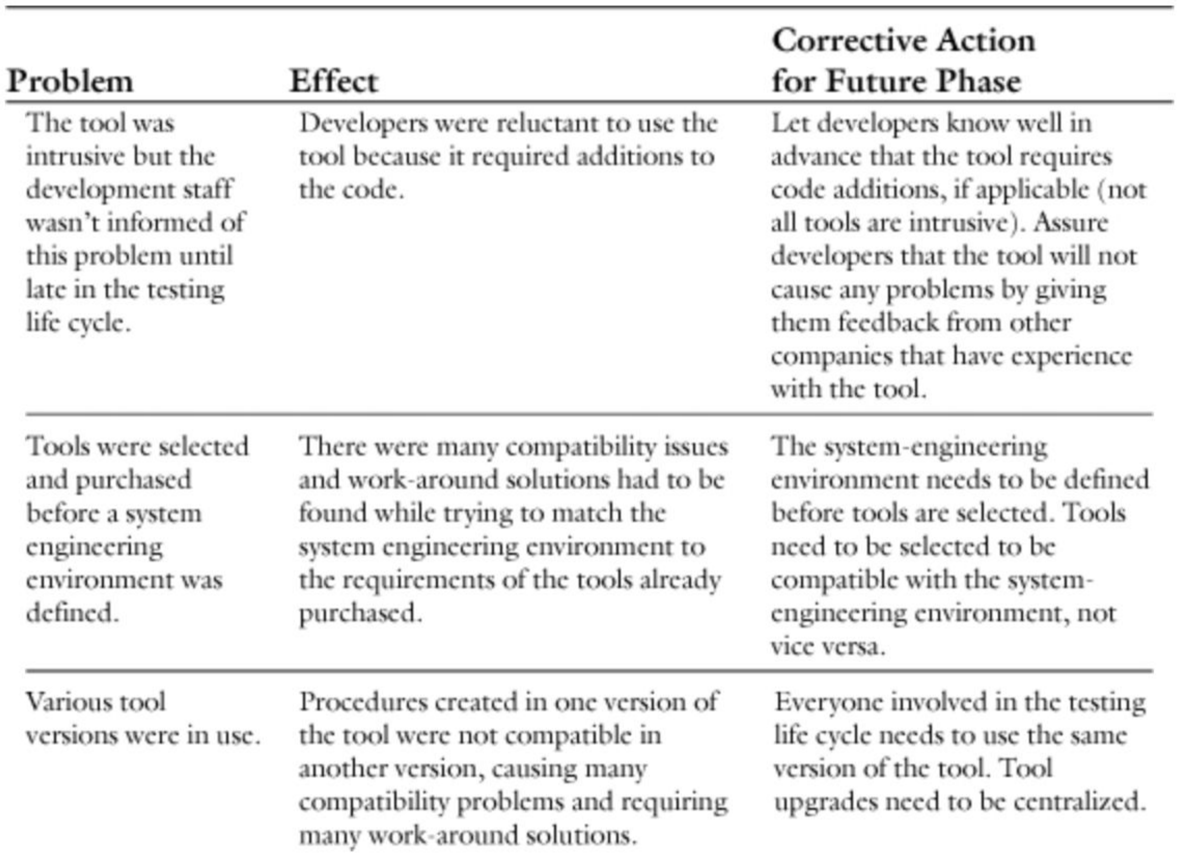
Test Program Lessons Learned

Examples of Test Program/Process-Related Problems and Corrective Actions



Problem	Effect	Corrective Action for Future Phase
Automated test script creation was cumbersome.	Creation of automated scripts took longer than expected. Work-around solutions had to be found to some incompatibility problems.	Conduct initial compatibility tests to manage expectations. The tool will not always be compatible with all parts of applications.
Tool expectations were not met.	Expectations were that the tool would narrow down testing scope, but instead it increased the testing scope.	Manage expectations. An automated testing tool does not replace manual testing, nor does it replace the test engineer. Initially, the test effort will increase, but it will decrease on subsequent builds.
Reports produced by the tool were useless.	Reports were set up, but necessary data were not accumulated to produce meaningful reports.	Set up only reports specific to the data that will be generated. Produce reports as requested by users.

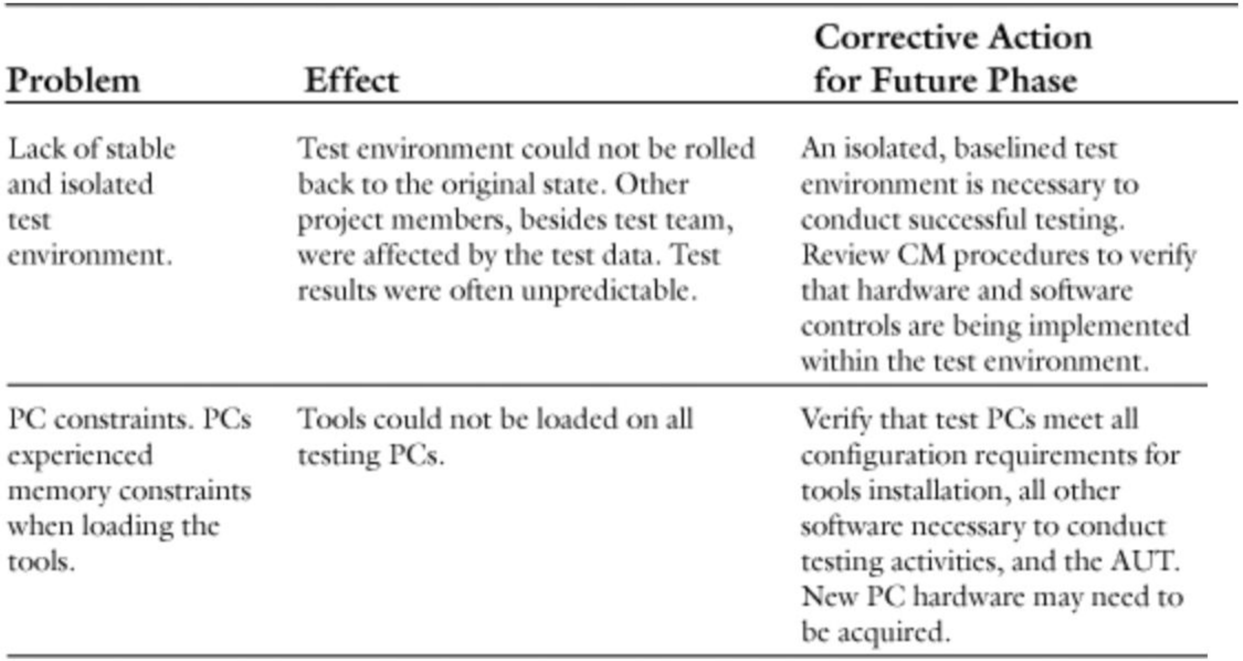
Test Program Lessons Learned



Problem	Effect	Corrective Action for Future Phase
The tool was intrusive but the development staff wasn't informed of this problem until late in the testing life cycle.	Developers were reluctant to use the tool because it required additions to the code.	Let developers know well in advance that the tool requires code additions, if applicable (not all tools are intrusive). Assure developers that the tool will not cause any problems by giving them feedback from other companies that have experience with the tool.
Tools were selected and purchased before a system engineering environment was defined.	There were many compatibility issues and work-around solutions had to be found while trying to match the system engineering environment to the requirements of the tools already purchased.	The system-engineering environment needs to be defined before tools are selected. Tools need to be selected to be compatible with the system-engineering environment, not vice versa.
Various tool versions were in use.	Procedures created in one version of the tool were not compatible in another version, causing many compatibility problems and requiring many work-around solutions.	Everyone involved in the testing life cycle needs to use the same version of the tool. Tool upgrades need to be centralized.

