System Test Pattern Language

David E. DeLano and Linda Rising,
AG Communication Systems
2500 W. Utopia Rd.
Phoenix, AZ 85027
delanod@agcs.com, risingl@agcs.com
(602) 581-4679, (602) 581-4699

Introduction

Testing of systems is presently more of an art than a science, even considering current procedures and methodologies that support a more rigorous approach to testing. This becomes even more apparent during the testing of large, embedded systems that have evolved over time. To deliver the best possible product, the role of a System Tester has become vital in the lifecycle of product development. This pattern language has been derived from the experience of veteran System Testers to help System Testers evaluate the readiness of a product for shipping to the customer. Though these patterns have been derived from experience during the system test phase of the product lifecycle, many of the patterns are orthogonal to all testing phases.

These patterns have been grouped according to their usefulness in the system testing process: The Test Organization, Testing Efficiency, Testing Strategy, and Problem Resolution.

Context

As a pattern language, these patterns share a common context. Each of the patterns may add to this context or slightly modify it, but it is this common context that helps focus these patterns into a pattern language. The context for an individual pattern will only be given when it modifies the common context.

A system is under development into which new features are being introduced. The system may be new development work, or may be an existing, working system. There are a limited number of customers for the system and the features are being developed at the request of these customers. The customers then sell the features to the end users. New features integrate well into the system, but introducing new features runs the risk of breaking old features. These features are implemented in the system by Designers in a design group.

The features of the system are designed in parallel. As code becomes available, incremental loads of the system are released for testing. Each of these loads may include a combination of complete or partial features. This process compresses the development schedule, but it also results in the introduction of problem resolution and features in parallel. Problems resolved in one load need to be resolved in any feature development that is in progress.

The system, in this case, is a large embedded system. It consists of multiple processor units and each processor is controlled by a multi-processing operating system. Events in the system are thus non-deterministic. Any feature of the system can have an interaction with any other feature in the system. Most of these patterns apply to the testing of any multi-processing, non-deterministic system. The set of all tests needed to completely verify such a system is infinitely large. The set of regression tests to reasonably test the system grows with every release, so that executing all of the tests would require more than the development schedule allows for a new feature release.

The features are being tested by an independent group known as System Test whose members are known as System Testers. The testing done by this group is largely “black box” in nature. There is a reliable process in place for testing the system, from unit testing to system testing. The System Test group is responsible for regression testing the system, the conformance of new features to the specification, the equivalence of the system to prior releases, and the evaluation of the release of the system according to criteria set by the customer. These criteria are consistent with every release. System testing concentrates on the functionality of the system and ensures that the system continues to operate in a consistent manner. To accomplish this, the testing often takes the form of stressing or breaking the system.

These patterns do not represent the process of system testing any more than the GOF book [Gamma+95] represents object-oriented design. Instead, they are a means of sharing some of the secrets of experienced System Testers.

Forces

Just as these patterns share a common context, they also share common forces. The individual patterns may place more emphasis on specific forces, ignore some forces or add forces that are not important to the pattern language as a whole. Forces for an individual pattern will only be given in the case where they are modified or emphasized.

Most of the forces for these patterns are elements of risk management. The patterns evaluate these risks to find an acceptable, intelligent balance among cost, time, content and quality. The common forces are:

- The scheduled interval for testing is short.
- Specifications are often unclear or ambiguous.
- Problems should be found and corrected as early as possible.
- Testing resources, such as prototype systems or manhours, are at a premium.
- Software load stability is critical to progress.
- The content of a given software load is uncertain.
- Designers are preoccupied with new development.
- Designers are primarily concerned with their own area of development.
- End users don’t always use features as specified by the customer.
- Compressing the development schedule risks introducing problems.
- Compressing the test interval risks not finding all critical problems.
- Not all problems can be found.
- A release with some non-critical problems is acceptable.

The Test Organization

The following patterns focus on the relationship of the System Testers to the rest of the organization. They are less technical in nature and are closely related to the patterns of James Coplien [Coplien95], Neil Harrison [Harrison96], Alistar Cockburn [Cockburn96], and Don Olson [Olson95].

