Envisioning the Next Generation of Functional Testing Tools

Jennitta Andrea, *clearStream*

A successful functional test-driven development strategy relies on effective tools across the application life cycle. What will next-generation tools look like?

The functional test-driven development (FTDD) cycle moves functional test specification to the earliest part of the software development life cycle.[1] Functional tests no longer merely assess quality; their purpose now is to drive quality. For some agile processes such as Extreme Programming, functional tests are the primary requirements specification artifact.[2] When functional tests serve as both the system specification and the automated regression test safety net, they must remain viable for the production code's lifetime.

Compared to production code, functional tests must be

- *Easier to write*: If writing them is a bottleneck to writing production code, they'll be considered optional and quickly become incomplete and obsolete.
- *More correct*: Because bugs in functional tests will create or mask bugs in production code, functional tests drive what's developed and are used to detect bugs introduced over time as the production code changes.
- *More readable*: Nontechnical subject matter experts must be able to validate the functional tests' correctness and completeness.
- *More easily and safely maintained*: Functional tests don't have the same type of regression safety net as production code.
- *More locatable*: All the relevant functional tests must be found and updated before the production code can be updated.

Although we need discipline and skill to execute FTDD to these standards, tool support is also absolutely essential for long-term success. Most projects don't have the time or money to include toolsmithing in the budget, so either they compromise their standards by letting the tool they have dictate their approach or they reduce their effectiveness with workarounds. For FTDD to be successful over the long term, we need a new generation of functional testing tools that will help teams realize its true power and effectiveness. To this end, this article reflects on FTDD teams' core tasks performed over the full application life cycle. It then envisions a conceptual functional testing framework and a concrete list of capabilities that will satisfy these needs.

0740-7459/07/$25.00 © 2007 IEEE

Current automated tool support

Searching the Internet reveals a significant number and variety of automated functional testing tools. How well do they support the TDD process?

- Record/playback testing tools predate FTDD and were built to support automated testing after development. Test scripts, generated by the tool while recording a set of user actions, tend to be difficult to read and maintain.
- TDD's introduction launched a new breed of testing tools, collectively known as xUnit (for example, SUnit for Smalltalk, JUnit for Java, and NUnit for .NET). These frameworks support tests written by developers in an existing programming language, using existing integrated development environments (IDEs). Over time, developers wrote other extensions (such as HttpUnit and JwebUnit) to support testing Web-based user interfaces. Although these tools are well suited for unit testing, they lack the readability and locatability required for functional testing.
- Framework for Integrated Test, or Fit, tools revolutionized FTDD by providing expressive, readable, tabular functional tests and detailed, visual context-sensitive reporting. Fit provides great flexibility by separating tests from the code that executes them (*fixtures*). The FitLibrary further enriches the specification of the business domain and workflow.[3] The FitNesse framework brings functional tests a step closer to subject-matter experts by enabling Fit tests to run in standard browsers. Although progressive, these tools are hard to write and maintain because no functional test development environment exists for tabular specifications.
- Many other tools introduce different capabilities and testing languages (for instance, WebTest tests are specified in XML and Watir tests are specified in Ruby), but they don't advance the state of the art much beyond the xUnit framework.
- Behavior-driven development (http://behaviour-driven.org) has been a significant evolutionary step forward by shifting focus toward specifying business-oriented behavior instead of tests. The emphasis is on establishing an unambiguous domain-specific testing language, resulting in declarative specifications. While BDD tools such as jbehave and rspec bring the state of the art in a new direction, we must keep pushing forward and adding more power to these types of tools.

Deficiencies such as these have forced FTDD teams to enhance the functional testing tools they use. For example, I enhanced the WebTest framework to significantly improve the readability and maintainability of the XML-based specification language.[4] I also added a spreadsheet front end to FitNesse to enhance the creation and maintenance of tabular tests for complex domains. I then integrated the spreadsheet-based FitNesse with the Quick Test Professional record/playback tool to support test-first porting of legacy systems. Finally, I developed various custom code-generation approaches to automating Fit-style tabular tests, targeting difficult-to-test legacy applications.[5]

Beyond red-green-refactor

The *red-green-refactor* mantra refers to the TDD development process:

1. Red: Develop automated tests, which are expected to fail because the system code hasn't been built yet.
2. Green: Develop just enough system code to make the tests pass.
3. Refactor: At any point when all tests are green, it's safe to clean up the code in small increments, ensuring that the tests continually pass.