Test Program Lessons Learned

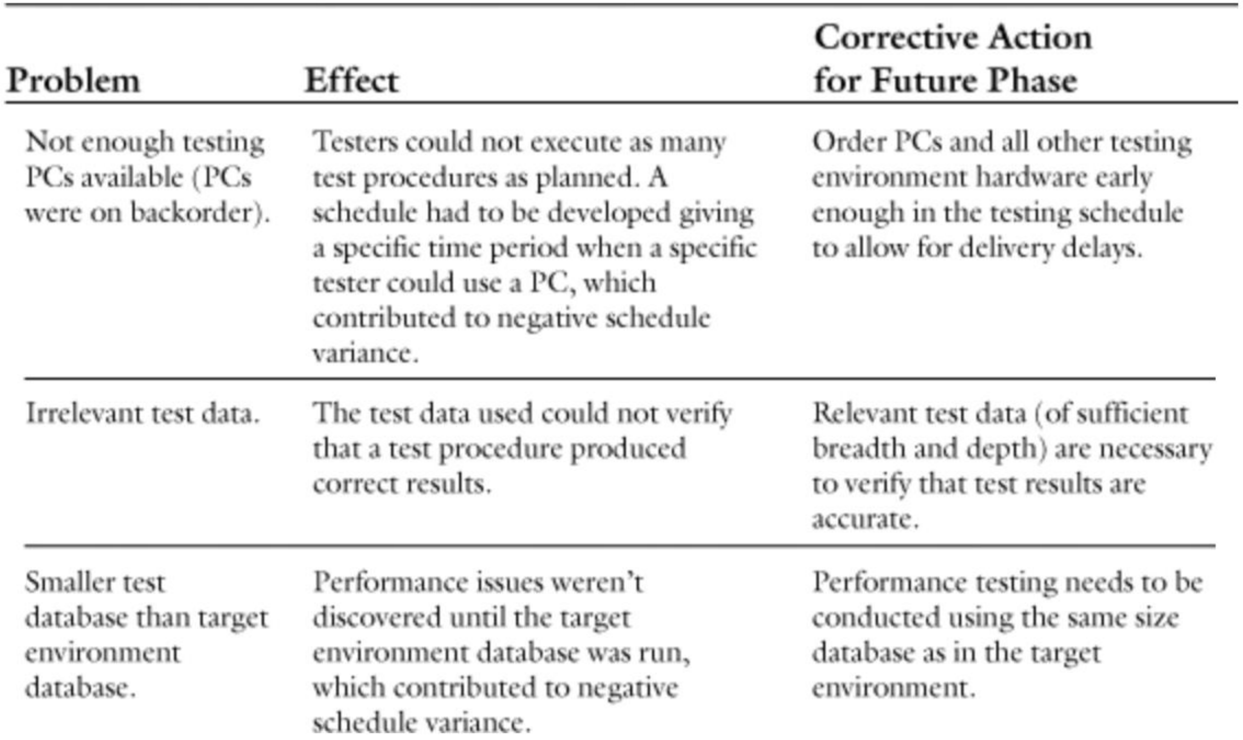
Examples of Environment-Related Problems and Corrective Actions



Problem	Effect	Corrective Action for Future Phase
Lack of stable and isolated test environment.	Test environment could not be rolled back to the original state. Other project members, besides test team, were affected by the test data. Test results were often unpredictable.	An isolated, baselined test environment is necessary to conduct successful testing. Review CM procedures to verify that hardware and software controls are being implemented within the test environment.
PC constraints. PCs experienced memory constraints when loading the tools.	Tools could not be loaded on all testing PCs.	Verify that test PCs meet all configuration requirements for tools installation, all other software necessary to conduct testing activities, and the AUT. New PC hardware may need to be acquired.

Test Program Lessons Learned

Examples of Environment-Related Problems and Corrective Actions



Problem	Effect	Corrective Action for Future Phase
Not enough testing PCs available (PCs were on backorder).	Testers could not execute as many test procedures as planned. A schedule had to be developed giving a specific time period when a specific tester could use a PC, which contributed to negative schedule variance.	Order PCs and all other testing environment hardware early enough in the testing schedule to allow for delivery delays.
Irrelevant test data.	The test data used could not verify that a test procedure produced correct results.	Relevant test data (of sufficient breadth and depth) are necessary to verify that test results are accurate.
Smaller test database than target environment database.	Performance issues weren't discovered until the target environment database was run, which contributed to negative schedule variance.	Performance testing needs to be conducted using the same size database as in the target environment.

Test Program Return on Investment

- After collecting lessons learned and other metrics, and defining the corrective actions, the test engineers also need to **assess** the **effectiveness** of the test program, including the test program's return on investment.

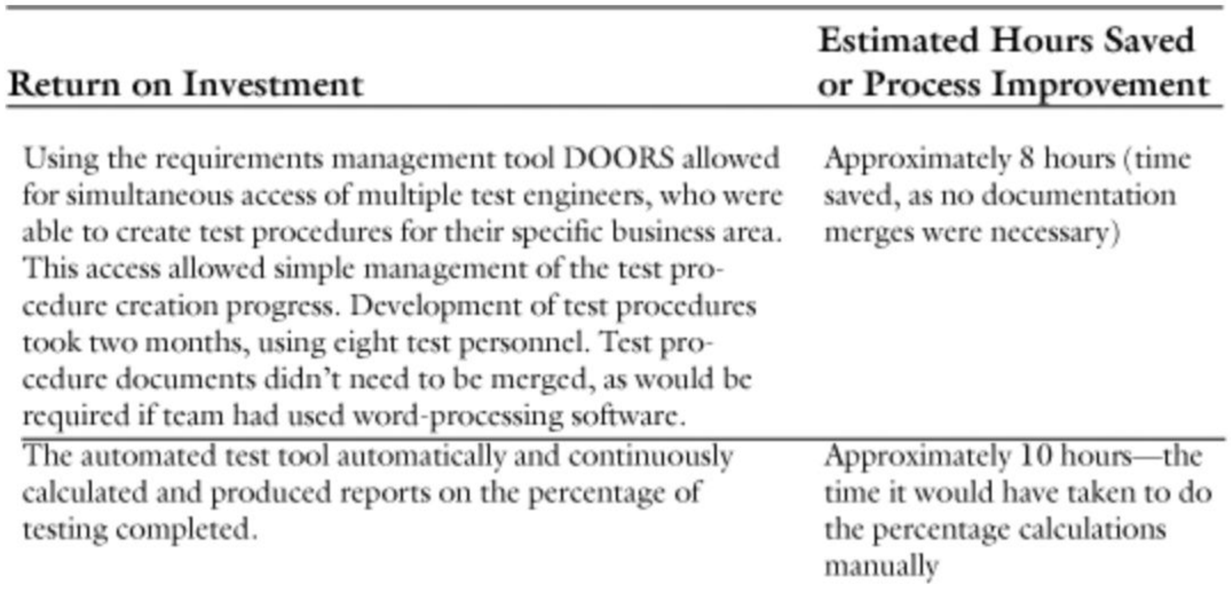
=> **Boost productivity** due to the increased speed of automating test activities in a repeatable fashion while contributing an element of maintainability.

=> **Focus on more complex, non-repeatable tests**

- Compare the **execution time of a manual test** to the **execution time of an automated test.**
- Use a **survey**.
- The return on investment exercise is aimed at **identifying** and **repeating** those activities that have proven to be effective.