Pattern: *The Tester's More Important Than The Test Cases*

Problem How should work be assigned to Testers to achieve maximum testing efficiency?

Forces

- All testers do not have the same capabilities.

Solution Assign tasks to System Testers based on their experience and talent.

Resulting Context Testers will be more effective if their skills are used appropriately. The resulting test activity will be more efficient and the testers will be more productive.

Beware, however, that repeated applications of this pattern can result in burnout of the Tester or the development of irreplaceable expertise. Testers who become too familiar with an area often overlook problems by making invalid assumptions. Career development often gets overlooked when existing skills are always used. No new skills are developed.

Rationale The Tester is just as important to the success of the project as the Designer. The Tester is the most important element in the testing equation. The same test case will not necessarily produce the same result in the hands of different Testers. Testers look at a list of errors differently. Some Testers know how to break the system vs. getting a correct result. Some Testers are good at finding problems. Some Testers are good at working with Designers. Matching the skills of the Testers with the appropriate tasks will produce more effective testing.

DeMarco and Lister [DeMarco+87] documented that there is a wide variance in performance between the worst and best performers for a given task. In order to be successful, the best performers should be assigned to the most critical tasks, and everyone should be given an assignment that best fits their skills.

Pattern: *Designers Are Our Friends*

Problem How can System Test work effectively with the Designers?

Solution Good System Testers build rapport with Designers. Approach Designers with the attitude that the system has a problem that both of you need to work together on to resolve. This gives the Designer and the System Tester a common goal. Be certain to always *Document the Problem*.

Resulting Context The System Tester/Designer relationship becomes one of cooperation in an attempt to resolve a problem. Designers are less likely to become defensive and "run and hide" when a System Tester approaches.

Rationale We are all employees of the same company and should work with each other to solve problems. Designers and System Testers are partners. Building good relationships with Designers gets things done faster. Personalizing problems makes Designers defensive. We can learn from each other and benefit both.

Related Patterns *Document the Problem* helps the System Tester to provide the Designer with the information needed to resolve a problem.

There is some commonality with *Don't Flip the Bozo Bit* [McCarthy95, Rising96].

Pattern: *Get Involved Early*

Problem How can System Test maximize the support from the Designers?

Forces

- During early development phases, System Testers have test plans to write.
- System Testers have a broader view of the context of the system.
- Impacts of social relationships aren't measured by the project.

Solution Establish a good working relationship with the Designers early in the project. Don't wait until you need to interact with a Designer to develop a working relationship. By that time it is too late. Trust must be built over time.

One way to accomplish this is to learn the system and the features at the same time the Designer is learning them. Attend reviews of the requirements and design documentation. Invite the Designer to test plan reviews.

Resulting Context When a good working relationship is built over time, it is easier to resolve problems when they are discovered.

Be sure that the relationship with the Designer does not affect your judgment as an effective System Tester. There can be a tendency to avoid areas of conflict when friends are involved, to look the other way when problems are found in a certain area where the Designer is a close friend. It is also dangerous to have too much information about an area. This can lead to certain kinds of testing that depends on this knowledge instead of an objective black box view.

Rationale We are all more willing to work with people we know well. Waiting until the heat of battle to get to know the people who can help you resolve problems leads to delays in solutions. If a trusting relationship has already been established, the problem solving process is smoother.

Related Patterns This pattern is closely related to *Designers Are Our Friends*.

Pattern: *Time to Test*

Alias *Test When There's Something to Test*

Problem When is it appropriate to start testing?

Context The system can be broken into areas, such as features or functional areas, which are minimally dependent on each other. Ideally, once an area is released, it would not be impacted in future software loads.

Forces

- Designers don't want testing to start until everything is "perfect."
- Testers want to start testing early.
- Testing an incomplete system may require retesting on later software loads.

Solution Ideally, system testing should not start until all of the software is finished. However, waiting until everything is complete leaves no time for doing all of the system testing. Start testing when an area of functionality is available for testing, but not before. An agreement needs to be reached with the Designers that the area is ready for testing (see *Designers Are Our Friends*). Track areas that aren't ready for testing so you don't waste time in those areas.

