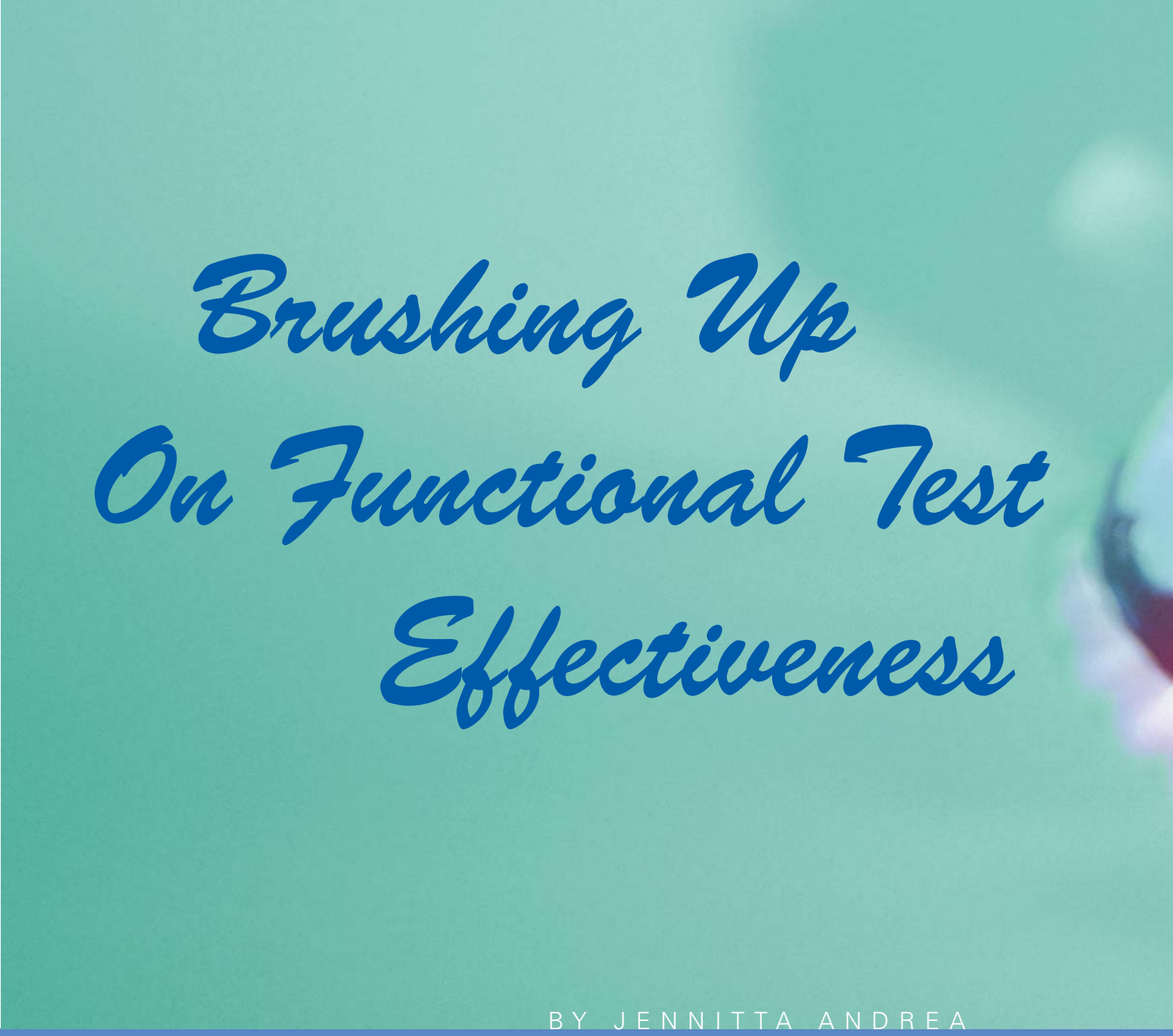
Brushing Up On Functional Test Effectiveness



BY JENNITTA ANDREA

When I was ten years old, I sat in the dentist's chair and stared at a poster of a smiling monkey. The caption was: "Do you need to floss *all* of your teeth? No, just the ones you want to keep." The monkey's smile, filled with missing, crooked, and brown teeth, clearly showed the negative consequences of not flossing. I have faithfully flossed *all* of my teeth ever since. This was an enduring lesson in risk assessment—a core skill in automated functional testing. This visit to the dentist's office and several since have had a profound influence on my approach to automated functional testing.