Over the last decade, developers have gained significant practical experience with automated functional testing on several types of FTDD projects, using a variety of tools. FTDD goes beyond the red-green-refactor cycle,[6] which focuses on a single piece of functionality. Indeed, this approach affects an application's whole life cycle, often spanning multiple years, projects, and teams.

Figure 1 traces a single application's typical life cycle, beginning with a greenfield project developed in FTDD fashion by a dedicated project team. Once the team releases the application into production (R1), responsibility for the code and the automated tests shifts to an operations team. That team tweaks the tests when it fixes defects or develops minor system enhancements (R1.1, R1.2). When the application needs major enhancements that are too large for the operations team to tackle, management forms a new proj-

Most projects don't have the time or money to include toolsmithing in the budget.

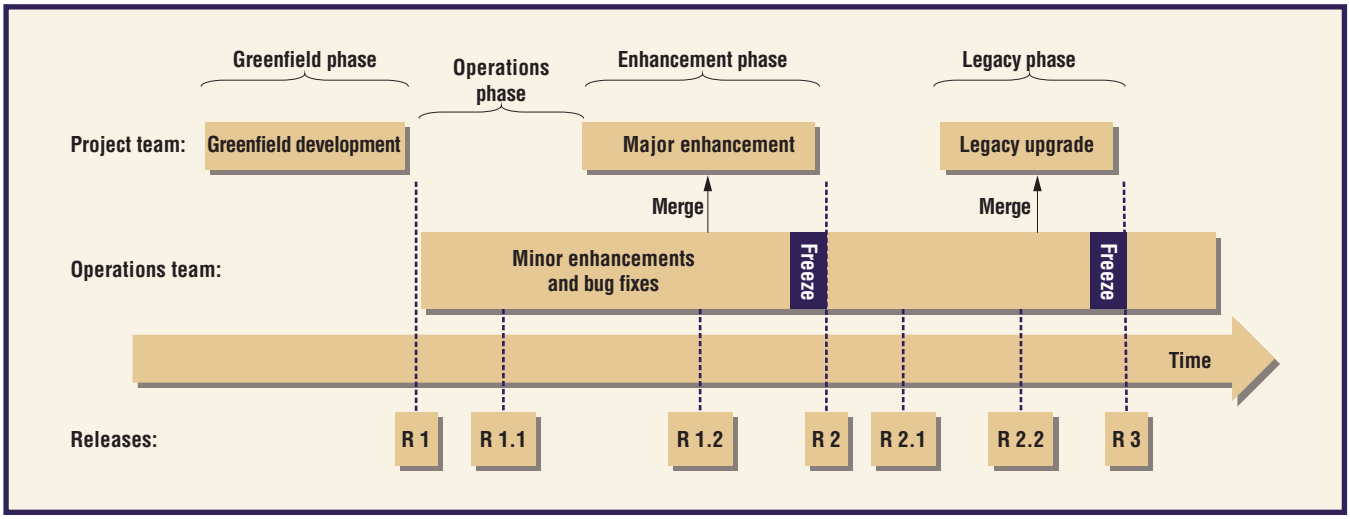

Figure 1. A typical application life cycle.

ect team. The application code branches into two parallel FTDD streams—enhancement and operations—and then merges before the production release (R2). Over time, the application might become outdated (legacy) and might need to be ported to a different technology. Again, a separate project team undertakes this major endeavor, performing test-driven porting[7] to build a new application, using the existing legacy tests as the specification.

The tests' multiple hand-offs from team to team over time are a key factor when considering tools to support automated functional testing. The following discussion about the FTDD process and the necessary tool support addresses all four parts of the life cycle:

- Greenfield phases support writing and running functional tests in a FTTD fashion.
- Operations and enhancement phases support quickly locating and understanding someone else's tests and later merging the tests.
- Legacy phases support running the same test against multiple different implementations of an application to prove equivalence.