Resulting Context More interval is available for testing if testing begins on early increments of the software loads. This also exposes the load to more testing, which will uncover more problems. More critical problems will be discovered earlier, when there is time to correct them.

By starting early, there is a risk that regression testing of the area will need to be done. This testing should follow *End User Viewpoint*.

Rationale System Test is often pushed to start testing as early as possible. However, starting to test the system too early is a waste of time. It is possible, however, to test parts of the system that are ready earlier than others.

Related Patterns *Take Time* must be considered when determining whether an area is ready for testing.

Pattern: *Take Time*

Problem How much time should Designers be given to finish work that is behind schedule?

Context Testing for an area is scheduled to start. Design for the current release is not complete. This pattern should be applied in the context of *Designers Are Our Friends*.

Forces

- Designers always want more time.
- More effort up front usually produces a better product in the end.

Solution Give Designers the time they request, within reason. Use *Time to Test* to find something else to do in the mean time.

Resulting Context System testing will take less time if the design quality is better.

Rationale Higher quality systems take less time to test than poorer quality systems. By giving Designers time they truly need, there will be a significant pay-back in the end in avoiding the effort to test and fix all the problems in a poor quality system.

Increasing planned development time 15% cuts number of defects in half [Remer96].

Note that all the forces of *Take Time* and *Time to Test* must be carefully weighed before deciding which pattern to follow.

Testing Efficiency

As stated in the context, it is not possible to completely test a system. The following patterns steer the Tester toward areas that are more likely to contain errors. Test plan development and testing can concentrate on these areas to find more problems earlier in the project.

Pattern: *"Unchanged" Interfaces*

Problem A project fails to consider the impact of scheduling little or no testing of third party interface functionality developed outside the company and incorporated into the new product.

Context Interfaces developed outside the company are used in the system, in whole or in part, without changes to that interface. The initial assumption is that if third party interface functionality is to be supported in the new system and if the same interface has been used in other systems and has not changed, then no testing or very limited testing is required.

Forces

- Time can be saved by no or limited testing of unchanged third party interface functionality.
- Limited knowledge, training on third party software.
- Feeling that unchanged code will function the same in the new product .

Solution The System Tester must not fall into the trap of assuming that unchanged interfaces will function correctly in the new system. The System Tester must pay particular attention to any interface ported from outside the company that is not assigned for inclusion in the feature list of the new system. Since no one else is directly impacted by this change, the System Tester needs to be pro-active in scheduling this testing.

Testing the correct operation of the interface on the new system should begin as soon as that interface is chosen for the new system. This will verify the functionality of that interface. Once the ported interface is operational in the new system, testing should begin immediately to ensure that the functionality is the same and that it does not affect the rest of the system. This can be done while the Designers are working on the new features for the system. As a System Tester you must push the project to look at these areas in greater detail. In most cases, you will have to assume the responsibility for this testing and plan for it in your schedule, in conjunction with your other duties.

Resulting Context System Test will find problems with ported interfaces early in the project's life cycle and limit the impact of any problems on the system release. This will allow focused effort on fixing the problems while there is still time to change how the system functions, what is included in the system, or when a system is ready to be released. Less overall effort and stress will be expended during system development, time will be saved and the system will be released as planned.

Rationale No matter how established a product is or how well written a chunk of code is, if it is not an in-house product there will always be problems with porting that functionality into a new system. If testing of the functionality is not immediately begun, problems will invariably be found at the worst possible time–right before release of the system.

Any product brought in from outside the company will not meet the quality, functionality, or customer expectations of the company. Thus, changes will be made.

Pattern: *Ambiguous Documentation*

Problem How can possible problem areas of the system be pinpointed so that the most problems can be found in the least amount of time?

Context Feature design documentation and/or user documentation is available. This can be the requirements from which the feature is developed, documentation developed by the Designer, or documentation developed for the customer.

Forces

- Some areas of a system are more likely to have problems than others.

Solution Study the documentation available on the system. Look for areas that seem ambiguous or ill-defined. Write test plans that cover these areas more thoroughly and concentrate testing in these areas (see *End User Viewpoint*). If Designers can tell you all about a feature, it probably works. It's what they can't tell you that needs your attention during testing.

