**Introducing the ATLM**

Software project managers and software developers building today's applications face the challenge of doing so within an ever-shrinking schedule and with minimal resources. As part of their attempt to do more with less, organizations want to test software adequately, but as quickly and thoroughly as possible. To accomplish this goal, organizations are turning to automated testing.

Faced with this reality and realizing that many tests cannot be executed manually, such as simulating 1,000 virtual users for volume testing, software professionals are introducing automated testing to their projects. But these software professionals may not know what's involved in introducing an automated test tool to a software project, and they may be unfamiliar with the breadth of application that automated test tools have today.

By using the systematic approach outlined within the ATLM, organizations can organize and execute test activities in such a way as to maximize test coverage within the limits of testing resources. This structured test methodology involves a multi-stage process, supporting the detailed and interrelated activities that are required to introduce and utilize an automated test tool:

* Develop test design.
* Develop and execute test cases.
* Develop and manage test data and the test environment.
* Document, track, and obtain closure on issue/trouble reports.

Clearly, the emphasis on automated testing represents a paradigm change for the software industry. This change doesn't simply involve the application of tools and the performance of test automation. Rather, it blankets the entire test lifecycle and the system development lifecycle. The ATLM implementation takes place in parallel with the system development lifecycle. For software professionals to make a successful leap to automated testing, they must embrace structured approaches to testing. The ATLM is revolutionary in the fact that it promulgates a new structured, building-block approach to the entire test lifecycle, which enables software test professionals to approach software testing in a methodical and repeatable fashion.

The growth of automated test capability has stemmed in large part from the growing popularity of the iterative and incremental development lifecycle, a software development methodology that focuses on minimizing the development schedule while providing frequent, incremental software builds. The objective of this incremental and iterative development is to engage the user and the test team early throughout the design and development of each build in order to refine the software, thereby ensuring that it more closely reflects the needs and preferences of the user and thus addressing the riskiest aspects of development in early builds.

In this environment of continual changes and additions to the software through each software build, software testing itself takes on an iterative nature. Each new build is accompanied by a considerable number of new tests as well as rework to existing test scripts, just as there is rework on previously released software modules. Given the continual changes and additions to software applications, especially Web applications, automated software testing becomes an important control mechanism to ensure accuracy and stability of the software through each build.

The ATLM, invoked to support testing efforts involving automated test tools, incorporates a multi-stage process. The methodology supports the detailed and interrelated activities that are required to decide whether to acquire an automated testing tool. The methodology includes the process of how to introduce and utilize an automated test tool, covers test development and test design, and addresses test execution and management. The methodology also supports the development and management of test data and the test environment, and addresses test documentation to include problem reports.

The ATLM methodology represents a structured approach, which depicts a process with which to approach and execute testing. This structured approach is necessary to help steer the test team away from these common test program mistakes:

* Implementing the use of an automated test tool without a testing process in place, resulting in an ad hoc, non-repeatable, non-measurable test program.
* Implementing a test design without following any design standards, resulting in the creation of test scripts that are not repeatable and therefore not reusable for incremental software builds.
* Attempting to automate 100% of test requirements, when tools or in-house–developed automated test harnesses do not support automation of all tests required.
* Using the wrong tool or developing a too-elaborate in-house test harness.
* Initiating test tool implementation too late in the application-development lifecycle—not allowing sufficient time for tool setup and test tool introduction process (learning curve).
* Initiating test engineer involvement too late in the application-development lifecycle, resulting in poor understanding of the application and system design, which results in incomplete testing.

The Automated Test Lifecycle Methodology (ATLM) comprises six primary processes or components:

1. Decision to Automate Testing.
2. Test Tool Acquisition.
3. Automated Testing Introduction Process.
4. Test Planning, Design, and Development.
5. Execution and Management of Tests.
6. Test Program Review and Assessment.

The following sections describe each primary process, as well as the subordinate processes contained within each primary process

## Phase 1: Decision to Automate Testing

The decision to automate testing represents the first phase of the ATLM. This phase covers the entire process that goes into the automated testing decision. During this phase, it's important for the test team to manage automated testing expectations and to outline the potential benefits of automated testing when implemented correctly. A test tool proposal needs to be outlined, which will be helpful in acquiring management support.

### Overcoming False Expectations for Automated Testing

While it has been proven that automated testing is valuable and can produce a successful return on investment, there isn't always an immediate payback on investment. It's important to address some of the misconceptions that persist in the software industry and to manage the automated testing utopia. Following is a list of just a few of the misconceptions that need to be addressed. People often see test automation as a silver bullet; when they find that test automation requires a significant short-term investment of time and energy to achieve a long-term return on investment (ROI) of faster and cheaper regression testing (for example), the testing tool often becomes "shelf-ware." This is why it's important to manage expectations in order to introduce automated testing correctly into a project.

#### Automatic Test Plan Generation

**Currently, there is no commercially available tool that can automatically create a comprehensive test plan while also supporting test design and execution.**

Throughout a software test career, the test engineer can expect to witness test tool demonstrations and review an abundant amount of test tool literature. Often the test engineer will be asked to stand before one or more senior managers to give a test tool functionality overview. As always, the presenter must bear in mind the audience. In this case, the audience may represent individuals with just enough technical knowledge to make them enthusiastic about automated testing, while unaware of the complexity involved with an automated test effort. Specifically, the managers may have obtained secondhand information about automated test tools, and may have reached the wrong interpretation of the actual capability of automated test tools.

What the audience at the management presentation may be waiting to hear is that the tool you're proposing automatically develops the test plan, designs and creates the test procedures, executes all the test procedures, and analyzes the results automatically. Meanwhile, you start out the presentation by informing the group that automated test tools should be viewed as *enhancements to manual testing*, and that automated test tools will not automatically develop the test plan, design and create the test procedures, or execute the test procedures.

Shortly into the presentation and after several management questions, it becomes very apparent just how much of a divide exists between the reality of the test tool capabilities and the perceptions of the individuals in the audience. The term *automated test tool* seems to bring with it a great deal of wishful thinking that's not closely aligned with reality. An automated test tool will not replace the human factor necessary for testing a product. The proficiencies of test engineers and other quality assurance experts will still be needed to keep the test machinery running. A test tool can be viewed as an additional part of the machinery that supports the release of a good product.