Writing tests

Functional tests must be easier to write and maintain than production code. When functional tests become a core requirement specification artifact, we must take great care to ensure they convey business requirements clearly.

Goal: Effective requirement specification. A functional test written in a declarative style captures the essence of a business process or business rule, as opposed to an imperative style that describes how to use the system. Consider the functional test in figure 2a (we'll call it version 1, or v1), written in an imperative style. Test scripts are typically written this way, and traditional testing tools generally support this script style.

Tests that express functional requirements will look different. In the version in figure 2b (version 2), I've rewritten the same test in a declarative style. V2 is substantially shorter and has richer content than v1. I achieved this improvement by creating a domain-specific testing language[8] (DSTL) that permits the expression of business rules at a higher level of abstraction. For example, line 1 of v2 references a DSTL element that encapsulates all the details of lines 1–8 in v1.

The added benefit of writing declarative tests is that readers can more easily see the big picture and can critically assess the test content. The functional test must be correct and complete because it's the basis for the requirements specification and part of the automated regression safety net.

Version 3 (see figure 2c) is even more declarative and addresses significant deficiencies in v2. V3 removed excess detail (v2 wasn't as succinct as possible), added a precondition statement (v2 was ambiguous), and enhanced the DSTL for the final validation statement to verify that the same breed wasn't added twice (v2 wasn't sufficient). A step-by-step walkthrough of this transformation is available elsewhere.[9]

Goal: An efficient development environment. Modern IDEs such as IntelliJ and Eclipse offer such powerful and efficient features that they seem to practically write the code for you. In contrast, functional test development environ-

Version 1: Imperative

1. Start at the Maintain Dog Breeds page.
2. The page title should be "Pet Store Admin—Maintain Dog Breeds."
3. Click the Add New Dog Breed button.
4. The page title should be "Pet Store Admin—Add New Dog Breed."
5. Enter text "Poodle" into the Breed field.
6. Select "Medium" from the Size selection list.
7. Select "Curly" from the Fur selection list.
8. Click the Save button.
9. The page title should be "Pet Store Admin—Maintain Dog Breeds."
10. The message should be "New dog breed successfully added."
11. Breeds should be listed as follows:

Breed	Size	Fur	No. of dogs	No. in store	No. adopted
Collie	Medium	Straight	4	2	2
Husky	Large	Straight	1	1	0
Poodle	Medium	Curly	0	0	0
Terrier	Small	Straight	0	0	0

12. Click the Add New Dog Breed button.
13. The page title should be "Pet Store Admin—Add New Dog Breed."
14. Enter the text "Poodle" into the Breed field.
15. Select "Medium" from the Size selection list.
16. Select "Curly" from the Fur selection list.
17. Click the Save button.
18. Page title should be "Pet Store Admin—Add New Dog Breed."
19. Message should be "Error: The medium curly dog breed Poodle already exists."

(a)

Version 2: Declarative

1. Add the dog breed: Poodle, medium, curly.
2. Verify the dog breed inventory.

Breed	Size	Fur
Husky	Large	Straight
Poodle	Medium	Curly
Terrier	Small	Straight

3. Add the dog breed: Poodle, medium, curly.
4. Verify the Add Dog Breed message: "Error: The medium curly dog breed Poodle already exists."

(b)

Version 3: Declarative, succinct, unambiguous

1. Begin with kennel without: Poodle, medium, curly.
2. Add dog breed: Poodle, medium, curly.
3. Expect kennel contains: Poodle, medium, curly.
4. Add dog breed: Poodle, medium, curly.
5. Expect failure: "Error: the medium curly dog breed Poodle already exists."
6. Expect kennel unchanged.

(c)

Figure 2. A functional test called "Add Dog Breed: Reject Duplicates," written in (a) an imperative style, (b) a declarative style, and (c) a declarative, succinct, unambiguous style.

ments are either nonexistent or little more than text editors with command-line compilers. We desperately need FTDEs that support the same types of capabilities that IDEs do. Powerful, safe refactoring capabilities are even more important for an FTDE than for an IDE because, as the introduction mentioned, functional tests don't have the same regression safety net that production code has.