Test Program Lessons Learned

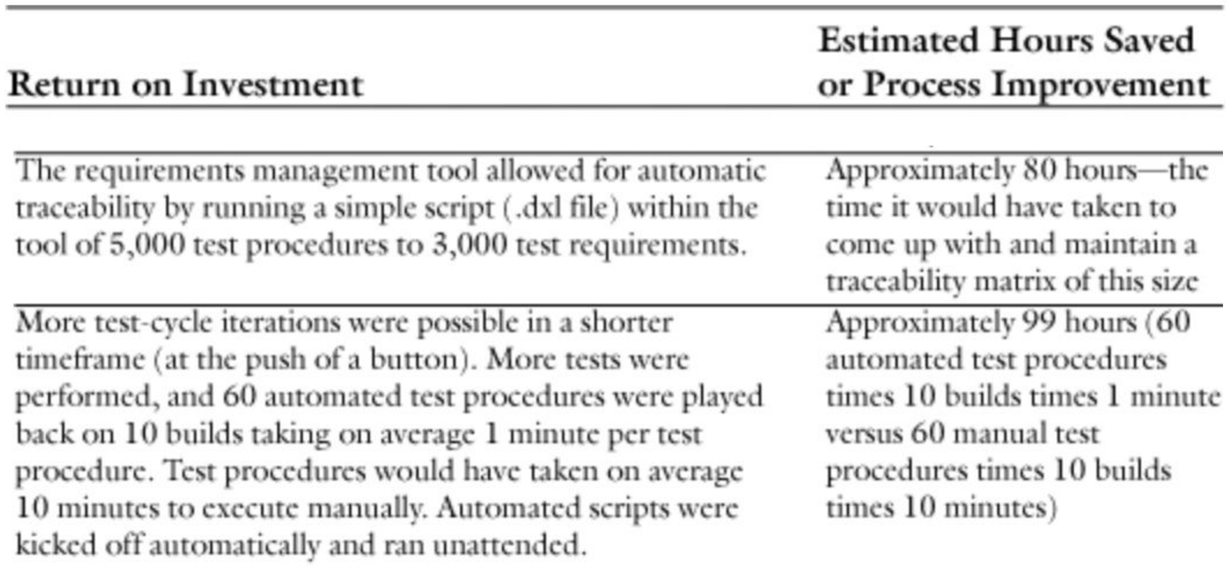
Test Program Return on Investment



Return on Investment	Estimated Hours Saved or Process Improvement
Using the requirements management tool DOORS allowed for simultaneous access of multiple test engineers, who were able to create test procedures for their specific business area. This access allowed simple management of the test procedure creation progress. Development of test procedures took two months, using eight test personnel. Test procedure documents didn't need to be merged, as would be required if team had used word-processing software.	Approximately 8 hours (time saved, as no documentation merges were necessary)
The automated test tool automatically and continuously calculated and produced reports on the percentage of testing completed.	Approximately 10 hours—the time it would have taken to do the percentage calculations manually

Test Program Lessons Learned

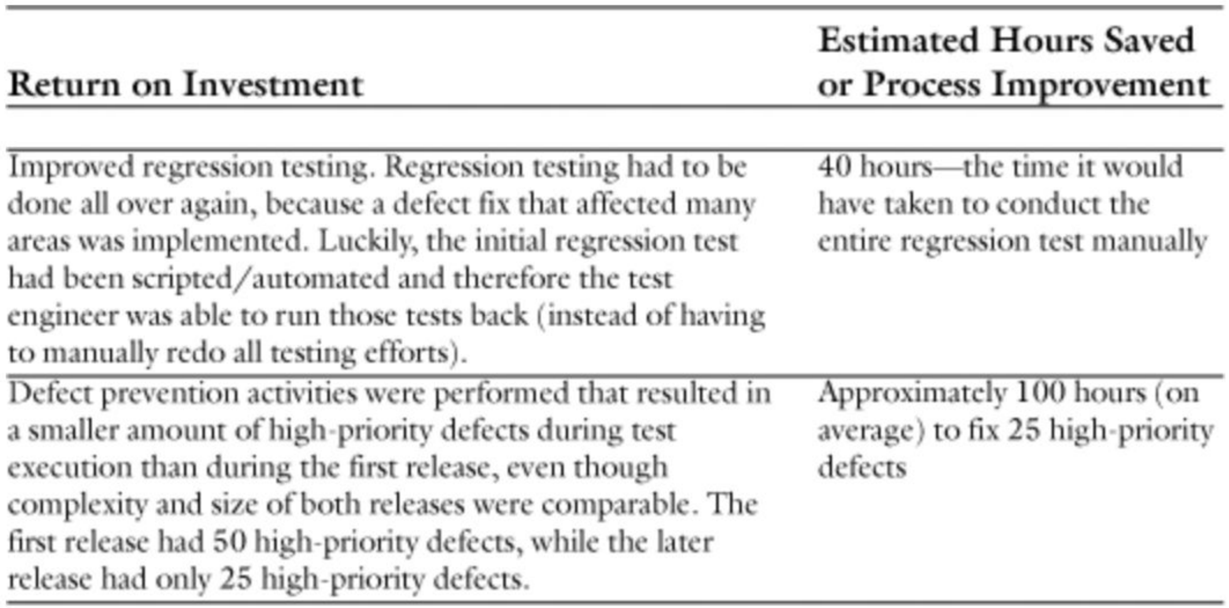
Test Program Return on Investment



Return on Investment	Estimated Hours Saved or Process Improvement
The requirements management tool allowed for automatic traceability by running a simple script (.dxl file) within the tool of 5,000 test procedures to 3,000 test requirements.	Approximately 80 hours—the time it would have taken to come up with and maintain a traceability matrix of this size
More test-cycle iterations were possible in a shorter timeframe (at the push of a button). More tests were performed, and 60 automated test procedures were played back on 10 builds taking on average 1 minute per test procedure. Test procedures would have taken on average 10 minutes to execute manually. Automated scripts were kicked off automatically and ran unattended.	Approximately 99 hours (60 automated test procedures times 10 builds times 1 minute versus 60 manual test procedures times 10 builds times 10 minutes)