#### One Test Tool Fits All

**Currently, no single test tool exists that can be used to support all operating system environments.**

Generally, a single test tool will not fulfill all the testing requirements for an organization. Consider the experience of one test engineer encountering such a situation. The test engineer was asked by a manager to find a test tool that could be used to automate the testing of all the department's applications. The department was using various technologies including mainframe computers and Sun workstations; operating systems such as Windows 3.1, Windows 95, Windows NT, and Windows 2000; programming languages such as Visual C++ and Visual Basic; other client/server technologies; and Web technologies such as DHTML, XML, ASP, and so on.

After conducting a tool evaluation, the test engineer determined that the tool of choice was not compatible with the Visual C++ third-party add-ons (in this case, Stingray grids). Another tool had to be brought in that was compatible with this specific application.

#### Immediate Reduction in Schedule

**An automated test tool will not immediately minimize the testing schedule.**

Another automated test misconception is the expectation that the use of an automated testing tool on a new project will immediately minimize the test schedule. The testing schedule will not experience the anticipated decrease at first, and an allowance for schedule increase is required when initially introducing an automated test tool. This is due to the fact that when rolling out an automated test tool, the current testing process has to be augmented or an entirely new testing process has to be developed and implemented. The entire test team—and possibly the development team—needs to become familiar with this new automated testing process (such as ATLM) and needs to follow it. Once an automatic testing process has been established and effectively implemented, the project can expect to experience gains in productivity and turnaround time that have a positive effect on schedule and cost.

### Benefits of Automated Testing

The previous discussion points out and clarifies some of the false automated testing expectations that exist. The test engineer will also need to be able to elaborate on the true benefits of automated testing, when automated testing is implemented correctly and a process is followed. The test engineer must evaluate whether potential benefits fit required improvement criteria and whether the pursuit of automated testing on the project is still a logical fit, given the organizational needs. There are three significant automated test benefits (in combination with manual testing):

* Producing a reliable system.
* Improving the quality of the test effort.
* Reducing test effort and minimizing schedule.

Many return on investment case studies have been done with regard to the implementation of automated testing. One example is a research effort conducted by [imbus GmbH](http://www.imbus.de/). They conducted a test automation value study in order to collect test automation measurements with the purpose of studying the benefits of test automation versus the implementation of manual test methods. Their research determined that the breakeven point of automated testing is on average at 2.03 test runs. (T. Linz and M. Daigl, "GUI Testing Made Painless: Implementation and Results of the ESSI Project Number 24306," 1998.)

### Acquiring Management Support

Whenever an organization tries to adopt a new technology, they encounter a significant effort when determining how to apply it to their needs. Even with completed training, organizations wrestle with time-consuming false starts before they become capable with the new technology. For the test team interested in implementing automated test tools, the challenge is how to best present the case for a new test automation technology and its implementation to the management team.

Test engineers need to influence management's expectations for the use of automated testing on projects. Test engineers can help to manage expectations of others in the organization by forwarding helpful information to the management staff. Bringing up test tool issues during strategy and planning meetings can also help develop better understanding of test tool capabilities for everyone involved on a project or within the organization. A test engineer can develop training material on the subject of automated testing and can advocate to management that a seminar be scheduled to conduct the training.

The first step in moving toward a decision to automate testing on a project requires that the test team adjust management understanding of the appropriate application of automated testing for the specific need at hand. For example, the test team needs to check early on whether management is cost-averse and would be unwilling to accept the estimated cost of automated test tools for a particular effort. If so, test personnel need to convince management about the potential return on investment by conducting cost/benefit analysis.

If management is willing to invest in an automated test tool, but is unable or unwilling to staff a test team with individuals having the proper software skill level or to provide for adequate test tool training, the test team needs to point out the risks involved and/or may need to reconsider a recommendation to automate test.

Management also needs to be made aware of the additional cost involved when introducing a new tool—not only for the tool purchase, but for initial schedule/cost increase, additional training costs, and for enhancing an existing testing process or implementing a new testing process.

Test automation represents highly flexible technology, which provides several ways to accomplish an objective. Use of this technology requires new ways of thinking, which only amplifies the problem of test tool implementation. Many organizations can readily come up with examples of their own experience of technology that failed to deliver on its potential because of the difficulty of overcoming the "Now what?" syndrome. The issues that organizations face when adopting automated test systems include those outlined below:

* Finding/hiring test tool experts.
* Using the correct tool for the task at hand.
* Developing and implementing an automated testing process, which includes developing automated test design and development standards.
* Analyzing various applications to determine those that are best suited for automation.
* Analyzing the test requirements to determine the ones suitable for automation.
* Training the test team on the automated testing process, automated test design, development, and execution.
* Initial increase in schedule and cost.

## Phase 2: Test Tool Acquisition

Test tool acquisition represents the second phase of the ATLM. This phase guides the test engineer through the entire test tool evaluation and selection process, starting with confirmation of management support. Since a tool should support most of the organizations' testing requirements, whenever feasible the test engineer will need to review the system's engineering environment and other organizational needs and come up with a list of tool evaluation criteria. A review of the different types of tools available to support aspects of the entire testing lifecycle is provided in [Automated Software Testing: Introduction, Management, and Performance](http://www.informit.com/content/index.asp?product_id=%7bCBB19A2E-D0BE-40B9-9250-15E4D89A61EF%7d&t=%7b94AE5B48-1D7D-462A-A4A6-83CE19EC0705%7d&n=%7b1CBD305F-D503-49FA-9699-8B2C06C9C520%7d) (as part of the ATLM), enabling the reader to make an informed decision with regard to the types of tests to be performed on a particular project. The test engineer then needs to define an evaluation domain to pilot the test tool. Finally, after all those steps have been completed, the test engineer can make vendor contact to bring in the selected tool(s). Test personnel then evaluate the tool, based on sample criteria provided.