It is also advantageous to raise these areas to the attention of the Designers and Systems Engineers so that problems in these areas can be resolved prior to system testing. These areas should still be tested more thoroughly than areas that are well documented and well understood.

Resulting Context Problems are likely to be found and corrected earlier.

Rationale If there's more than one interpretation of the documentation, Designers will write more than one code interpretation. This must be uncovered during system test if it is not detected earlier. If you are involved early, there is a better chance of getting documentation fixed early, and thus the implementation correct. If problems still exist, finding them early makes it more likely that they will be fixed. Good specs help everyone!

Related Patterns *Get Involved Early*, *Designers Are Our Friends*

Pattern: *Old Problem Reports*

Problem What areas of the system should be tested first so that the most problems can be found in the least amount of time?

Context Testing of existing features is being considered. Problem reports from previous releases are available.

Solution Use problem reports from previous releases to help select regression tests. Since it would be inefficient to retest for all old problems, look at problems reported after the last valid "snapshot" of the system. Categorize problem reports to see if a pattern is determined that could be used for additional testing.

Resulting Context Problems that still exist will be found and corrected.

Rationale Since a problem report represents something that escaped in a previous release, this could be a good indicator of a problem in the current load. Problem reports tend to point to areas where problems always occur. Problem reports often represent a symptom that is difficult to connect with an underlying problem.

Additionally, fixes of a previous release are often done in parallel with new development on the current releases. These fixes don't always find their way into the current load.

Pattern: *Problem Area*

Problem What areas of the system should receive concentrated testing, regardless of the features being implemented?

Context Historical data is available for the system under test.

Solution Keep a list of persistent problem areas and test cases to verify them, not just for resolution of the current problems, but also for use in subsequent testing. Retest regularly using these test cases, even one last time before the release goes out the door.

Identify areas of the system that historically have problems. Test these areas thoroughly, even if there are no new features going into them.

These areas can be identified by considering: the experience of the System Testers; documented evidence from previous releases (see *Old Problem Reports*); documentation following *Ambiguous Documentation*; observing areas that Designers tend to avoid working in from one release to the next.

Resulting Context By testing problem areas early in the schedule, more time is available to correct any problems found.

Rationale Areas of the system that historically have problems don't magically become error-free overnight. Some problems occur in every incremental release of the system. Some areas have problems in every release. These problems and problem areas should be tracked so that they can be systematically retested on every release of the system.

Testing Strategy

Once a test plan has been set into place and testing has commenced, patterns have been followed that result in finding problems that might not be found until it is too late to correct them. These patterns also help ensure that the delivered product will be acceptable to the customer. The following patterns address the question, "How should the system be tested?"

Pattern: *Busy System*

Problem Under what conditions should system tests be executed to find the most problems?

Context Simulators exist that provide a reasonable amount of activity on the system.

Solution Test in an environment which simulates a busy system. The level of activity need not stress the system, but should approach a level that the system regularly experiences.

Resulting Context Things that work properly under normal conditions may break in a busy system.

Rationale Tests that work fine under normal conditions often fail in an unacceptable manner when the system is under stress.

Related Patterns Be sure to follow *Don't Trust Simulations*.

Pattern: *Don't Trust Simulations*

Problem How should the test environment be configured when using test simulations?

Context Simulations of system use are available, including simulators of real uses of the system.

Forces

- Simulators are often a more accessible testing environment.
- Some testing cannot be accomplished without using simulations.
- It's impossible to have a real-world environment for all testing.

Solution Try to test in an environment as close to the real world as possible. However, system behavior may be correct when using a simulation, but not work at all in the real world. Apply simulations appropriately but don't use them as a substitute for real world testing.

Resulting Context Systems that handle simulations, which tend to give predictable input to the system, often fail in the real world of unpredictable behaviors. By supplementing the simulation with real world testing, such situations are minimized.

Rationale There is a proper use for simulations, but they are not substitutes for real-world testing. While a large portion of the testing can be done using simulations, testing of the system is not complete without providing real world scenarios in the testing process.

Related Patterns *End User Viewpoint* provides hints at how to accomplish real world testing.