Test Program Lessons Learned

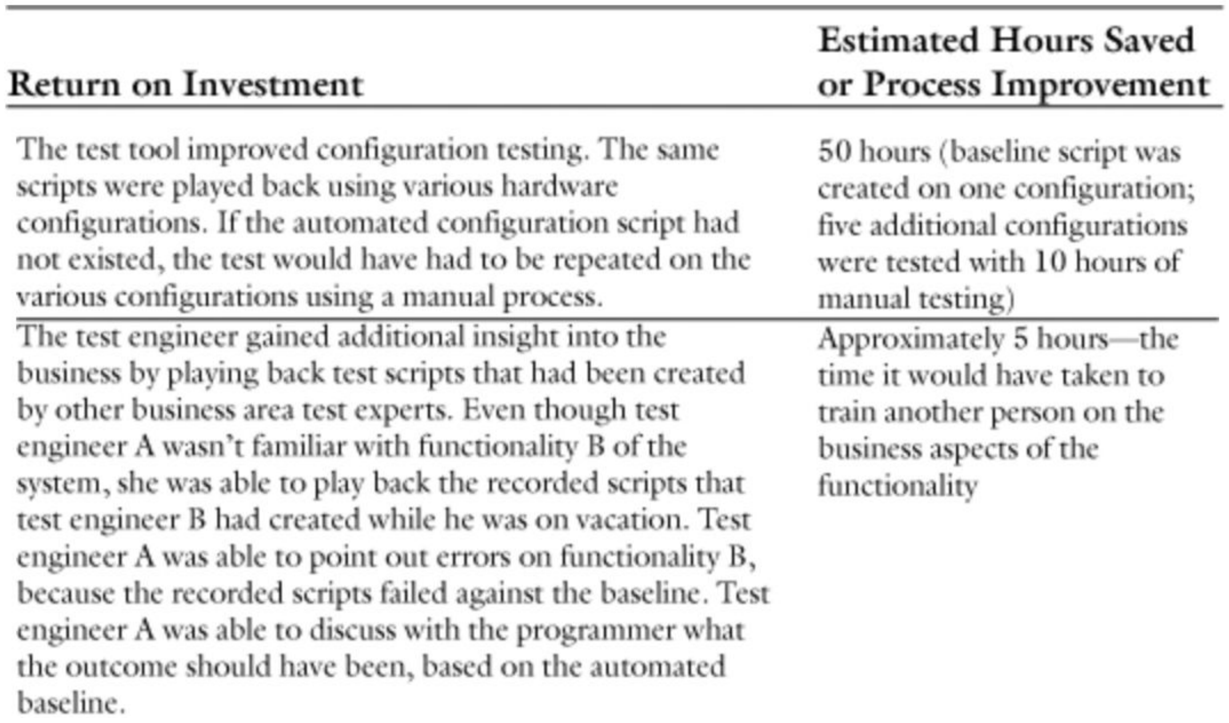
Test Program Return on Investment



Return on Investment	Estimated Hours Saved or Process Improvement
Improved regression testing. Regression testing had to be done all over again, because a defect fix that affected many areas was implemented. Luckily, the initial regression test had been scripted/automated and therefore the test engineer was able to run those tests back (instead of having to manually redo all testing efforts).	40 hours—the time it would have taken to conduct the entire regression test manually
Defect prevention activities were performed that resulted in a smaller amount of high-priority defects during test execution than during the first release, even though complexity and size of both releases were comparable. The first release had 50 high-priority defects, while the later release had only 25 high-priority defects.	Approximately 100 hours (on average) to fix 25 high-priority defects

Test Program Lessons Learned

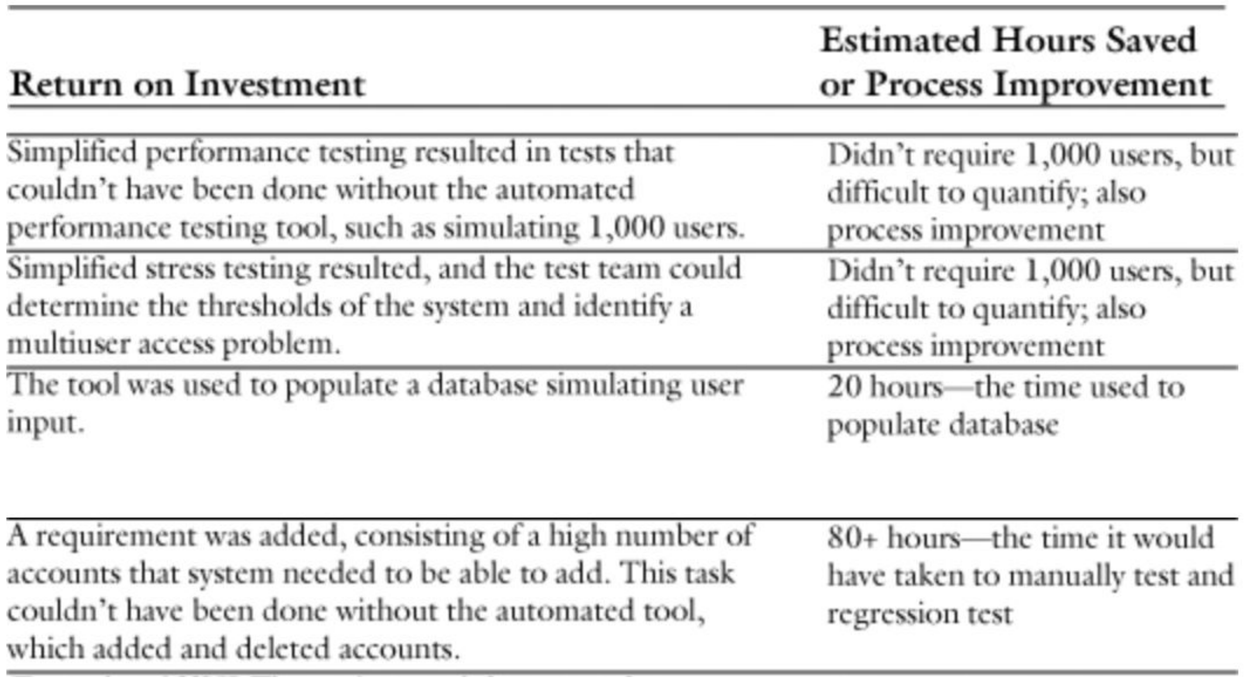
Test Program Return on Investment



Return on Investment	Estimated Hours Saved or Process Improvement
The test tool improved configuration testing. The same scripts were played back using various hardware configurations. If the automated configuration script had not existed, the test would have had to be repeated on the various configurations using a manual process.	50 hours (baseline script was created on one configuration; five additional configurations were tested with 10 hours of manual testing)
The test engineer gained additional insight into the business by playing back test scripts that had been created by other business area test experts. Even though test engineer A wasn't familiar with functionality B of the system, she was able to play back the recorded scripts that test engineer B had created while he was on vacation. Test engineer A was able to point out errors on functionality B, because the recorded scripts failed against the baseline. Test engineer A was able to discuss with the programmer what the outcome should have been, based on the automated baseline.	Approximately 5 hours—the time it would have taken to train another person on the business aspects of the functionality

Test Program Lessons Learned

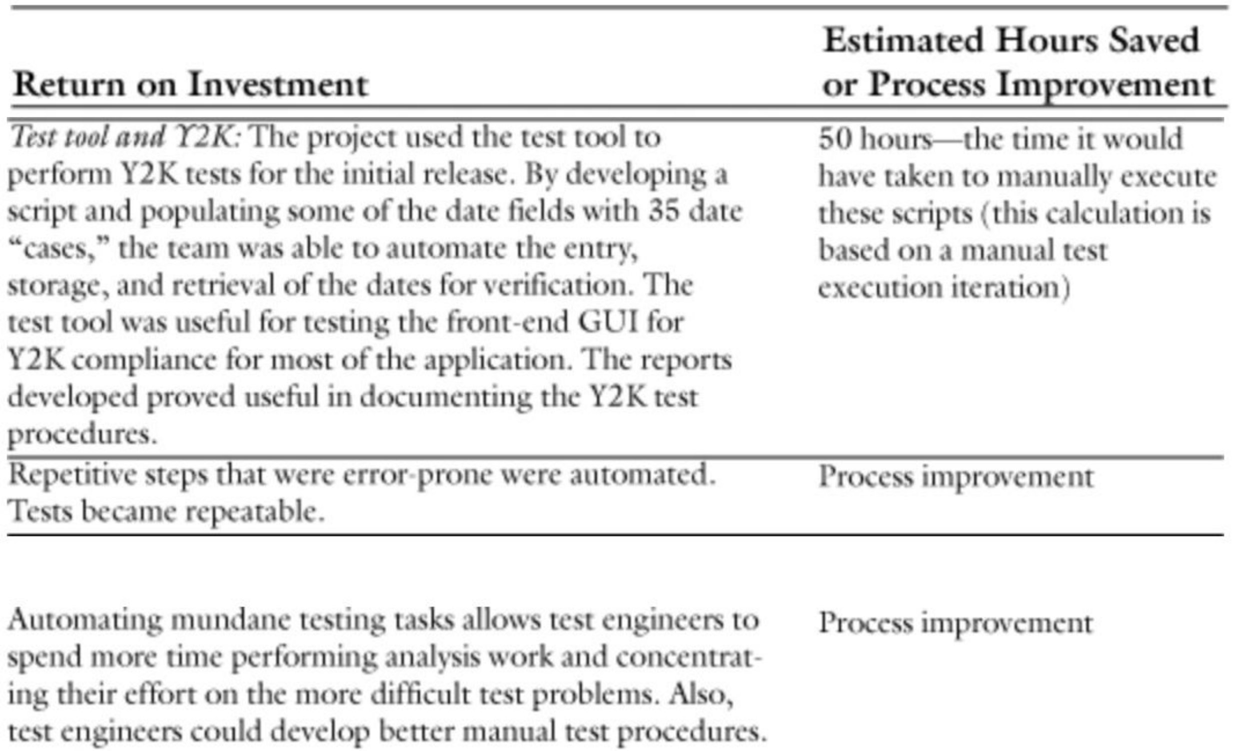
Test Program Return on Investment



Return on Investment	Estimated Hours Saved or Process Improvement
Simplified performance testing resulted in tests that couldn't have been done without the automated performance testing tool, such as simulating 1,000 users.	Didn't require 1,000 users, but difficult to quantify; also process improvement
Simplified stress testing resulted, and the test team could determine the thresholds of the system and identify a multiuser access problem.	Didn't require 1,000 users, but difficult to quantify; also process improvement
The tool was used to populate a database simulating user input.	20 hours—the time used to populate database
A requirement was added, consisting of a high number of accounts that system needed to be able to add. This task couldn't have been done without the automated tool, which added and deleted accounts.	80+ hours—the time it would have taken to manually test and regression test