## Phase 3: Automated Testing Introduction Process

The process of introducing automated testing to a new project team constitutes the third phase of the ATLM. This phase outlines the steps necessary to successfully introduce automated testing to a new project, which are summarized in the following sections.

### Test Process Analysis

Test process analysis ensures that an overall test process and strategy are in place and are modified, if necessary, to allow automated testing to be introduced in a successful fashion. The test engineers define and collect test process metrics in order to allow for process improvement. Here test goals/objectives and strategies need to be defined and test process needs to be documented and communicated to the test team. In this phase, the kinds of testing applicable for the technical environment will be defined, and tests are defined that can be supported by automated tools.

During the test process analysis, techniques are defined. Best practices are laid out, such as conducting performance testing during the unit-testing phase.

Plans for user involvement are assessed, and test team personnel skills are analyzed against test requirements and planned test activities. Early test team participation is emphasized, supporting refinement of requirement specifications into terms that can be adequately tested while also supporting test team understanding of application requirements and design.

### Test Tool Consideration

The test tool consideration process includes steps that investigate whether incorporation of automated test tools that have been brought into the company without a specific project in mind now would be beneficial to a specific project, given the project testing requirements, available test environment, personnel resources, user environment, platform, and product features of the application under test. Schedule is reviewed to ensure sufficient time for test tool setup and development of requirements hierarchy; potential test tools and utilities are mapped to test requirements, test tool compatibility with the application and environment is verified, and workaround solutions are investigated for incompatibility problems that surfaced during compatibility tests.

## Phase 4: Test Planning, Design, and Development

Test planning, design, and development is the fourth phase of the ATLM. These subjects are summarized in the following sections.

### Test Planning

The test planning stage represents the need to review long–lead-time test planning activities. During this phase, the test team identifies test procedure creation standards and guidelines; hardware, software, and network required to support test environment; test data requirements; a preliminary test schedule; performance measure requirements; a procedure to control test configuration and environment; as well as defect-tracking procedure(s) and associated tracking tool(s).

The test plan contains the results of each preliminary phase of the structured test methodology (ATLM). The test plan will define roles and responsibilities, project test schedule, test planning and design activities, test environment preparation, test risks and contingencies, and acceptable level of thoroughness (test acceptance criteria). Test plan appendices may include test procedures, naming conventions, test procedure format standards, and a test procedure traceability matrix.

The test environment setup is part of test planning. It represents the need to plan, track, and manage test environment setup activities, where material procurements may have long lead times. The test team needs to schedule and track environment setup activities; install test environment hardware, software, and network resources; integrate and install test environment resources; obtain/refine test databases; and develop environment setup scripts and test bed scripts.

### Test Design

The test design component addresses the need to define the number of tests to be performed, the ways that testing will be approached (paths, functions), and the test conditions that need to be exercised. Test design standards need to be defined and followed.

An effective test program, incorporating the automation of software testing, involves a mini-development lifecycle of its own, complete with strategy and goal planning, test requirement definition, analysis, design, and coding. Similar to software application development, test requirements must be specified before test design is constructed. Test requirements need to be clearly defined and documented, so that all project personnel will understand the basis of the test effort. Test requirements are defined within requirement statements as an outcome of test requirement analysis.

After test requirements have been derived using the described techniques, test procedure design can begin. Test procedure design consists of the definition of logical groups of test procedures and a naming convention for the suite of test procedures. With a test procedure definition in place, each test procedure is then identified as either an automated or a manual test. During the test planning phase, the test team gets an understanding of the number of test techniques being employed and an estimate for the number of test procedures that will be required. The test team also will have an estimate of the number of test procedures that will need to be performed manually, as well as with an automated test tool.

Much like a software development effort, the test program must be mapped out and consciously designed to ensure that test activities performed represent the most efficient and effective tests for the system under test. Test program resources are limited, yet ways of testing the system are endless. A test design is developed to portray the test effort, in order to give project and test personnel a mental framework on the boundary and scope of the test program.

Following test analysis, the test team develops the test program design models. The first of these design models, the *test program model*, consists of a graphical illustration that depicts the scope of the test program. This model typically depicts the test techniques required to support the dynamic test effort and also outline static test strategies.

Having defined a test program model, the test team constructs a *test architecture*, which depicts the structure of the test program and defines the way that test procedures will be organized in support of the test effort.

The next step in the test procedure design process (see Table 1) is to identify those test procedures that stand out as being more sophisticated, and as a result are required to be defined further as part of detailed test design. These test procedures are flagged and a detailed design document is prepared in support of the more sophisticated test procedures. Following detailed test design, test data requirements are mapped against the defined test procedures. To create a repeatable, reusable process for producing test procedures, the test team needs to create a document that outlines test procedure design standards. Only when these standards are followed can the automated test program achieve real efficiency and success, by being repeatable and maintainable.

**Table 1 Test Procedure Design Process**

|  |  |
| --- | --- |
| Step | Description |
| 1 | **Test Architecture Review**. The test team reviews the test architecture in order to identify the test techniques that apply. |
| 2 | **Test Procedure Definition (Development Level)**. A test procedure definition is constructed at the development test level, identifying the test procedure series that applies for the various design components and test techniques. |
| 3 | **Test Procedure Definition (System Level)**. A test procedure definition is constructed at the system test level, identifying the test procedure series that applies for the various test techniques. |
| 4 | **Test Procedure Design Standards**. Design standards are adopted and a unique naming convention is adopted that distinguishes the test procedures on the project from test procedures developed in the past or on other projects. |
| 5 | **Manual Versus Automated Tests**. Test procedures will be depicted as being either performed manually or as part of an automated test. |
| 6 | **Test Procedures Flagged for Detailed Design**. Test procedures that stand out as more sophisticated are flagged. These test procedures are further defined as part of detailed test design. |
| 7 | **Detailed Design**. Those test procedures flagged as part of step 7 are designed in further detail within a detailed test design file or document. Test procedure detailed design may consist of pseudo-code of algorithms, preliminary test step definition, or pseudo-code of test automation programs. |
| 8 | **Test Data Mapping**. Test procedure matrix is modified to reflect test data requirements for each test procedure. |

The exercise of developing the test procedure definition not only aids in test development, but helps to quantify or bound the test effort. The development of the test procedure definition involves the identification of the suite of test procedures that need to be developed and executed in support of the test effort. The design exercise involves the organization of test procedures into logical groups and the definition of a naming convention for the suite of test procedures.