At my last dental appointment, I was paired up with Joan, a senior hygienist. Almost immediately she asked whether I flossed regularly. Following a brief flashback to the monkey poster, I proudly assured her that I floss every night after brushing my teeth. She said there were signs that my flossing was not effective enough. Joan explained that new research and technology have brought about great improvements in home dental care and described the floss, brush, irrigate (or FBI) approach. My nightly routine was given a complete overhaul—I began flossing before brushing, my techniques were improved, and two new tools were introduced: a specific type of floss and an irrigating pick. As devastating as it was to hear my flossing had not been completely effective, it was comforting to learn that immediate improvements were possible with very little extra time and effort.

As I reclined in the dental chair, the parallels between floss-first oral hygiene and test-first software development amused me—both are disciplines for improving quality beneath the surface. Teams new to Agile development eagerly embrace test-first development, having experienced the software equivalent of a monkey poster—war stories of projects plagued by inadequate requirements and poor quality. They are hopeful that requirements can be clearly communicated in the form of functional tests, and they are motivated to automate all of their tests in order to create a strong regression test safety net. Despite their diligence, many teams wind up with functional tests that are a bottleneck to progress, a maintenance nightmare, expensive, and an incomprehensible form of requirements. In short, their functional tests are ineffective.

The root cause of these problems can be traced to deficiencies in both techniques and tools. Let's start by exploring characteristics that enable functional tests to be effective requirements artifacts. Effective functional tests are: *declarative, succinct, autonomous, sufficient, and locatable.*

Declarative: *What, Not How*

As with any requirements specification technique, functional tests must simultaneously serve two completely different audiences: subject matter experts (SMEs) and the technical team (developers, testers, etc). SMEs must be able to read the tests and verify that functional requirements are captured correctly. Developers must be able to read the tests and find enough specific detail to drive their design and coding work.

Consider Version 1 of a functional test for a video store administration system. The test illustrates that duplicate movie titles are not permitted in the system.

An SME can easily understand this test because it is not overly technical. It is precise and detailed enough to guide a developer's work. On the surface this test appears readable; however, it is riddled with many different problems that will be highlighted in the remainder of this article. The purpose and intent of the test are lost within the forest of detail. The reader must expend significant brainpower to understand a series of tactical user interface interactions. A mental model of different screens and their relationships must be constructed, and the workflow path overflows the reader's short-term memory. Tests written in this manner must be read uninterrupted from beginning to end, and must be re-read numerous times.

Readability is improved when functional tests are declaratively expressed in the language of the business domain *(the what)* rather than in the language of a graphical user interface or the public API *(the how)*. SMEs and developers collaborate to define a domain-specific testing language (DSTL), consisting of statements that describe how to set up precondition data, perform system actions, and verify system state. Each DSTL statement has an intent-revealing name based on the vocabulary of the business domain. (See Eric Evans's book *Domain Driven Design* for more information.)

Refactoring Version 1 in this manner produces Version 2. Note, for example, lines one through eight in Version 1 have collapsed into line one: Add Movie Title in Version 2. The functional test is now completely specified by three DSTL statements (Add Movie Title, Verify Title Inventory, and Verify Add Movie Title Message). The business intent is much more obvious now that the tactical details have been moved out of the test and into the functional testing framework.

Succinct: *Less Is More*

Due to our limited short-term memory capacity—typically no more than nine items (see the StickyNotes for a link to George Miller's study),— comprehension is markedly improved when the functional test is brief and to the point. One effective approach for critiquing a functional test is to read it in reverse order (from bottom to top). Start with the validation statements, the last lines of the test, and ensure the business rules of interest are clearly expressed. The validation statements serve as the anchor for the test; every other element (statement, object, and attribute) within the test is in place to support these assertions. Continue reading upward, statement by statement, ruthlessly asking whether the test result would be the same without this element. In the end, each element should clearly contribute