Especially for the legacy phase, we still

Ideally, readers can choose the format for reading any individual test that best suits their needs and preferences.

need to create tests by recording user actions as they use an existing application's user interface. Once the recording step is complete, the FTDE should treat these tests the same as a hand-coded test to enable the writer to refactor the recorded actions into DSTL elements.

Finally, the functional tests must be version controlled, ideally in the same version control tool as the production code. The functional tests' version tag should be in synch with the system code. Their format should permit detailed file comparisons to facilitate test analysis and merging when the operations and project teams work in parallel.

Reading tests

A functional test must be easier to read and locate than production code, because it will be read far more often than it will be written.

Goal: Multiple readers. There are many hand-offs throughout an application's life cycle, so different roles must be able to comprehend functional tests quickly to gain a clear, unambiguous understanding of system capabilities. For example,

- subject-matter experts read a test to validate the requirement specification,
- testers read a test to develop automated functional tests,
- project developers read a test to develop correct system code, and
- operations developers read a test to fix or enhance a specific part of the system.

Goal: Readable format. Readability must be at the forefront when we craft DSTLs and functional tests. In addition, the test format can significantly affect readability. We can express the same test in several different formats—for example, textual, tabular, storyboard, graphical, or multimodal (see www.jennittaandrea.com for an example of each format). The reader's preference and the domain complexity drive the choice of format. Ideally, readers can choose the format for reading any individual test that best suits their needs and preferences, regardless of what format the test was originally written in. For example, a reader might want to convert a tabular test into a graphical format to make visualizing complex object relationships easier, or view a declarative test at a more detailed level by expanding the DSTL elements inline.

Goal: Locatable tests. We must find a test before we can read it. Furthermore, a single functional test is much like a single puzzle piece: it's difficult to know what the big picture is just by looking at one piece. Any of the participants in the various roles might need to see how various functional tests relate to each other and to the overall business process they describe. A variety of situations expose different reasons for locating tests:

- To find a particular test because it failed a regression test run, we need search by name.
- To find all the tests in a particular functional area to prepare for an enhancement or to rerun the tests as a group, we need search by functional area. We must support navigation through the (conceptual) hierarchical linking of business process to functional test to unit test.
- To find all tests related to a crosscutting concern (such as security), we need search by metadata (for example, by keyword).

Running tests

Functional tests are executed frequently over a system's life to prove system stability when it, a related system, or part of the underlying infrastructure changes.

Goal: Multiple runners. Various project roles must be able to run functional tests in different environments:

- Testers run tests in the FTDE as they create the functional test during the "red" part of the FTDD cycle.
- Developers run tests in their IDE as they create system code during the "green" part of the FTDD cycle.
- Automated build tools run tests from the command line to automatically regression-test the system when the code changes.
- Subject-matter experts run tests from a desktop tool (such as a browser) to sign off on functional testing for the system.

Users should be able to debug a test to some degree regardless of the environment it's executed in.

Goal: Meaningful outcome. When a test is run, the user needs a clear indicator of the outcome (pass or fail). If a test fails, the user needs a

precise indication of why—what was expected and what was actually encountered.

Acceptance sign-off and auditing often require a detailed trace of system execution in the form of a sequence of screen shots or API calls. The final-outcome report might also need to be archived and might require custom formatting to conform to a corporate standard.

Goal: Adaptable touch-points. A touch-point is the part of the system that the test interfaces with. Experts generally recommend that automated functional tests run against the system API because the tests will run faster and this touch-point is much more stable than the user interface. However, sometimes testing through the UI is the best (or only) option. An effective strategy for testing in the legacy phase is to prove the new system is equivalent to the legacy system by having a single test run against both environments. The single test specification must remain unchanged when it's run against different technologies and touch-points.

Envisioning the next generation

Current FTDD tools still lack three important elements: a powerful FTDE, support for multiple test formats, and support for multiple touch-points. Each enhancement suggests various framework capabilities and references the conceptual framework diagrams. The architecture diagrams in this article are conceptual rather than concrete because there are myriad ways to implement a satisfactory solution.