At the system level, it may be worthwhile to develop a detailed test design for sophisticated tests. These tests might involve test procedures that perform complex algorithms, consist of both manual and automated steps, and test programming scripts that are modified for use in multiple test procedures. The first step in the detailed design process is to review the test procedure definition at the system test level. This review is conducted for the purpose of identifying those test procedures that stand out as being more sophisticated and that, as a result, are required to be defined further as part of detailed test design.

Detailed test design may take the form of test program pseudo-code, when test programming is required. The detailed design may be represented simply as a sequence of steps that need to be performed in support of a test. When programming variables and multiple data values are involved, the detailed design may reflect the programming construct of a loop supporting an iterative series of tests involving different values, together with a list or table identifying the kinds of data or ranges of data required for the test.

Following the performance of detailed test design, test data requirements need to be mapped against the defined test procedures. Once test data requirements are outlined, the test team needs to plan the means for obtaining, generating, or developing the test data.

The structure of the test program (test architecture) is commonly portrayed in two ways. One test procedure organization method involves the logical grouping of test procedures with the system application design components, and is referred to as a *design-based test architecture*. Another method represents a test technique perspective and associates test procedures with the various kinds of test techniques represented within the test program model, and is referred to as a *technique-based test architecture*.

An understanding of test techniques is necessary when developing test design and the test program design models. Personnel performing testing need to be familiar with the test techniques associated with the white box and black box test-approach methods. *White box* test techniques are aimed at exercising software program internals; *black box* techniques generally compare the application under test behavior against requirements that address testing via established public interfaces such as the user interface or the published application programming interface (API).

### Test Development

For automated tests to be reusable, repeatable, and maintainable, test development standards need to be defined and followed.

After performing test analysis and design, the test team is now ready to perform test development.

Keep in mind that the test design and development activities follow an iterative and incremental approach, in order to address the highest risk functionality up front. Table 2 correlates the development process phases to the test process phases. The testing processes and steps outlined in the table are strategically aligned with the development process, and the execution of these steps results in the refinement of test procedures at the same time as developers are creating the software modules. Automated and/or manual test procedures are developed during the integration test phase with the intention of reusing them during the system test phase.

**Table 2 Development/Test Relationship**

|  |  |  |
| --- | --- | --- |
| **Phase** | **Development Process** | **Test Process** |
| Module (Unit) Development | Design module from requirements. | Perform test planning and test environment setup. |
|  | Code module. | Create test design and develop test data. |
|  | Debug module. | Write test scripts or record test scenario using module. |
|  | Unit test module. | Debug automated test script by running against module. Use tools that support unit testing. |
|  | Correct defects. | Rerun automated test script to regression test as defects are corrected. |
|  | Conduct performance testing. | Verify that system is scalable and will meet performance requirements. |
| Integration | Build system by connecting modules.  Integration-test connected modules.  Review trouble reports. | Combine unit test scripts and add new scripts that demonstrate module interconnectivity. Use test tool to support automated integration testing. |
|  | Correct defects and update defect status. | Rerun automated test script as part of regression test, as defects are corrected. |
|  | Continued performance testing activities. | Verify that system is scalable and will meet performance requirements. |
| System Test | Review trouble reports. | Integrate automated test scripts into system-level test procedures where possible, and develop additional system-level test procedures. Execute system test and record test results. |
|  | Correct defects and update defect status. | Rerun automated test script as part of regression test as defects are corrected. |
| Acceptance Test | Review incident reports. | Perform subset of system test as part of demonstration of user acceptance test. |
|  | Correct defects. | Rerun automated test script as part of regression test as defects are corrected. |

Many preparation activities need to take place before test development can begin. A test development architecture is developed (described in the next section), which provides the test team with a clear picture of the test development preparation activities or building blocks necessary for the efficient creation of test procedures. The test team will need to tailor the sample test development architecture to reflect the priorities of their particular project. Part of these setup and preparation activities involves the need to track and manage test environment set up activities, where material procurements may have long lead times. Prior to the commencement of test development, the test team also needs to perform analysis to identify the potential for reuse of existing test procedures and scripts within the automation infrastructure (reuse library).

The test team needs to develop test procedures according to a test procedure development/execution schedule. This schedule needs to allocate personnel resources and reflect development due dates, among other factors. The test team needs to monitor development progress and produce progress status reports. Prior to the creation of a complete suite of test procedures, the test team performs a modularity relationship analysis. The results of this analysis help to incorporate data dependencies, plan for workflow dependencies between tests, and identify common scripts that can be applied repeatedly to the test effort. As test procedures are being developed, the test team needs to ensure that configuration control is performed for the entire test bed to include test design, test scripts, and test data, as well as for each individual test procedure. The test bed needs to be baselined using a configuration management tool.

Test development involves the development of test procedures that are maintainable, reusable, simple, and robust, which in itself can be as challenging as the development of the application under test. Test procedure development standards need to be in place supporting structured and consistent development of automated tests. Test development standards can be based on the scripting language standards of a particular test tool. For example, Rational's Robot uses SQABasic, a Visual Basic–like scripting language, and therefore the script development standards could be based on the Visual Basic development standards, outlined in a number of books on the subject.

Usually internal development standards exist that can be followed if the organization is developing in a language similar to the tool's scripting language. The adoption or slight modification of existing development standards is generally a better approach than creating a standard from scratch. If no development standards exist within the organization for the particular tool scripting language, it's important for the test team to develop script development guidelines. Such guidelines can include directions on *context independence*, which addresses the particular place where a test procedure should start and where it should end. Additionally, modularity and reusability guidelines need to be addressed.

By developing test procedures based on development guidelines, the test team creates the initial building blocks for an automation infrastructure. The automation infrastructure will eventually contain a library of common, reusable scripts. Throughout the test effort and in future releases, the test engineer can make use of the automation infrastructure to support reuse of archived test procedures, minimize duplication, and thus enhance the entire automation effort.