Pattern: *End User Viewpoint*

Problem How do you test a system without repeating testing that has already been completed?

Context Testing of the new features has already been completed. The feature is being retested as part of the system.

Forces

- Duplicate testing takes more time in the schedule.
- Designers can only tell you what they know about a feature.
- There is a perception that things that haven't changed don't need to be tested.
- System test must attempt to look at everything.
- System test must find what's missing.

Solution Test outside the normal scope of the features. Take the end user's point of view. Don't system test with the same tests used for feature testing.

Resulting Context By testing from an end user viewpoint, flaws in the system, as the end user will use it, can be discovered and corrected before the system is shipped. These tests expand the scope of previous testing and are not redundant.

Rationale End users don't use systems as they are designed. Designers develop features, but the system is sold to provide services to the end user. It is these services that the end user sees, not the features that are developed by the Designers.

Pattern: *Unusual Timing*

Problem What additional testing should be done that may not be covered by the test plans for an area?

Context Features that have previously passed feature testing are being system tested.

Solution Test unusual timing. Run tests more quickly or slowly than expected. Abort tests in the middle of execution. Real end users will often use the system in this manner (see *End User Viewpoint*).

Resulting Context Errors detected by unusual timings are detected and corrected.

Rationale Things that work properly under normal timing conditions may break under unusual timing conditions.

Testing for unusual timing scenarios is often difficult to set up and run. Test cases that are difficult to run are the ones that probably need to be run the most.

Related Patterns *Busy System* is another alternative to finding problems in a working feature.

Pattern: *Multiple Processors*

Problem Once testing is started, **w**hat is a good strategy for determining what to test next?

Context The system is composed of multiple processors. Portions of the software are replicated on multiple processors.

Solution Test across multiple processors.

Resulting Context Problems that occur in one processor will probably occur in other processors. Tests that pass on one processor may fail on another.

Rationale When a problem is found in one processor, that feature will usually have problems running on other processors. A dirty feature is a dirty feature.

Features that run on multiple processors aren't always designed to run on all processors.

Related Patterns This pattern is similar to *Scratch 'n Sniff*.

Pattern: *Scratch 'n Sniff*

Alias *Problem Cluster*

Problem Once testing is started, **w**hat is a good strategy for determining what to test next?

Context A problem has already been found in an area.

Solution Increase the testing in areas where problems have already been found.

Resulting Context Problematic areas will be targeted sooner and problems resolved earlier.

Rationale Problems tend to be found in clusters. A problem found in an area is a good indicator of other problems in the same area. *Scratch 'n Sniff*: if it smells bad, it probably is.

Testing can be organized so that a "first pass" is taken which tests the breadth of the system. Deeper testing of the areas with problems will likely find more problems.

Related Patterns *Old Problem* Reports is similar in that past reported problems indicate areas which may need more testing.

Pattern: *Strange Behavior*

Problem What should be done when a feature is working, but not as expected?

Context The System Tester has participated in the system testing of previous releases of the product and is familiar with system behavior.

Solution Take any unusual behavior as an indication of a possible problem and follow up. This should be done even if the problem is not related to the test being executed. Look for features that behave differently. Be wary when familiar tests produce results that, while acceptable, are not what was expected.

Resulting Context A problem won't be missed because it doesn't produce system behavior that is not significantly different from the expected outcome.

Rationale Changes to the way a feature works, even though the feature may still work correctly, often indicate a change to the system that may have broken an existing feature. If the change is a deliberate change by a Designer, all System Testers should be notified.

Pattern: *Killer Test*

Problem How can the quality of a system under development be determined?

Context Development is drawing to a close. The system is stable. The features are stable and all parties involved, especially management, are interested in how close the system is to being ready for release.

Solution Develop a favorite killer test that can be run at any time, a test that always seems to find problems. This test, or set of tests, should provide good system coverage and should be expected to fail, in some manner, most of the time. It is permissible to borrow a killer test, but it is preferred that the test be based on the tester's own experience, as the effectiveness of the test is dependent on the individual's skill and knowledge of the system.