A powerful FTDE

Providing a powerful FTDE would deliver the widest benefits because it would apply to all four phases (greenfield, operations, enhancements, and legacy). The benefits delivered would also be the deepest because this would substantially improve the effectiveness of writing, reading, and running tests.

Specification language. The functional testing language is expressive, letting the writer

- incorporate rich layouts, colors, and fonts in the test specification;
- create custom methods for pretest setup and post-test cleanup using the framework's specific testing language or an existing programming language (such as Java or C#);
- define and develop DSTL elements using the framework's testing language or an existing programming language;
- declaratively control and validate external system mock objects (www.mockobjects.com);
- perform detailed comparisons between test versions;
- provide a mechanism for referring to previous test steps to support complex calculations and multistep processing; and
- provide a mechanism to string together functional tests to support the succinct expression of cycles and dependencies in the business workflow.

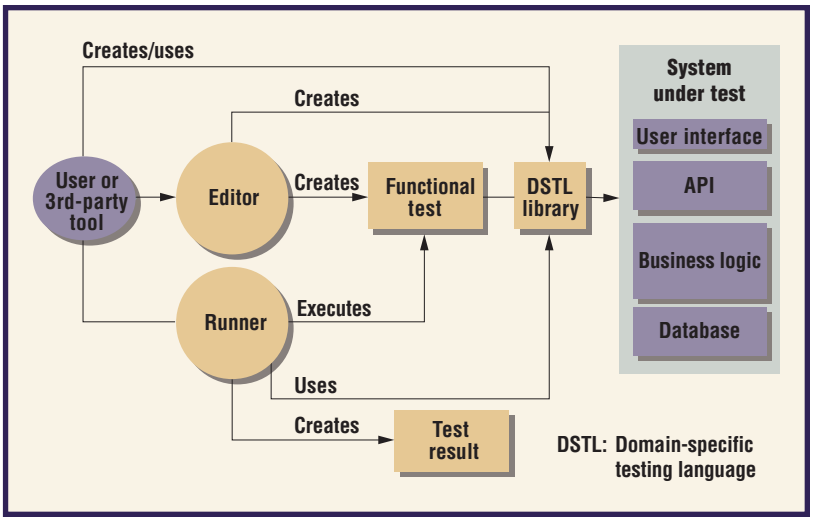

Figure 3. A basic conceptual architecture for the functional test framework.

Writing tests. The Editor (see figure 3) provides features for both the functional-test and the custom DSTL elements such as intelliSense-style (http://en.wikipedia.org/wiki/IntelliSense) code completion; incremental syntax validation; and efficient navigation between functional-test and DSTL implementations. The Editor can also search for DSTL elements and find all usages of a DSTL element. The refactoring features it supports include

- creating a DSTL element from test statements or from another DSTL element;
- expanding a DSTL element directly in the test;
- organizing and moving tests and DSTL elements;
- renaming tests and DSTL elements;
- changing a DSTL signature (adding, updating, deleting parameters, or returning values); and