### Test Development Architecture

Test team members responsible for test development need to be prepared with the proper materials. Test team personnel need to follow a test development architecture that includes, for example, a listing of the test procedures assigned to them and a listing of the outcome of automated versus manual test analysis. Also, test team personnel need to decide when to automate. At times a test team might want to avoid automating using a GUI testing tool before the interface—whether API, character UI, or GUI—is stabilized, to avoid having to reengineer the automated tests in response to non–bug-related changes. At other times, the test team might find workaround solutions when automating an unstable GUI, such as focusing automation on the known stable parts only.

The test engineer needs to adhere to the test procedure development and execution schedule, test design information, automated test tool user manuals, and test procedure development guidelines. Armed with the proper instructions, documentation, and guidelines, test engineers will have the foundation that allows them to develop a more cohesive and structured set of test procedures. Repeating a process and repeatedly demonstrating a strong test program depends on the availability of documented processes and standard guidelines such as the test development architecture.

[Figure 2](javascript:popUp('/content/images/art_dustin2_atlm/elementLinks/dustin2_fig2.gif')) shows an example of a graphical illustration containing the major activities to be performed as part of the test development architecture. Test development starts with test environment setup and preparation activities, discussed earlier. Once they're concluded, the test team needs to make sure that all pertinent information necessary to support development has been documented or gathered. The test team will need to tailor the sample test development architecture in Figure 2 to reflect the priorities of their particular project. Note that [Figure 2](javascript:popUp('/content/images/art_dustin2_atlm/elementLinks/dustin2_fig2.gif')) should be read from bottom to top.

[**Figure 2**](javascript:popUp('/content/images/art_dustin2_atlm/elementLinks/dustin2_fig2.gif')) Building blocks of the test development architecture.

### Technical Environment

Test procedure development needs to be preceded by several setup activities. The test development activity needs to be supported by a technical environment, which facilitates the development of test procedures. As a result, the test environment needs to be set up and ready to go. The test environment includes the technical environment, which may include facility resources as well as the hardware and software necessary to support test development and execution. The test team needs to ensure that there are enough workstations to support the entire team. The various elements of the test environment need to be outlined within the test plan, as discussed earlier.

Environment setup activities can also include the use of an environment setup script to load test data or restore a drive image, and to calibrate the test tool to the environment. When test tool compatibility problems arise with the application under test, workaround solutions have to be identified. When developing test procedures, it's important that the schedule for developing test procedures is consistent with the test execution schedule. It's also important that the test team follow test procedure development guidelines.

The test team must ensure that the proper test room or laboratory facilities are reserved and set up. Once the physical environment is established, the test team ensures that all necessary equipment is installed and operational. The test plan defined the required technical environment and addressed test environment planning. Within the test environment section of the test plan, the test team has already identified operational support required to install and check out the operational readiness of the technical environment. The test team needs to ensure that operational support activities have been properly scheduled and must monitor progress of these tasks.

Specific tasks and potential issues outlined in the test plan should now have been addressed and resolved. Such issues could include network installation, network server configuration and allocated disk space, network access privileges, required desktop computer processing speed and memory, number and types of desktop computers (clients), video resolution requirements, and any additional software required to support the application, such as browser software. Automated test tools that apply should have been scheduled for installation and checkout. These tools now should be configured to support the test team and be operational within the test environment.

The test environment setup activity includes the need to track and manage test environment setup activities, where material procurements may have long lead times. These activities include the need to schedule and track environment setup activities; install test environment hardware, software, and network resources; integrate and test-install test environment resources; obtain/refine test databases; and develop environment setup scripts and test bed scripts.

The hardware supporting the test environment must be sufficient to ensure complete functionality of the production application. Test environment hardware needs to be sufficient to support performance analysis. In cases where the test environment utilizes hardware resources that are also supporting other development or management activities, special arrangements may be necessary during actual performance testing. During system test, the software configuration loaded within the test environment must be a complete, fully integrated release with no patches and no disabled sections. The hardware configuration supporting the test environment needs to be designed to support processing, storage, and retrieval activities, which may be performed across a local or wide area network, reflecting the target environment.

The test environment design also needs to consider stress testing requirements. Stress and load tests may require the use of multiple workstations that will run multiple test procedures simultaneously; some automated test tools include a virtual user simulation functionality that eliminates or minimizes the need for multiple workstations.

Test data will need to be obtained with enough lead time to support refinement and manipulation to support test requirements. Data preparation activities include the identification of conversion data requirements, the preprocessing of raw data files, loading of temporary tables—possibly in a relational database management system format, and the performance of consistency checks. Identifying conversion data requirements involves performing in-depth analysis on data elements, which includes defining data-mapping criteria, clarifying data-element definitions, confirming primary keys, and defining data-acceptable parameters.

During test planning, the test team defined and scheduled the test environment activities. Now the team needs to track the test environment setup activities. Resources need to be identified to install hardware, software, and network resources into the test environment and integrate and test installed test environment resources. The test environment materials and the application under test need to be baselined within a configuration management tool. Additionally, test environment materials may include test data and test processes.

The test team needs to obtain and modify test databases necessary to exercise software applications, and develop environment setup scripts and test bed scripts. The test team should perform product reviews and validation of all test source materials. The location of the test environment for each project or task should be defined within the test plan for each project. Early identification of the test site is critical to cost-effective test environment planning and development.

## Phase 5: Execution and Management of Tests

At this stage, the test team has addressed test design and test development. Test procedures are now ready to be executed in support of exercising the application under test. Also, test environment setup planning and implementation was addressed consistent with the test requirements and guidelines provided within the test plan.

With the test plan in hand and the test environment now operational, it's time to execute the tests defined for the test program. When executing test procedures, the test team must comply with a test procedure execution schedule, as discussed earlier. The test procedure execution schedule implements the strategy defined within the test plan. Plans for unit, integration, system, and user acceptance testing are executed. Together, these testing phases make up the steps that are required to test the system as a whole. The various steps involved during execution and management of tests are outlined below.