Killer Test is only used toward the end of a release, and is above and beyond a common regression test. This test is a combination of regression and break-it testing, and tends to be free-wheeling in nature and typically hard to document. Because the success of this type of testing is directly related to the individual running the test, the individual should be carefully selected using *The Tester's More Important Than The Test Cases*.

While killer tests don't always find the same problem every time they are run, they usually find some problems. Don't wait until the very end of the release, however, or there won't be time to correct the problems found.

Resulting Context The results of the tests are a good gauge of the stability of the system. By finding and fixing the uncovered problems, the system becomes incrementally more stable.

Rationale A test that usually fails and can be run in a reasonably short time gives a good measure for how the system is stabilizing. Additionally, since problems are likely to be found, this type of testing is highly efficient.

Problem Resolution

The following patterns aid in communication and resolution of problems.

Pattern: *Document The Problem*

Problem How should problems found in testing be communicated?

Context A problem tracking system is available for problem documentation.

Forces

- Testers want to be sure that problems are fixed.
- Designers have good intentions but don't always get problems resolved.
- Problem tracking systems are often cumbersome.
- Problem tracking systems make problem areas very visible.
- People inherently avoid documentation.
- Problem reports are often used as indicators of developers competence.

Solution Write a problem report. Don't argue with a Designer. Don't accept a well-intentioned promise that may or may not get results. Don't informally document the problem.

It is also very important that Testers are not penalized for documenting problems. In the same manner, Designers should not be punished when problems are documented against them.

Do your homework before reporting problems to Designers. Be sure you can explain what happened. Designers always want to see a system debug output.

Resulting Context Problems documented in a problem tracking system are more likely to get resolved in a timely manner.

Rationale Use all the tools that are available, even if they are cumbersome. By thoroughly and officially documenting a problem, information does not get lost and a timely resolution is much more likely to occur.

Documenting problems is always an area that a project wants to cut back. Cutting back, however, causes many more problems than it seems to solve.

Related Patterns *Designers Are Our Friends* and *Get Involved Early* hint at how to handle the System Tester/Designer relationship.

Pattern: *Adopt-A-Problem*

Problem How can nagging problems be resolved efficiently so that productive testing can continue?

Context A problem has been uncovered in the system that has no clear cut solution. Resolving the problem will most likely take a great deal of effort. This pattern should be followed in the context of *Designers Are Our Friends*.

Forces

- Some problems are difficult to reproduce.
- Ambiguous problems result in ambiguous fixes.
- Designers don't have time to track down problems they may not be able to resolve.

Solution Adopt a problem. Treat it as if it were your child. If you uncover a difficult problem, stick with it until it is resolved. Ensure that *Document the Problem* has been followed. Retest the problem periodically to gather more data on it. Become the "feature chief" for the problem. Once the problem is resolved, retest for it periodically to ensure that it stays fixed. If the problem reappears consistently, follow *Killer Problem*.

Resulting Context Following the progress of the solution for a problem shows the Designer that you are concerned about getting the problem resolved. Taking an active role in resolving the problem can prevent the problem from bouncing among Designers. Periodically retesting the problem leads to a better understanding of the problem and more symptomatic data. As a result, the probability that the Designer can solve the problem and provide a solution in a timely manner is increased.

Rationale Designers have a multitude of things to do, including new development and resolving problems found during testing. Designers tend to work first on things that are known, concrete, and easy. Because of this, trying to resolve a difficult problem often gets pushed to a lower priority. By adopting the problem, there are in effect two people working on the problem. As the System Tester, you can continue to test and debug the problem to gain more information on it, be it information that the Designer requests, or information that appears to be different from previously collected data. The attention you give to the problem also communicates to the Designer that you think it is important to get the problem resolved and are willing to help in facilitating the problem resolution. This second point is important, because you should not come across as a nagging irritant that won't go away until the problem is resolved, but as a willing participant in the problem resolution.

Related Patterns It is important that *Designers Are Our Friends* be followed to ensure the success of this pattern. *Killer Problem* should be followed if the problem tends to recur. Pay close attention to *Pet Peeve*.

Pattern: *Pet Peeve*

Problem The validity of a problem has been debated to the point of holding up progress. What should be done to resolve the debate?