Test Program Lessons Learned

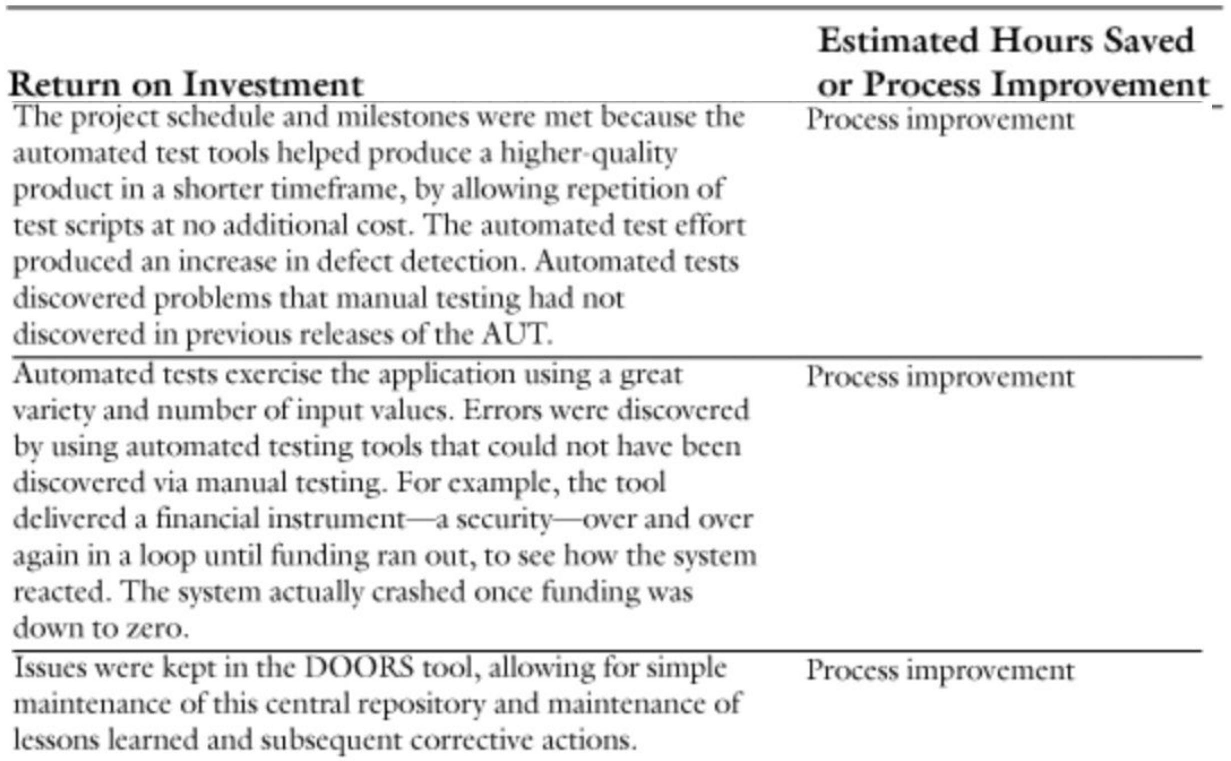
Test Program Return on Investment



Return on Investment	Estimated Hours Saved or Process Improvement
Test tool and Y2K: The project used the test tool to perform Y2K tests for the initial release. By developing a script and populating some of the date fields with 35 date "cases," the team was able to automate the entry, storage, and retrieval of the dates for verification. The test tool was useful for testing the front-end GUI for Y2K compliance for most of the application. The reports developed proved useful in documenting the Y2K test procedures.	50 hours—the time it would have taken to manually execute these scripts (this calculation is based on a manual test execution iteration)
Repetitive steps that were error-prone were automated. Tests became repeatable.	Process improvement
Automating mundane testing tasks allows test engineers to spend more time performing analysis work and concentrating their effort on the more difficult test problems. Also, test engineers could develop better manual test procedures.	Process improvement

Test Program Lessons Learned

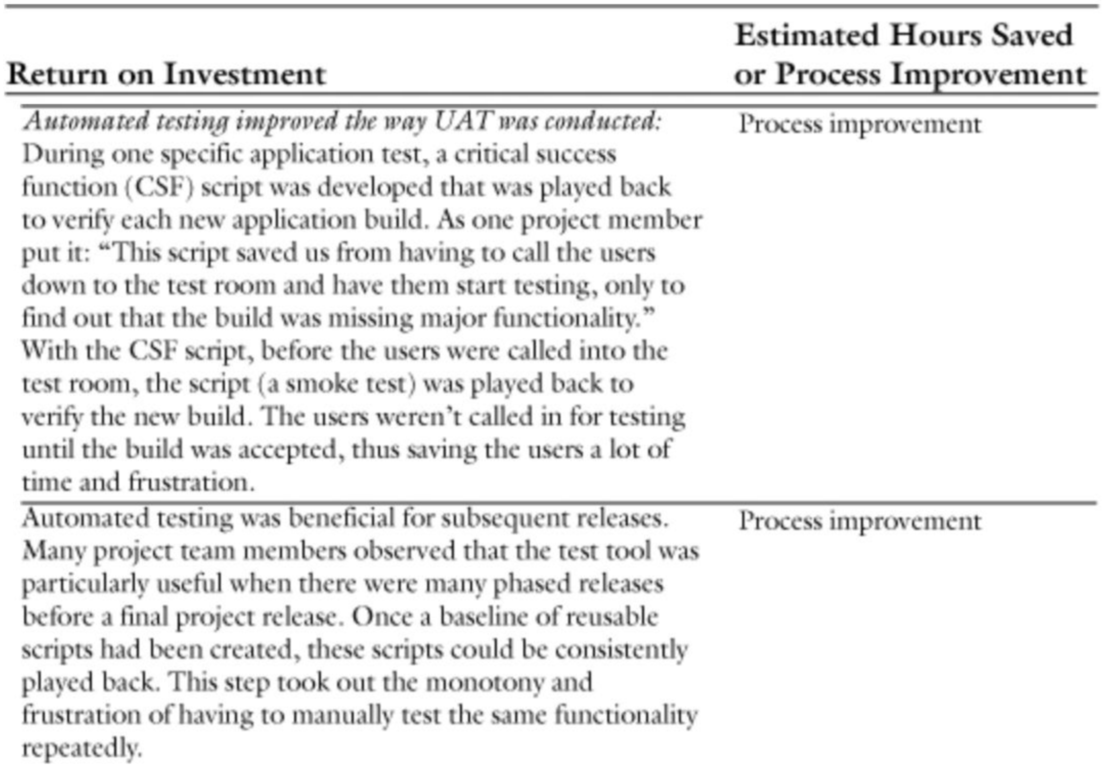
Test Program Return on Investment



Return on Investment	Estimated Hours Saved or Process Improvement
The project schedule and milestones were met because the automated test tools helped produce a higher-quality product in a shorter timeframe, by allowing repetition of test scripts at no additional cost. The automated test effort produced an increase in defect detection. Automated tests discovered problems that manual testing had not discovered in previous releases of the AUT.	Process improvement
Automated tests exercise the application using a great variety and number of input values. Errors were discovered by using automated testing tools that could not have been discovered via manual testing. For example, the tool delivered a financial instrument—a security—over and over again in a loop until funding ran out, to see how the system reacted. The system actually crashed once funding was down to zero.	Process improvement
Issues were kept in the DOORS tool, allowing for simple maintenance of this central repository and maintenance of lessons learned and subsequent corrective actions.	Process improvement