* When executing test procedures, the test team needs to comply with a test procedure execution schedule. Following test execution, test outcome evaluations are performed and test result documentation is prepared.
* Plans for unit, integration, system, and user acceptance testing are executed, which together make up the steps that are required to test the system as a whole. During the unit test phase, code profiling can be performed. Traditionally, profiling is a tuning process that determines whether an algorithm is inefficient or a function is being called too frequently. Profiling can discover instances where there is improper scaling of algorithms, instantiations, and resource utilization.
* Integration testing is performed, which focuses on the application internals. During integration testing, units are incrementally integrated and tested together based on control flow. Since units may consist of other units, some integration testing, also called *module testing*, may take place during unit testing.
* During system test, the test engineer is testing the integration of parts that comprise the entire system. A separate test team usually performs system-level tests. The test team implements the test procedure execution schedule and the system test plan.
* The test team also performs analysis to identify particular components or functionality that are experiencing a greater relative number of problem reports. As a result of this analysis, additional test procedures and test effort may need to be assigned to the components. Test results analysis can also confirm whether executed test procedures are proving to be worthwhile in terms of identifying errors.
* Each test team needs to perform problem-reporting operations in compliance with a defined process. The documentation and tracking of software problem reports is greatly facilitated by an automated defect-tracking tool.
* The test team manager is responsible for ensuring that tests are executed according to schedule, and test personnel are allocated and redirected when necessary to handle problems that arise during the test effort. To perform this oversight function effectively, the test manager needs to perform test program status tracking and management reporting.
* Test metrics provide the test manager with key indicators of the test coverage, progress, and the quality of the test effort. During white box testing, the test engineer measures the *depth*of testing, by collecting data relative to path coverage and test coverage. During black box testing, metrics collection focuses on the *breadth* of testing, to include the amount of demonstrated functionality and the amount of testing that has been performed.

## Phase 6: Test Program Review and Assessment

Test program review and assessment activities need to be conducted throughout the testing lifecycle, to allow for continuous improvement activities. Throughout the testing lifecycle and following test execution activities, metrics need to be evaluated and final review and assessment activities need to be conducted to allow for process improvement. The various steps necessary for test program review and assessment are outlined below.

* Following test execution, the test team needs to review the performance of the test program to determine where changes can be implemented to improve the test program performance on the next project. This test program review represents the final phase of the Automated Test Lifecycle Methodology (ATLM).
* Throughout the test program, the test team collected 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. The review also includes an evaluation of earned value progress measurements and other metrics collected.
* As part of its culture, the test team needs to adopt an ongoing iterative process of *lessons learned* activities. Such a program encourages test engineers to take the responsibility to raise corrective action proposals immediately, when such actions potentially have significant impact on test program performance. Throughout the entire test lifecycle, it's good practice to document and begin to evaluate lessons learned at each milestone. The metrics that are collected throughout the test lifecycle and especially during the test execution phase help pinpoint problems that need to be addressed.
* Lessons learned, metrics evaluations, and corresponding improvement activity or corrective action need to be documented throughout the entire test process in a central repository that's easily accessible.
* After collecting lessons learned and other metrics, and defining corrective actions, test engineers also need to assess the effectiveness of the test program to include an evaluation of the*test program return on investment*. Test engineers capture measures of the benefits of automation realized throughout the test lifecycle in order to support this assessment.
* Test teams can perform their own surveys to inquire about the potential value of process and tool changes. A survey form can be used to solicit feedback on the potential use of requirement-management tools, design tools, and development tools. Surveys are helpful to identify potential misconceptions and gather positive feedback.

## Summary

ATLM is a structured methodology geared toward ensuring successful implementation of automated testing. The ATLM approach mirrors the benefits of modern rapid application development efforts, where such efforts engage the user early in the development cycle. The end user of the software product is actively involved throughout analysis, design, development, and test of each software build, which is augmented in an incremental fashion.

Rational Corporation lists [Automated Software Testing: Introduction, Management, and Performance](http://www.informit.com/content/index.asp?product_id=%7bCBB19A2E-D0BE-40B9-9250-15E4D89A61EF%7d&t=%7b94AE5B48-1D7D-462A-A4A6-83CE19EC0705%7d&n=%7b1CBD305F-D503-49FA-9699-8B2C06C9C520%7d) and the ATLM as recommended reading as part of the Rational Unified Process. Many universities and professional software test training organizations have adopted the book for their classroom. Many companies (such as Imbus GMBH, Moehrendorf, Germany) have adopted the book and ATLM as their company standard for automated software testing. Others believe that industry automated test tool vendors will soon be incorporating the book's structured methodology within their tools. Instead of performing the entire test lifecycle haphazardly, software test managers will use an ATLM-compliant test tool that automatically supports (and possibly enforces) the book's sound building-block approach to the test effort.

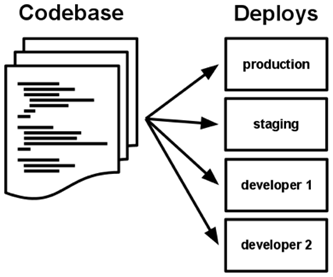
Almost all of the modern cloud or API applications are developed in the new micro-services architecture, there are certain factors or requirements the application should closely adhere to make sure it is cloud ready or can be used to easily expand and change services/behaviors as per market demand. These standards help us achieve applications to be in sync with the [cloud computing definition](http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf) from [NIST](https://www.nist.gov/) as closely as possible to be used and deployed on a cloud infrastructure.

The 12 Factor App requirements make sure that the app can be made infrastructure agnostic, be it cloud or standalone.

From a Quality Engineering perspective, we have to make sure the below standards are maintained and the arrangements of the infrastructure/resources are as per the 12 factor requirements below:

**I. Codebase**

One codebase tracked in revision control, many deploys. Each app should have only one code base which is version tracked with proper version maintenance of the code. One App cannot have different code bases. There can be different deploys of the same app running. Deploys are running instances of the same app. Each deploy can have different version of code depending upon the code build pipeline status.