Version 1: Functional Test Example

1. Start at the *Maintain Titles* page
2. Page title should be *Video Store Admin—Maintain Movie Titles*
3. Click the **Add New Title** button
4. Page title should be: *Video Store Admin—Add New Title*
5. Enter text *Star Trek* into the field labeled **Title**
6. Select *Science Fiction* from **Category** selection list
7. Select *DVD* from **Media Type** selection list
8. Click the **Save** button
9. Page title should be *Video Store Admin—Maintain Movie Titles*
10. Message should be **New title successfully added**
11. Titles should be listed as:

Title	Category	Media Type	# Copies	# In Store	# Rented
Aladdin	Children	Video	4	2	2
Star Trek	Sci Fi	DVD	1	1	0
Star Wars	Sci Fi	DVD	0	0	0
Toy Story	Children	Video	0	0	0

12. Click the **Add New Title** button
13. Page title should be: *Video Store Admin—Add New Title*
14. Enter text *Star Trek* into the field labeled **Title**
15. Select *Science Fiction* from **Category** selection list
16. Select *DVD* from **Media Type** selection list
17. Click the **Save** button
18. Page title should be *Video Store Admin—Add New Title*
19. Message should be **Error: The science fiction DVD movie title Star Trek already exists.**

Version 2: More Declarative

1. Add Movie Title (Star Trek, Sci Fi, DVD)
2. Verify Title Inventory

Title	Category	Media Type	# Copies	# In Store	# Rented
Aladdin	Children	Video	4	2	2
Star Trek	Sci Fi	DVD	1	1	0
Star Wars	Sci Fi	DVD	0	0	0
Toy Story	Children	Video	0	0	0

3. Add Movie Title (Star Trek, Sci Fi, DVD)
4. Verify Add Movie Title Message **(Error: The science fiction DVD movie title Star Trek already exists.)**

Version 3: More Succinct

1. Add Movie Title (Star Trek, Sci Fi, DVD)
2. Verify Title Inventory

Title	Category	Media Type
Aladdin	Children	Video
Star Trek	Sci Fi	DVD
Star Wars	Sci Fi	DVD

3. Add Movie Title (Star Trek, Sci Fi, DVD)
4. Verify Add Movie Title Message **(Error: The science fiction DVD movie title Star Trek already exists.)**
5. Verify Title Inventory

Title	Category	Media Type
Aladdin	Children	Video
Star Trek	Sci Fi	DVD
Star Wars	Sci Fi	DVD

to the core concept being tested. Elements that do not contribute to the core concept may be handled in one of two ways:

1. If the element is necessary but tactical, move it to the DSTL within the underlying testing framework.
2. If the element adds nothing to the test, remove it.

You'll know that you've trimmed away too much detail if the test becomes ambiguous. Functional tests that are initially written in reverse order—starting with validation statements and built upward statement by statement—will be remarkably succinct and focused.

Returning to Version 2 of our example, recall that the core business rule is that duplicate movie titles are not permitted. Assessing the validation statement (Step 4), we find it does not completely express the business rule because it only verifies that an error message is displayed. The test is enhanced to explicitly verify that the movie was not added twice. Now that we have strong anchoring validation statements, we examine the remainder of the test and ask whether Step 2 contains excess attributes. The SME explains that, while all of the columns listed are displayed on the GUI, only the first three (Title, Category, and Media Type) relate to detecting duplicate movie titles. The last three columns (# Copies, # In Store, and # Rented) are removed from the test. However, they are not removed from the GUI. We continue to examine Step 2 and ask whether it contains excess movie title objects. Since we are testing the behavior of adding *Star Trek* twice, why are *Aladdin*, *Star Wars*, and *Toy Story* included? The SME explains that when a movie title is successfully added, it must be displayed in alphabetical order on the screen. The inclusion of the movie titles *Aladdin* and *Star Wars* showcases this aspect of the business rule. *Toy Story* is superfluous and can be removed from the test. Our succinct functional test is displayed in Version 3.

Autonomous: *All in One*

Now that the meaning of the functional test is clear and the unnecessary elements have been eliminated, we can turn our attention to the issue of autonomy. One measure of an autonomous functional test is that it is completely self-contained, enabling different readers to understand the test the same way. Another measure is that it behaves in the same way whether executed alone or as part of a suite. Autonomy is achieved when the test is free of objects that materialize out of thin air and instead clearly lists the essential precondition state. Missing preconditions introduce considerable ambiguity because the reader must speculate about the origin of the mystery object. Is it a part of global, shared test data? If so, what else in the global shared data affects this test? Was it left over from another test? If so, which one? Is the object unique to this test?