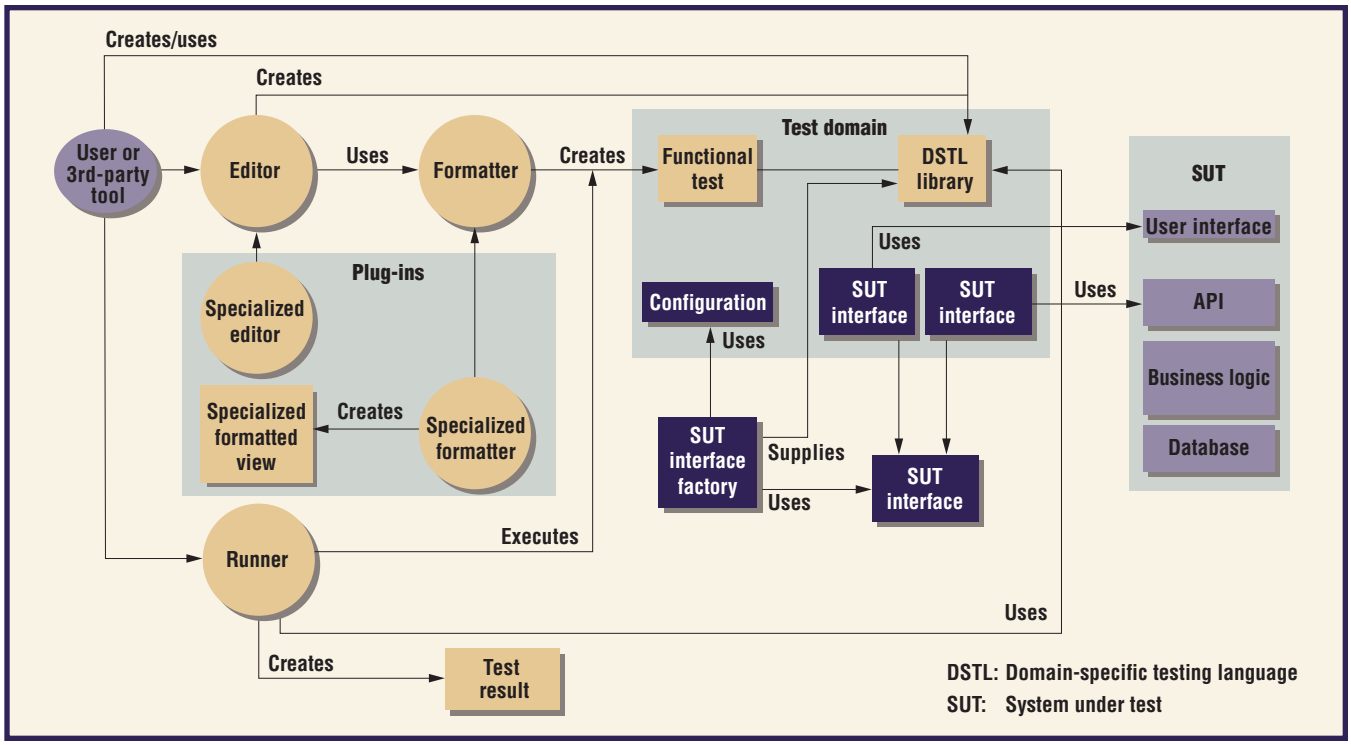

Figure 4. Multiple test formats (gold) and touch-points (blue).

- introducing a variable, constant, or parameter in the DSTL.

A plug-in developer can add a new version control system to the framework.

Users can write functional tests using third-party tools (for example, a spreadsheet or other IDE) through the Editor API.

Reading tests. The Editor provides search capabilities to locate one or more tests by name, content, or metadata. Users can view multiple tests and DSTL elements in separate windows at the same time. They can also visualize the relationships between an overarching business process description and related functional tests and navigate through them in all directions. Finally, users can read functional tests using third-party tools (such as a spreadsheet, another IDE, or a Web browser).

Running tests. Users can run tests through the Runner (see figure 3) or through third-party tools (such as an IDE or browser) using the Runner API. They can debug a test to some degree regardless of the environment it's executed in. Debugging features include setting breakpoints, stepping through a test, stepping into DSTL elements and system code (even if the IDEs aren't the same), inspecting variables, setting watch points, and viewing the call stack. Users can archive a test result and drill down from the declarative level into the detailed steps actually executed.

Multiple test formats

These capabilities are especially relevant to the operations and enhancements phase, which requires flexibility to support different stakeholders' needs. (The dark blue parts of figure 4 relate to the idea of multiple system touch-points, discussed in the next section.)

Writing tests. Users can specify tests in many different formats: textual, tabular, graphical, storyboard, multimodal, and so on. Thus, tests are independent of their format. A plug-in developer can define a new format by defining a specialized editor and formatter. Users can dynamically select the format in which to write a test and can incorporate multiple, different formats in a single test.

Reading tests. Users can view a test in multiple formats at the same time and view a test in a different format from the one it was written in. Multiple editors can be open at the same time.

Running tests. Users can view test results in various formats and define custom result for-

mats. The Runner separates the result data from the result format.

Multiple touch-points

These capabilities are especially relevant to the legacy phase, which requires flexibility to support running the same test against different environments (see the dark blue parts in figure 4).