**II. Dependencies**

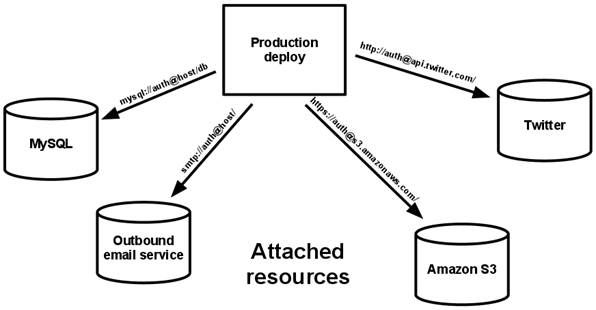
Explicitly declare and isolate dependencies. The dependencies should not be tied implicitly inside app or classes and should be declared in one place for the whole app. Any new developer looking at the app should be able to just pull in dependencies from the dependency management tool and not worry much about pulling in implicit dependencies and should be able to start building and running the app within no time.

**III. Config**

Store config in the environment. Each environment (deploys) should have its own specific variables declared in an environment specific config files. The easy test of it is that the app should be able to be made open source anytime for its config to be kept separated.

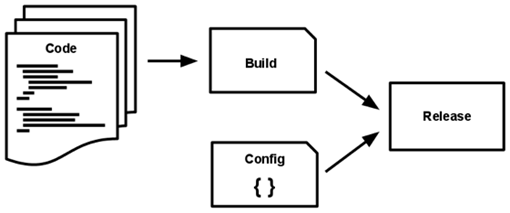
**IV. Backing services**

Treat backing services as attached resources. All the resources whether local DBs or external services should be loosely coupled and can be replaced easily in future if we want to. The backing services should be easily configurable and can be easily replaced in case of failures.



**V. Build, release, run**

Strictly separate build and run stages. The build stage for an app should be separate and be executed before the run(release) stage to make sure the build doesn't have any issue before running the app itself. This helps to isolate the problem and track it easily to fix any issues in lesser time.



**VI. Processes**

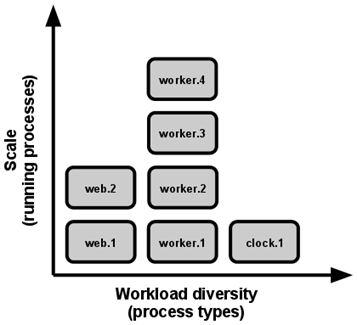
Execute the app as one or more stateless processes. Everything should be stored in a data store and any future processes should not depend on the cached data on filesystem or any place. Each process should be independent on its own and should not depend on data of any other process' state.

**VII. Port binding**

Export services via port binding. Expose apps via URL: port for easy access to other external or internal applications. Each app and its services should be easily accessible via URL endpoint whether it is used by an external or internal API or application.

**VIII. Concurrency**

Scale out via the process model. Each app should have its own process types/groups defined like below to scale out the processes as per the requirements in that particular process type. The array of process types and number of processes of each type is known as the process formation. App should not demonize or write PID files in system, instead it should let the OS handle the processes spun out from the APP which can be distributed over different systems as per the process loads.



**IX. Disposability**

Maximize robustness with fast startup and graceful shutdown. Processes should be able to start up and shutdown in small amount of time and should be able to resume the program execution in minimal amount of time. The system should be robust enough to handle such failures and get back online in no matter of time.

**X. Dev/prod parity**

Keep development, staging, and production as similar as possible. There should be minimal gaps in processes to take app from development to production. The different gaps which need to be minimized are as below:

1. The time gap: A developer may work on code that takes days, weeks, or even months to go into production.
2. The personnel gap: Developers write code, ops engineers deploy it.
3. The tools gap: Developers may be using a stack like Nginx, SQLite, and OS X, while the production deploy uses Apache, MySQL, and Linux.

**XI. Logs**

Treat logs as event streams. A 12 factor app should not worry about storing log files, it should just be writing all its logging events to stdout and let the deployment infrastructure take care of handling logs. The log are events of things happening in the app sorted in a timely manner which need to be routed to appropriate tools to handle logs for analysis and alerting.

**XII. Admin processes**

Run admin/management tasks as one-off processes. All the admin related tasks whether it is in development or production should be executed as on-off processes which once triggered should handle all the admin related task in one go than executing multiple processes for the same.

Depending upon the organization structure and priority, the above factors should be incorporated in an application. Below are some the priorties currently in place for the Cloud Native Architecture requirements:

High = Cloud ready

Medium = Cloud friendly

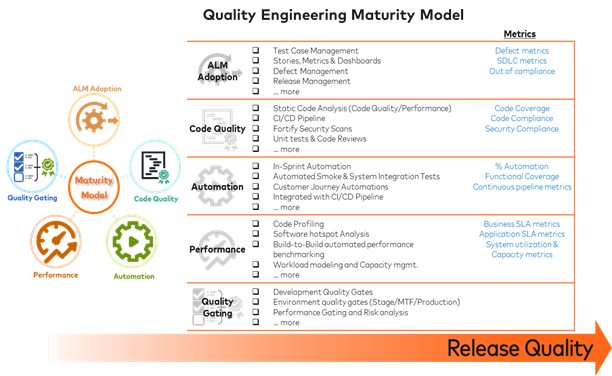
Low = Cloud Resilient

|  |  |  |
| --- | --- | --- |
| Factor | Priority | Description |
| [1. Codebase](https://12factor.net/codebase) | High | One code base tracked in revision control, many deploys |
| [2. Dependencies](https://12factor.net/dependencies) | High | Explicitly declare and isolate dependencies |
| [3. Config](https://12factor.net/config) | Medium | Store config in the environment |
| [4. Backing Services](https://12factor.net/backing-services) | Medium | Treat backing services as attached resources |
| [5. Build, Release, Run](https://12factor.net/build-release-run) | Low | Strictly separate build and run stages |
| [6. Processes](https://12factor.net/processes) | High | Execute the app as one or more stateless processes |
| [7. Port Binding](https://12factor.net/port-binding) | Low | Export services via port binding |
| [8. Concurrency](https://12factor.net/concurrency) | High | Scale out via the process model |
| [9. Disposability](https://12factor.net/disposability) | Medium | Maximize robustness with fast startup and graceful shutdown |
| [10. Dev/Prod Parity](https://12factor.net/dev-prod-parity) | Medium | Keep development, staging, and production as similar as possible |
| [11. Logs](https://12factor.net/logs) | High | Treat logs as event streams |
| [12. Admin Processes](https://12factor.net/admin-processes) | Low | Run admin/management tasks as one-off processes |