Context This problem should be applied in the context of *Adopt-A-Problem. Document the Problem* has been followed.

Solution When a problem is adopted, be sure that the problem doesn't become an annoying thorn in the Designer's side. Don't carry concern for an unresolved problem to extremes, especially when existing documentation is not on your side.

When the status of the problem becomes a detriment to progress, the System Tester should bring in a third party, such as a Systems Engineer or requirements group, to help resolve the impasse. At this point the System Tester should stop following the status of the problem and start testing other areas. Don't involve the third party too early in the process of trying the resolve the problem.

Resulting Context Problems are resolved and not forgotten but no one goes overboard in focusing on one problem, to the detriment of the rest of the system.

Rationale When following *Adopt-A-Problem*, a System Tester can become so focused on a particular problem that it becomes a Pet Peeve. Carried to an extreme, this can destroy a good working relationship between the System Tester and the Designer. Problems in the system should not be taken personally by the System Tester.

Known Uses

The individuals involved in the mining of these patterns have many years of experience in System Testing on the GTD-5® and other projects at AG Communication Systems. The GTD-5® is a central office telephone switch that has gone through many major releases over the past 16 years. The other projects have benefited from this experience and validate the existence of these patterns. Some releases of the GTD-5® and other projects have suffered the consequences of not applying one or more of these patterns. In addition, these patterns have been observed and used by some of the individuals at other companies involved in the development of real-time embedded systems. More information on AG Communication Systems and its product line can be found at `http://www.agcs.com`.

References

[Cockburn96] A. Cockburn. "A Medical Catalog of Project Management Patterns." Submitted for PLoP96 (URL: `http//members.aol.com/acockburn/index.html`), 1996.

[Coplien95] J. O. Coplien. "A Generative Development—Process Pattern Language." In J. O. Coplien and D. C. Schmidt (eds.), *Pattern Languages of Program Design*. Reading, MA: Addison-Wesley, 1995, pp. 183-237.

[DeMarco+87] T. DeMarco and T. Lister. *Peopleware: Productive Projects and Teams*. New York, NY: Dorset House, 1987, pp. 45-47.

[Gamma+95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[Harrison96] N. B. Harrison. "Organizational Patterns for Teams." In J. M. Vlissides, J. O. Coplien and N. L. Kerth (eds.), *Pattern Languages of Program Design 2*. Reading, MA: Addison-Wesley, 1996, pp. 345-352.

[McCarthy96] J. McCarthy. "#4: Don't flip the bozo bit." In McCarthy, Dynamics of Software Development. Redmond, WA: Microsoft Press, 1995, pp.23, 30, 32.

[Olson95] D. Olson. "Don Olson's Patterns on the Wiki Wiki Web." Portland, OR: Portland Patterns Repository (URL: `http://c2.com/cgi/wiki?search=DonOlson`), 1995.

[Remer96] D. Remer. "Cost and Schedule Estimation for Software Development Projects." UCLA, 1996, pp. SW 11-8, Guideline #35.

[Rising96] L. Rising. "Don't Flip the Bozo Bit." Champaign, IL: Patterns Home Page (URL: `http//st-ww.cs.uiuc.edu/users/patterns/bozobit.html`), 1996.

Acknowledgments

These system test patterns were mined in a patterns mining workshop held on May 10, 1996. The patterns were captured by David DeLano and Linda Rising with Greg Stymfal serving as facilitator for the group. The workshop was attended by the following AGCS System Testers: Dave Bassett, Arvind Bhuskute, Bob Bianca, Ray Fu, Hubert Fulkerson, Eric Johnstone, Rich Lamarco, Krishna Naidu, Ed Nuerenberg, and Lori Ryan. In addition, the following AGCS System Testers have participated in reviews of the patterns: John Balzar, Terry Bartlett, Carl Gilmore, Neil Khemlani, Jim Peterson, Mike Sapcoe, and Frank Villars. Thanks to Dave Strand for suggesting the name *Scratch 'n Sniff*.

"Unchanged" Interfaces has been adapted from a pattern written by Mike Sapcoe during an earlier Patterns Writing course held by David and Linda.