Test Program Lessons Learned

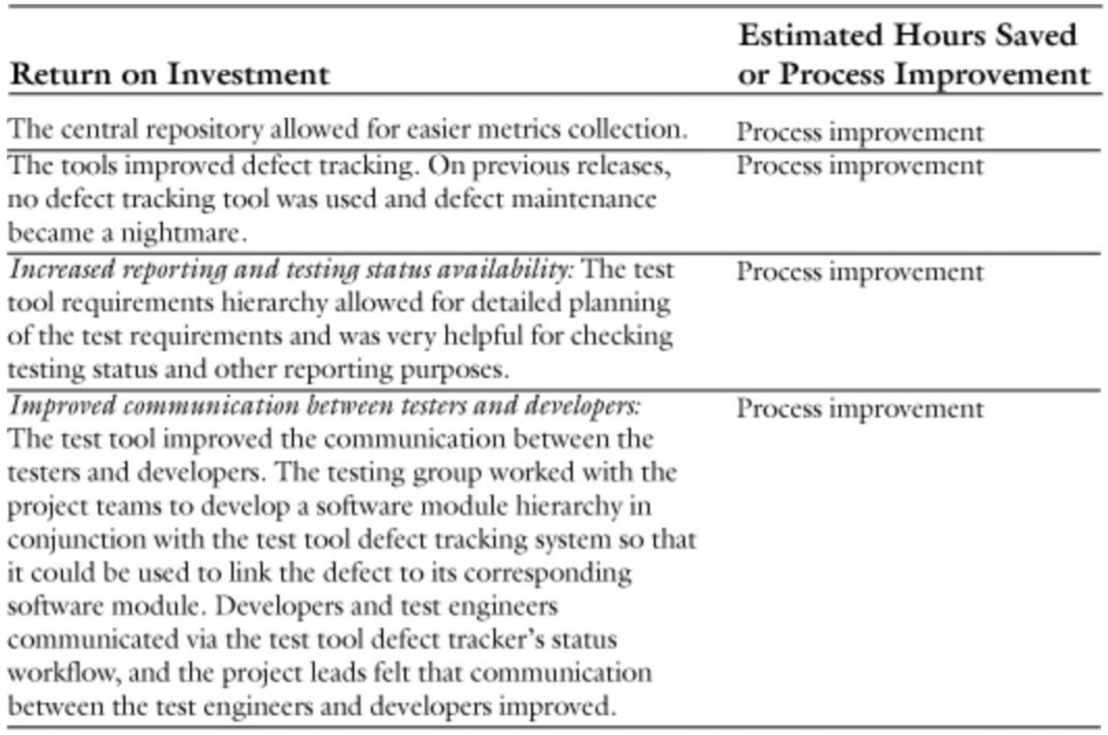
Test Program Return on Investment



Return on Investment	Estimated Hours Saved or Process Improvement
Automated testing improved the way UAT was conducted: During one specific application test, a critical success function (CSF) script was developed that was played back to verify each new application build. As one project member put it: “This script saved us from having to call the users down to the test room and have them start testing, only to find out that the build was missing major functionality.” With the CSF script, before the users were called into the test room, the script (a smoke test) was played back to verify the new build. The users weren’t called in for testing until the build was accepted, thus saving the users a lot of time and frustration.	Process improvement
Automated testing was beneficial for subsequent releases. Many project team members observed that the test tool was particularly useful when there were many phased releases before a final project release. Once a baseline of reusable scripts had been created, these scripts could be consistently played back. This step took out the monotony and frustration of having to manually test the same functionality repeatedly.	Process improvement

Test Program Lessons Learned

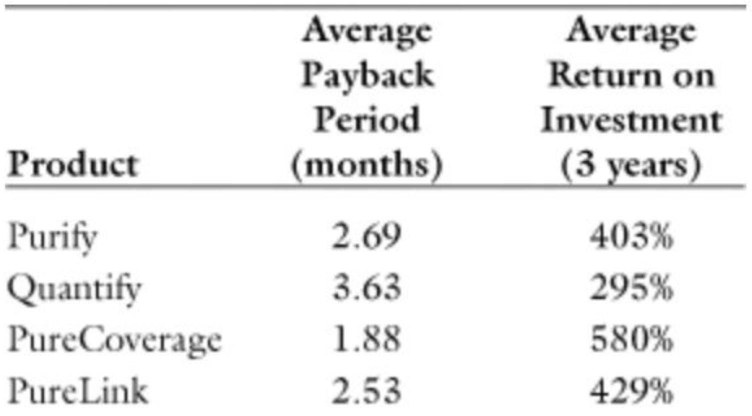
Test Program Return on Investment



Return on Investment	Estimated Hours Saved or Process Improvement
The central repository allowed for easier metrics collection.	Process improvement
The tools improved defect tracking. On previous releases, no defect tracking tool was used and defect maintenance became a nightmare.	Process improvement
Increased reporting and testing status availability: The test tool requirements hierarchy allowed for detailed planning of the test requirements and was very helpful for checking testing status and other reporting purposes.	Process improvement
Improved communication between testers and developers: The test tool improved the communication between the testers and developers. The testing group worked with the project teams to develop a software module hierarchy in conjunction with the test tool defect tracking system so that it could be used to link the defect to its corresponding software module. Developers and test engineers communicated via the test tool defect tracker's status workflow, and the project leads felt that communication between the test engineers and developers improved.	Process improvement

Test Program Lessons Learned

Test Tool Return on Investment



Product	Average Payback Period (months)	Average Return on Investment (3 years)
Purify	2.69	403%
Quantify	3.63	295%
PureCoverage	1.88	580%
PureLink	2.53	429%

Test Program Lessons Learned

Rational’s Quantify Tool’s User Results:

- The average programmer spends 4% to 8% of his or her time optimizing or improving performance. The calculation uses 5%.
- Quantify 5× productivity factor (range 5×−25×)

Time spent addressing performance issues without Quantify (weeks)

0.05×48 weeks per year = 2.4 weeks/year

Time spent addressing performance issues with Quantify (weeks)

2.4 weeks/year × 1/5 (Quantify productivity factor) = 0.48 week/year

Quantify saves 2.4 - 0.48 = 1.92 hours/month per developer

Test Program Lessons Learned

Additional Considerations:

- Organizations that list performance as a critical product differentiation feature spent roughly 9.4 person-weeks per year addressing performance issues in addition to the 5% spent by the average developer. The additional savings calculated for dedicated performance individuals were 7.84 weeks. These additional savings are not included in the ROI calculations.
- Quantify reduced setup 15× to 20× with respect to current tools.
- Several users reported that Quantify allowed them to collect data from their complex applications that were previously unattainable.

Test Program Lessons Learned

- Test teams can conduct their own surveys to inquire about the potential value of testing process and tool changes.
- Include business analysts, requirements management specialists, developers, test engineers, and selected end users.

=> Gather benefits information on effectiveness of the test life cycle and the test tools and the improvement activities implemented.