To discover missing preconditions

Version 4: More Autonomous

1. Precondition
   a. Create Movie Title (Star Wars, Sci Fi, DVD)
   b. Create Movie Title (Aladdin, Children, Video)
2. Add Movie Title (Star Trek, Sci Fi, DVD)
3. Verify Title Inventory

Title	Category	Media Type
Aladdin	Children	Video
Star Trek	Sci Fi	DVD
Star Wars	Sci Fi	DVD

4. Add Movie Title (Star Trek, Sci Fi, DVD)
5. Verify Add Movie Title Message **(Error: The science fiction DVD movie title Star Trek already exists.)**
6. Verify Title Inventory

Title	Category	Media Type
Aladdin	Children	Video
Star Trek	Sci Fi	DVD
Star Wars	Sci Fi	DVD

Version 5: More Sufficient

1. Precondition
   a. Create Movie Title (Star Wars, Sci Fi, DVD)
   b. Create Movie Title (Aladdin, Children, Video)
2. Add Movie Title (Star Trek, Documentary, DVD)
3. Add Movie Title (Star Trek, Sci Fi, Video)
4. Add Movie Title (Star Trek, Sci Fi, DVD)
5. Verify Title Inventory

Title	Category	Media Type
Aladdin	Children	Video
Star Trek	Documentary	DVD
Star Trek	Sci Fi	DVD
Star Trek	Sci Fi	Video
Star Wars	Sci Fi	DVD

6. Add Movie Title (Star Trek, Sci Fi, DVD)
7. Verify Add Movie Title Message **(Error: The science fiction DVD movie title Star Trek already exists.)**
8. Verify Title Inventory

Title	Category	Media Type
Aladdin	Children	Video
Star Trek	Documentary	DVD
Star Trek	Sci Fi	DVD
Star Trek	Sci Fi	Video
Star Wars	Sci Fi	DVD

in our example, perform another reverse scan of the test, this time focusing on tracing the origin of each object. As we examine Version 3, it is evident that two of the movie titles mentioned in Step 2, *Star Wars* and *Aladdin*, have no known origin. The remedy is to introduce a precondition section, where these two objects are explicitly created. (See Version 4.)

Note that the new precondition DSTL statement is Create Movie Title and the workflow DSTL statement is Add Movie Title. For automation efficiency, precondition statements often are implemented differently than workflow statements. For example, preconditions may bypass the outer layers of the application or may be permitted to break business rules to facilitate setting up error conditions.

Sufficient: *How Much Is Enough*

After reading Version 4 of the functional test, the developer asks, "What should happen if the user tries to add *Star Trek* as a video?" The SME responds, "It should be added successfully because it has a different media type. A movie title is uniquely identified by its title, category, and media type." This exchange is typical in an Agile approach to requirements. Rather than aiming to have a completely defined requirements specification, the functional tests represent key examples (See the StickyNotes for more on examples.) The intent is to have an active reader engage in dialog with an SME until the big picture becomes clear. "Great," the developer responds. "Should we capture this detail in a functional test or a unit test?"

The automated regression test suite, composed of both functional and unit tests, forms the safety net that enables the team to fearlessly embrace change. (See Kent Beck's book *Extreme Programming Explained* for more on embracing change.) When deciding how many functional tests are enough for a project, it is helpful to reflect on the physical qualities of a safety net. The strength and structure of a safety net come from the segments that are tightly woven with heavy-duty material. The safety net would be too stiff and too expensive if it were made entirely of this material, so the majority of the net is built out of flexible material, loosely woven to absorb the force of a falling acrobat. The effectiveness of the safety net is a result of the correct distribution and placement of these different materials.

Functional tests are like the tightly woven segments of a safety net. They act as the skeleton of the automatic regression test suite by defining the requirements for the important workflow scenarios and key business rules. The strength of the functional tests comes from the fact that they are visible to the SME. The flexibility of the safety net comes from the unit tests, which fill in the granular detail for individual workflow steps and business rules.