Writing tests. Users can specify a test by scripting or recording it as someone uses an existing user interface. So, a plug-in developer can define a new Recording Specialized Editor. A recorded test can capture events at either the interface level or the API level. Once a test is recorded, it can be edited just like a hand-coded test.

Running tests. The same test can execute against various (API and user interface) touch-points and technologies at the same time. So, the DSTL Library steps are separated from the specific details for accessing the touch-point (the system-under-test interface). The SUT Interface Factory dynamically associates a particular SUT interface to the DSTL Library steps on the basis of entries in a configuration file.

A vision without a plan is merely a dream; with a plan, it's achievable. Effective action planning unfolds by working through four key questions that can guide us from an assessment of the present state to concrete suggestions for moving toward the future of functional test tools.

First, where are we? We have many partially effective functional-testing tools that have been used to create a large investment in existing functional tests. These tests have the potential to become a maintenance problem and bottleneck to progress. We also have a compelling vision for the future and many capable researchers and practitioners who can make the vision a reality.

Second, where do we want to be? We envision an active, cooperative community of researchers and practitioners working together to improve the state of the art of functional testing tools. We want FTDEs for existing tools that substantially increase FTDD effectiveness. We also want one or more functional-testing tools that fully support greenfield, operational and enhancement, and legacy phases. And we need mechanisms to make the old functional tests work with the new tools.

Third, what's in the way? Four things hinder us: time, money, imagination, and cooperation.

Finally, how do we get there? The vision I've described here is a concrete starting point based on a range of experiences. We must do three things to move forward.

We must complete the vision. A richer, more complete vision must include a broader collection of user stories and examples. We can gather these stories through a series of facilitated workshops and electronic discussion forums. The vision itself will be publicly available on a wiki moderated by the Agile Alliance (www.agilealliance.org).

We must assess current tools. It's prudent to survey key functional test tools to assess how well they support the vision. Some existing tools, such as Fit, are well on their way to supporting this vision. The assessment will involve creating user acceptance tests from framework stories and applying the tests to existing tools. The assessment results will be publicly available on a moderated Agile Alliance wiki.

We must get to work. With the vision and assessments in hand, teams can work on new functional-testing tools and meaningful enhancements to existing tools. If we apply FTDD to develop effective functional-testing tools, we'll achieve great things one test at a time. We'll achieve the vision sooner if we start now.

Acknowledgments

I thank Geoff Hardy, Brian Marick, and the *IEEE Software* reviewers for their feedback on earlier drafts of this article.

References

1. D. Nicolette, *Functional Test Driven Development*, 2005, www.davenicolette.net/articles/functional_tdd.html.
2. K. Beck, *Extreme Programming Explained: Embrace Change*, Addison-Wesley, 2000.

About the Author

Jennitta Andrea is a strategic partner and senior consultant with clearStream and has worked on more than a dozen agile projects. Her professional work, including simulation-based tutorials and in-house training, focuses on automated functional testing, agile requirements, process adaptation, and project retrospectives. She received her BS in computer science from the University of Calgary. She's a member of the board of the Agile Alliance and a member of the *IEEE Software* Advisory Board. Contact her at jennitta@agilecanada.com or www.jennittaandrea.com.

3. R. Mugridge and W. Cunningham, *Fit for Developing Software: Framework for Integrated Tests*, Prentice Hall, 2005.
4. J. Andrea, "Putting a Motor on the Canoo WebTest Acceptance Testing Framework," *Extreme Programming and Agile Processes in Software Engineering*, LNCS 3092, Springer, 2004, pp. 20–28.
5. J. Andrea, "Generative Acceptance Testing for Difficult-to-Test Software," *Extreme Programming and Agile Processes in Software Engineering*, LNCS 3092, Springer, 2004, pp. 29–37.
6. K. Beck, *Test-Driven Development by Example*, Addison-Wesley, 2003.
7. R. Bohnet et al., "Test-Driven Porting," *Proc. Agile Development Conf.* (ADC 05), IEEE CS Press, 2005, pp. 259–266.
8. E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*, Addison-Wesley, 2003.
9. J. Andrea, "Brushing Up on Functional Test Effectiveness," *Better Software*, Nov./Dec. 2005, pp. 26–31.