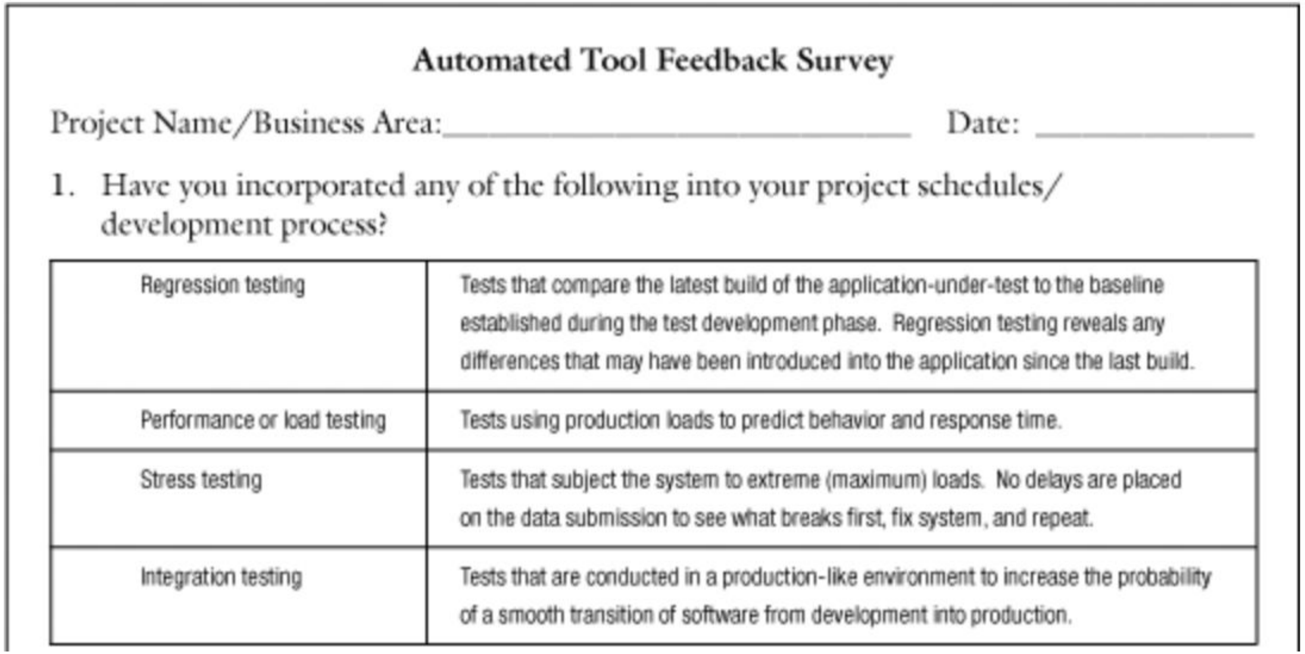
Test Program Lessons Learned

Short questions

1. When comparing the manual testing effort with the automated testing effort, which criteria need to be considered if the comparison is to be valuable?
2. What other approaches could be implemented to measure the test program return on investment?

Test Program Lessons Learned

Feedback Survey



Automated Tool Feedback Survey

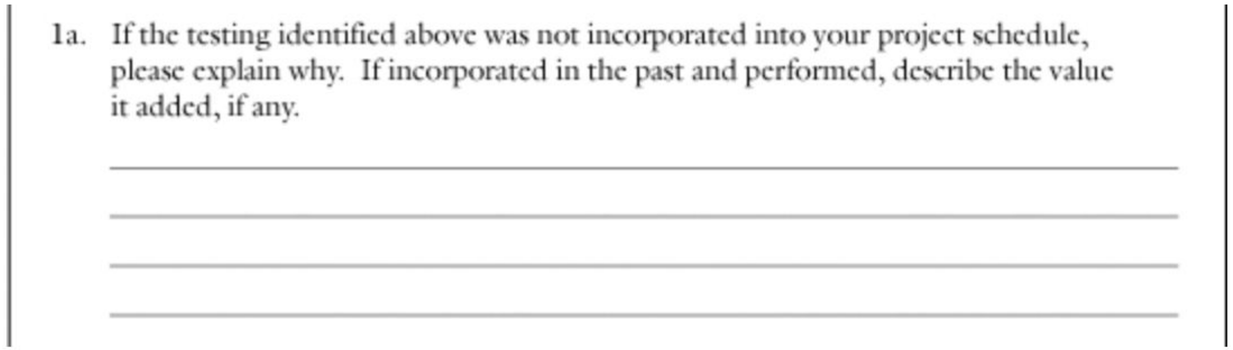
Project Name/Business Area:____________________________ Date: ______________

1. Have you incorporated any of the following into your project schedules/ development process?

Regression testing	Tests that compare the latest build of the application-under-test to the baseline established during the test development phase. Regression testing reveals any differences that may have been introduced into the application since the last build.
Performance or load testing	Tests using production loads to predict behavior and response time.
Stress testing	Tests that subject the system to extreme (maximum) loads. No delays are placed on the data submission to see what breaks first, fix system, and repeat.
Integration testing	Tests that are conducted in a production-like environment to increase the probability of a smooth transition of software from development into production.

Test Program Lessons Learned

Feedback Survey



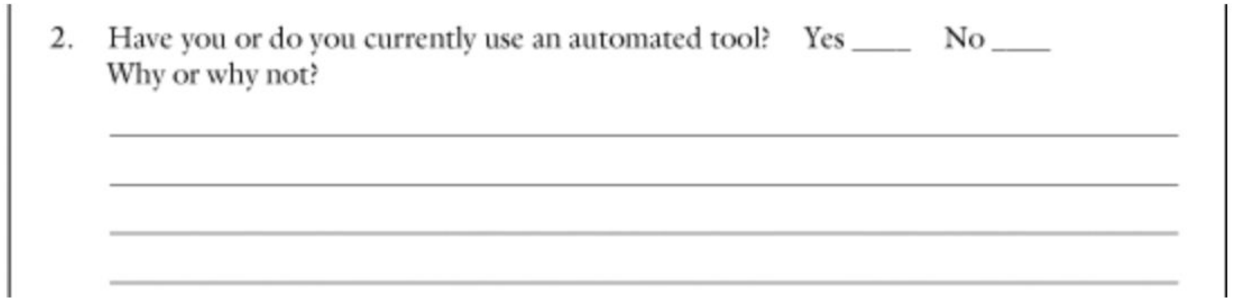
1a. If the testing identified above was not incorporated into your project schedule, please explain why. If incorporated in the past and performed, describe the value it added, if any.