Reflecting again on the developer/SME dialog above, it is clear that fundamental information about the business rule is absent from the test: A movie title is uniquely identified by the combination of title, category, and media. We are not just dealing with additional low-level detail or a variation on a theme, so the answer to the developers' question is that the functional test should be enhanced with additional steps as shown in Version 5. These are created as workflow steps, rather than precondition steps, because they are part of the business rule being tested—not merely part of the scenery.

Locatable: *The Name Game*

Each individual functional test is just one piece of a puzzle. In order for functional tests to effectively serve as a requirements specification, the reader must be able to connect the pieces together into a complete picture. An efficient approach to puzzle building is to first group related pieces together (e.g., edge pieces, those with the same colors, etc.), build up each section, and then join the sections together. Keeping this strategy in mind, it is essential that all of the tests related to a particular system feature or business rule are easily identified and grouped together.

In short, functional tests must be well named, organized, and searchable. A well-named functional test briefly summarizes the key feature and business rule being addressed. Our example test should be named something like: Add Movie Title: Reject Duplicate. An effective static organization integrates the functional tests within contextual information, such as history, business model, vision, charter, use cases, etc. A wiki is an ideal tool for facilitating fluid navigation between functional tests and other lightweight documents. Tagging each test with relevant search keywords enables the tests to be dynamically grouped in terms of their crosscutting concerns.

Automation: *Beyond "Flavor of the Month"*

Time and again teams select a tool before they develop a strategy and guiding principles for functional testing. It should now be clear that functional test effectiveness is primarily achieved through decisions about the vocabulary and detail contained in the tests. As previously stated, effective functional tests are: declarative, succinct, autonomous, sufficient, and locatable. While necessary, tools represent a small part of the effectiveness equation.

Selecting the right tool is critical. Prior to my dental appointment with Joan, I selected floss because of its flavor. Joan recommended a specific type of floss because the shape, width, and wax level was most appropriate for the characteristics of my teeth. Software teams also must be deliberate in selecting a functional testing tool based on the specific characteristics of their project. Judge the available tools by their level of support for techniques which enable effectiveness, such as: defining a DSTL, managing precondition data, organizing tests within context, and searching for tests based on user- defined attributes. It is essential that the tool provides detailed results of test execution and facilitates a clean-slate approach to testing by having the tests clean up after themselves. The good news is that many of the automated functional testing tools available today score quite well

on this checklist.

Significant efficiency gains can be realized by improving tool effectiveness. Encourage vendors to provide more powerful functional test development environments that support things like refactoring across a test suite, integration and execution of multi-modal elements within a single test (tables, text, shapes, color, etc.), incremental syntax validation, keyboard navigation into the DSTL statements, code completion for DSTL signatures, etc.

Repairing a Broken Safety Net

The systematic approach for improving a single test described here can be expanded if a number of functional tests within a suite are ineffective. If the tests are not readable, the first step is to refactor them into a DSTL. Next, review the tests, looking for the root causes of the problems, such as: excess detail, ambiguity, incompleteness, duplication, etc. Then, develop an incremental plan for refining the tests based on an assessment of the impact of the problem, the distribution of the problem, the relative priority of the tests that are impacted, and the cost to fix the problem. Finally, refactor the tests a small step at a time to ensure that the safety net has high availability. Occasionally a refactoring may cause the tests to become incompatible with the capabilities of your functional testing tool, requiring the tool to be enhanced or replaced (See the Sticky Notes for references). I encourage you to persevere, strive for effectiveness, and don't forget to floss first. **{end}**

Jennitta Andrea is a coach, developer, tester, analyst, and instructor with clearStream. She is regularly published, speaks at conferences, and is a board member of the Agile Alliance.

Sticky Notes

For more on the following topics, go to www.StickyMinds.com/bettersoftware

- References
- George Miller's study of memory capacity
- More on examples

Functional testing is a quality assurance (QA) process and a type of black-box testing that bases its test cases on the specifications of the software component under test. Functions are tested by feeding them input and examining the output, and internal program structure is rarely considered (unlike white-box testing). Functional testing is conducted to evaluate the compliance of a system or component with specified functional requirements. Functional testing usually describes what the system does.