References:

<https://12factor.net/>

<http://www.clearlytech.com/2014/01/04/12-factor-apps-plain-english/>



[QE Roadmap Baseline](https://fusion.mastercard.int/confluence/display/TAUTO/QE+Roadmap+Baseline)

[Go to start of metadata](https://fusion.mastercard.int/confluence/display/TAUTO/QE+Roadmap+Baseline#page-metadata-start)



Below is the template for the Agile projects to help define and follow the Quality Engineering Roadmap timelines for specific projects. This will be used to track the progress and impediments of your project.

Project Name :

PO:

Delivery Manager:

Scrum Master:

Dev lead:

SETs:

TEs:

| **Milestones** | **Planned Date** | **Accomplished Date** | **Comments/Blockers** |
| --- | --- | --- | --- |
| Design specifications |  |  |  |
| Initiatives defined in ALM |  |  |  |
| Acceptance Criteria Defined |  |  |  |
| Stake holders defined |  |  |  |
| ALM Compliance Guidelines implemented |  |  |  |
| Test Plan Defined |  |  |  |
| Code Base Setup in GIT/ Bit-bucket |  |  |  |
| SONAR Analysis Setup |  |  |  |
| Test Automation Framework Setup (e.g - REST-Assured) |  |  |  |
| CI/CD Jenkins Pipeline Defined(Test) |  |  |  |
| CI/CD Jenkins Pipeline Defined(Integrated with Dev Pipeline) |  |  |  |
| Jenkins and SONAR webhook integration |  |  |  |
| PCF Cloud Environment Setup |  |  |  |
| PCF and Jenkins Integration Pipeline Setup |  |  |  |
| All ALM initiatives moved to DONE by PO |  |  |  |
| Code freeze done for release |  |  |  |
| Release 1.0 done |  |  |  |
| Product Support Phase |  |  |  |
| Additional Items(if any) |  |  |  |

**Software Engineer in Test (SET)**

Responsible for enabling technical quality. This role helps delivery team with automation strategy, tooling in CI/CD (Continuous Integration/Continuous Delivery), labs, test data set ups, code quality and component level performance checks, security checks, and automation.

* Code Quality

Quality of our products are only as good as the quality of our foundation.  SETs will focus on ensuring proper design patterns, unit testing and peer reviews are conducted on a regular basis.  Target goals should be to get the teams to 80%, or higher,  unit test coverage, zero critical/major errors (Sonar) and establishing a reject process (Build Breakers) for code that does not meet these standards.

* Continuous Integration / Continuous Delivery

Speed and efficiency are critical steps in the quality process.  We need to ensure we have automated checks in place to provide quick responses on code commits.  Once software features/enhancements have been validated as meeting standards they should then be automatically promoted to the next environment for further evaluation.

* Labs

Environment requirements and setup must be determined and established to support automation and performance needs.

* Test Data Setup

Test data requirements for each story needs to be evaluated and documented for each environment where automation will executed.  Plans need to be establish to ensure the correct data is in place for automation execution.  Tooling should be leveraged, as needed, to accomplish this task.

* Performance Checks

Similar to automation, performance tests (both unit and functional) need to be established to ensure we are on target to meeting/exceeding SLAs.

* Security Checks

Scans and tests need to be established and executed to ensure vulnerabilities do not hinder Mastercard or it's customers.  SETs should help establish scanning processes using tools (Fortify) and work with Information Security to established penetration testing.

* Automation

Creating/supporting automation is integral to the SET role.  They should coaching (peer reviews), supporting (page object creation), and assisting in writing automation throughout the lifecycle of the project.

[Click here to view daily checklist.](https://fusion.mastercard.int/confluence/display/TAUTO/Software+Engineer+in+Test+%28SET%29+Daily+Tasks)

**Test Engineer (TE)**

Responsible for customer quality. This role works with product owners and puts focus on business feature acceptance and customer journeys. TEs are also responsible for integration testing, ALM best practices, and automation programming.

* Feature Acceptance

TEs will work with the Product Owner to ensure requirements have been met and suitable validations are in place.

* Customer Jouneys

Focus on the customer journeys is essential to ensure our products meet the needs for our customers.  These end-to-end tests should cover critical paths end users will face when interacting with the product.

* Integration

Checks (stuff in/stuff out) should be in place to ensure live integration of other products/services are performing as intended.

* ALM Best Practices

Test Cases, Test Sets and Test Folders will be managed by the TE to properly map scenarios with the role they play.

* Automation

Writing automation to cover all these scenarios so they can be executed as part of each quality gate as software migrates from development to production.

[Click here to view daily checklist.](https://fusion.mastercard.int/confluence/display/TAUTO/Test+Engineer+%28TE%29+Daily+Tasks)

**Performance Engineer (PE)**

Responsible for end-to-end performance strategy, performance testing, capacity modeling, tuning, and SLAs compliance. A senior performance consultant will be assigned at program level to help with architecture analysis and tuning.

* Performance Strategy

A strategy is documented to ensure the proper tools are being identified allowing us to best gauge and monitor product performance throughout the lifecycle of the product.

* Performance testing

Tests are created (smoke, stress, load, capacity, soak, spike and break tests) to validate various scenarios that can/will occur once a product is publicly available.

* Capacity Modeling

Documented needs of the product, for scaling purposes, over the next 12 months, 1 year, 3 years and 5 years of the product.

* Tuning

Validating configurations of supporting systems to ensure they are operating at peak performance (web servers, JVMs, etc).

* SLA Compliance

Monitoring system performance to meet, and exceed, established SLAs.  Helping isolate and resolve discovered issues.