____

____

____

____



2. Have you or do you currently use an automated tool? Yes ____ No ____
Why or why not?

____

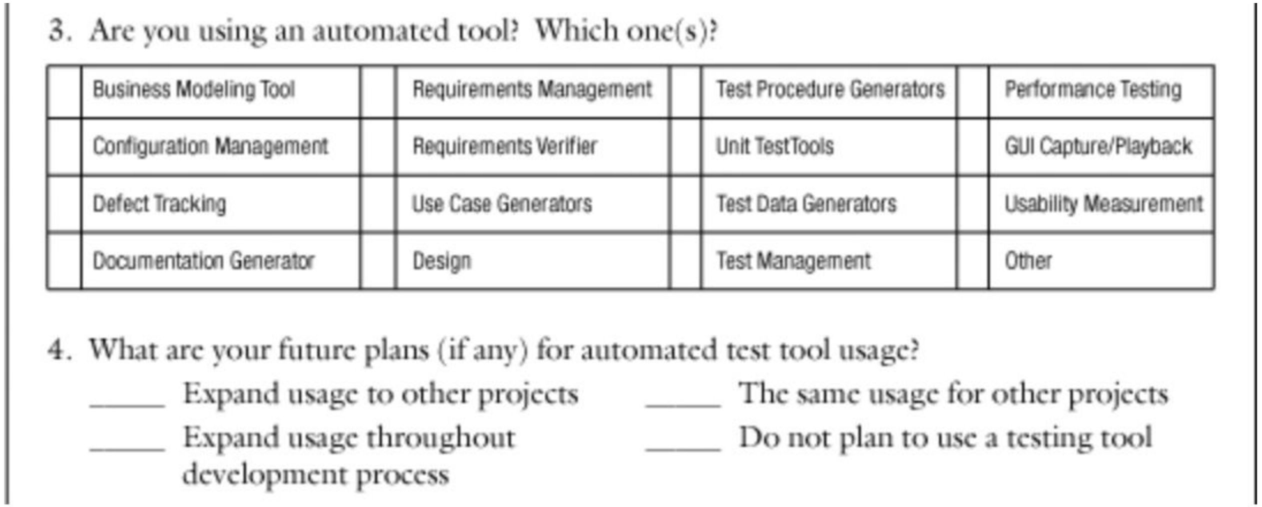
____

____

____

Test Program Lessons Learned

Feedback Survey



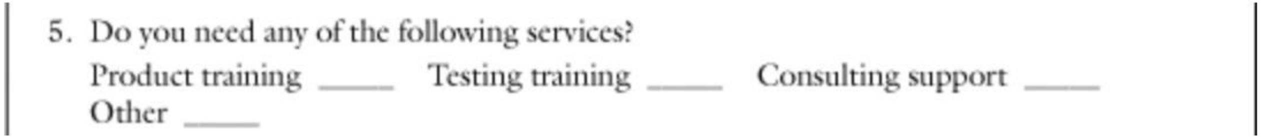
3. Are you using an automated tool? Which one(s)?

	Business Modeling Tool		Requirements Management		Test Procedure Generators		Performance Testing
	Configuration Management		Requirements Verifier		Unit TestTools		GUI Capture/Playback
	Defect Tracking		Use Case Generators		Test Data Generators		Usability Measurement
	Documentation Generator		Design		Test Management		Other

4. What are your future plans (if any) for automated test tool usage?

_____ Expand usage to other projects _____ The same usage for other projects

_____ Expand usage throughout development process _____ Do not plan to use a testing tool



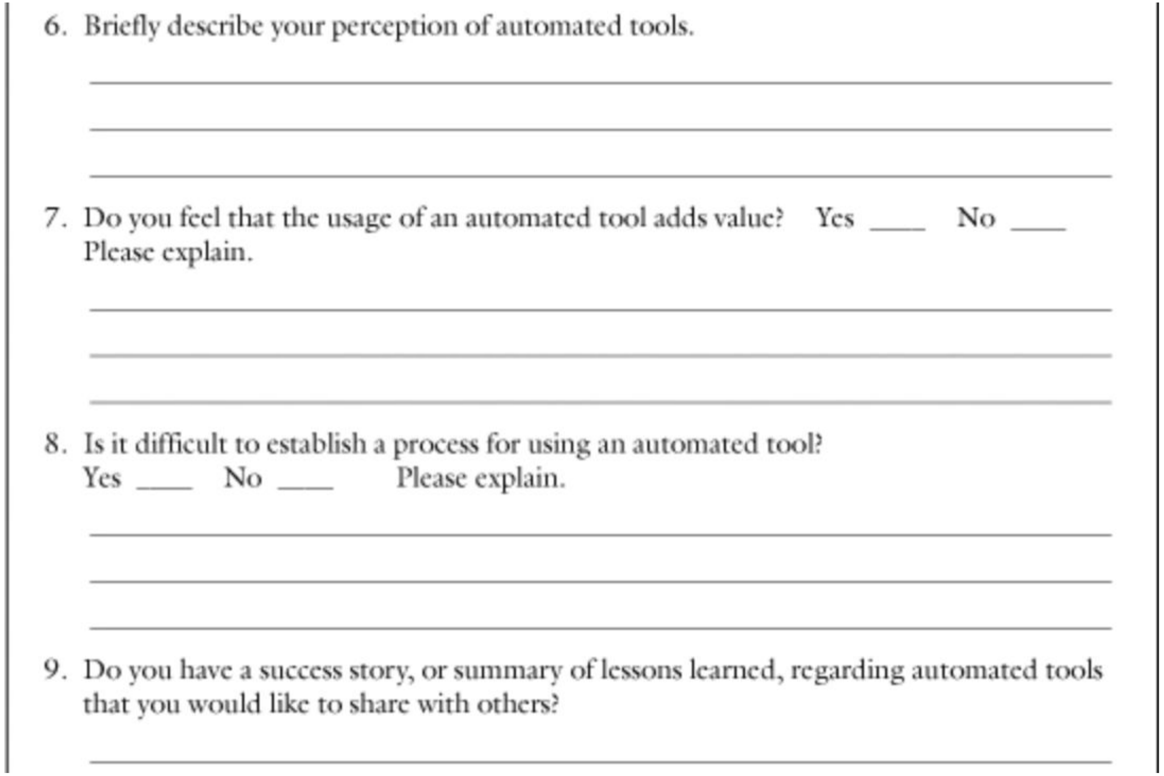
5. Do you need any of the following services?

Product training _____ Testing training _____ Consulting support _____

Other _____

Test Program Lessons Learned

Feedback Survey



6. Briefly describe your perception of automated tools.

____

____

____

7. Do you feel that the usage of an automated tool adds value? Yes ____ No ____
Please explain.

____

____

____

8. Is it difficult to establish a process for using an automated tool?
Yes ____ No ____ Please explain.

____

____

____

9. Do you have a success story, or summary of lessons learned, regarding automated tools that you would like to share with others?

____

Test Program Lessons Learned

"When I was at Apple, my team conducted a review of 17 test plans in use by our department. Our findings were that none of the plans [was] actually in use, at all. Some of them were almost pure boilerplate. In one case the tester wasn't aware that his product had a test plan until we literally opened a drawer in his desk and found it lying where the previous test engineer on that product had left it!"

=> Test automation is software development.

=> The test team also needs to be conscious of the benefits offered by a strategy that accepts *good enough* test automation => follow design vs guidelines.

Chapter summary

- Following test execution, the test team needs to review the performance of the test program to determine where improvements can be implemented to benefit the next project. This test program review represents the final phase of the ATLM.
- Throughout the test program, the test team collects various test metrics. The focus of the test program review includes an assessment of whether the application satisfies acceptance criteria and is ready to go into production. This review also includes an evaluation of earned value progress measurements and other metrics collected.

Chapter summary

- The test team needs to adopt, as part of its culture, an ongoing, iterative process composed of lessons learned activities. Such a program encourages test engineers to suggest corrective action immediately, when such actions might potentially have a significant effect on test program performance.
- Throughout the entire test life cycle, a good practice is to document and begin to evaluate lessons learned at each milestone. The metrics that are collected throughout the test life cycle, and especially during the test execution phase, help pinpoint problems that should be addressed.

Chapter summary

- Lessons learned, metrics evaluations, and corresponding improvement activity or corrective action need to be documented throughout the entire test process in an easily accessible central repository.
- After collecting lessons learned and other metrics, and defining the corrective actions, test engineers need to assess the effectiveness of the testprogram by evaluating the test program’s return on investment. Test engineers capture measures of the benefits of automation realized throughout the test life cycle so as to perform this assessment.

Chapter summary

- Test teams can perform their own surveys to inquire about the potential value of process and tool changes. A survey form can be employed to solicit feedback on the potential use of requirements management tools, design tools, and development tools. Surveys also help to identify potential misconceptions and gather positive feedback.

References

[1] Used with permission of Sam Guckenheimer, Rational Software. www.rational.com.

[2] Bach, J. *Process Evolution in a Mad World.* Bellevue, WA: Software Testing Laboratories, 1997.