# Functional Testing: A Pattern to Follow and the Smells to Avoid

Amr Elssamadisy
Gemba Systems

Amherst, MA
++1-435-207-1225

amr@elssamadisy.com

Jean Whitmore

1860 Sherman
Evanston, IL
++1-312-782-7156

jeanimal@gmail.com

## ABSTRACT

Functional tests are automated, business process tests co-owned by customers and developers. They are particularly useful for rescuing projects from high bug counts, delayed releases, and dissatisfied customers. Functional tests help projects by elucidating requirements, making project progress visible, and preventing bugs. We present functional testing in pattern format because it is especially expressive in conveying expert advice and enables the reader to make an informed decision regarding the applicability of the solution. The pattern presented aggregates multiple experiences with functional testing over several agile development projects. However, we have seen functional testing become more costly than its benefits, so we describe the symptoms—"smells"—of potentially costly problems. These are not problems with functional testing per se, but with the misinterpretation and mis-implementation of this practice. We draw on our experience to suggest ways of addressing these smells. Done right, functional testing successfully increases software's quality and business value.

## Categories and Subject Descriptors

D.2.1 [Requirements/Specifications] D.2.5 [Testing and Debugging]

## General Terms

Testing, Patterns, Agile Development Practices

## Keywords

Functional Testing, Acceptance Testing, Patterns, Agile Development Practices

## 1. COST-EFFECTIVE FUNCTIONAL TESTING

Many of us are familiar with projects that began as a joy to work on, but as they grew, increasingly suffered from high bug counts, delayed releases, and dissatisfied customers. Functional testing—the practice of customers and developers co-writing business process tests that execute automatically—can help solve these problems. Functional tests speed releases by preventing bugs and shortening the testing cycle. They can automatically determine whether an application is doing what customers expect. They can also help customers communicate requirements in a precise, consistent way to developers as a form of "executable requirements." A project with functional tests can continue to be a source of pride and joy even as it grows in size and complexity.

So why has functional testing not been embraced as commonly as unit testing in the agile community? Why do many of our colleagues complain that the costs of functional testing exceed the benefits? We believe that people who have given up on functional testing have lacked the right tools and techniques. After all, how many unit tests would you write without xUnit and a continuous build? How much refactoring would you do if you were coding in a text editor? The right tools and techniques can make functional testing easy and cheap.

In particular, we recommend techniques for making functional testing fast enough to be in the continuous build (and at least as fast as the typical check-in cycle in a non-agile development environment). We also explore techniques that make diagnosis of test failures relatively easy.

However, sometimes even the right tools aren't enough. If setting up a functional test is onerous, the root problem may be the architecture of the system under test. This phenomenon is similar to the idea that if setting up an object in a unit test harness is especially hard, then the object probably has too many dependencies. We will suggest architectural changes such as improved modularization of subsystems and moving business logic out of the Graphical User Interface (GUI) and into a service layer [1]. These changes make functional testing easier while making the architecture better.

In this paper, we assume functional testing is done within an *agile development* [2] environment, although we offer a few variations for a traditional development environment. Our focus is also on functional tests that exercise all layers except the GUI, but most of our patterns and smells apply to other types of functional test. We will point out when they do not.

We begin by describing functional testing in a pattern format so that readers can determine whether the practice is appropriate for their projects. Then we identify functional testing smells—

A preliminary version of this paper was workshopped at Pattern Languages of Programming (PLoP) '06 October 21-23, 2006, Portland, OR, USA. 

signs of costly problems—and the technical and architectural solutions that address them. We hope people will recognize the need for better techniques rather than giving up on functional testing. The benefits are just too good to pass up.

## 2. Functional Testing: An Agile Practice Pattern

Patterns allow us to propose development practices as potential solutions to a common set of problems. By describing functional testing in a pattern format, we empower readers to make their own evaluation of this development practice. Using functional testing is then not a stark black or white decision; it depends on how much a development team has experienced the problems and whether this pattern as a proposed solution is within reasonable costs. We include several stories and narratives to bring home the points based on specific experiences we have had.

### Automated, Business Process Tests

In this paper, we define *functional tests* as

- business process tests that are
- co-owned by customers and developers and that
- can be automatically executed.

Functional tests can be better understood by comparing them with what they are not.

First, functional tests are not owned by a testing department (which may or may not be part of a Quality Assurance department). Instead, they are owned by—i.e. created and maintained by—customers. In order for customers to be owners, the functional testing tool must provide a way for customers to read, write, and execute test specifications, although developers may implement tests and the testing department may help develop more effective tests.

Second, functional tests are not manually run. No one needs to click on screens or set up data in order to execute them. Instead, functional tests, like unit tests, are completely automated.

However, unlike unit tests, functional tests are not focused on isolated units of code, whose proper behavior a developer defines. Instead, they exercise a useful business process, whose correct outcome is defined by a customer. We speak of a business *process* because we mean more than just the static business rules; we mean also the sequence of steps that invoke the business rules to generate a useful outcome. If use cases are used, then each scenario of a use case can be covered by a functional test. Our goal is to assure that the program does something useful for a real user.

If functional tests cover more than a unit, just how much should they cover? There are several options, depending on the type of testing you want to do. Our experience is primarily with functional tests that are driven from the *service layer* (or control layer or system-façade layer), a layer between the GUI and domain layers on n-tier systems. That is, our functional tests exercise all layers except the GUI so that they are almost end-to-end. We will call these service-driven functional tests.

Many functional testing tools drive tests through the GUI. Some of the patterns we describe also apply to these GUI-driven tools. However, the tests of GUI-driven tools are often more fragile than those of service-driven tools because they may break when a button is moved. More importantly, GUI-driven tools do not have the architectural benefits of service-driven tools. For example, they do not help drive business logic out of the GUI [3].

### Forces

The *forces* in a pattern are the driving factors that lead to the implementation of the pattern. Patterns can be considered as problem/solution pairs. The forces are the problems that are addressed by the pattern as a potential solution. The main forces pushing us to try functional testing are too many bugs, delayed releases, and poorly captured requirements.

#### *Bugs Increase As Inter-Module Dependencies Grow*

Most development groups that we have seen try functional testing were motivated primarily by a desire to reduce bugs. That is, when they hear the phrase “functional testing,” they particularly focus on the word “testing.” Unit tests can keep individual classes fairly free of bugs, but they do not address inter-module bugs. Furthermore, as the code base grows, the number of potential inter-module bugs grows faster.

#### *Delayed Releases*

As the application grows and the product matures, the testing department cycle can take longer, causing increasingly delayed releases.

#### *Slow Manual Testing*

Manual testing by a testing department will take significantly longer with a large product than a small one. Because manual testing is slow, the feedback about a bug occurs long after the code changes that caused the problem were made. The delayed feedback makes it hard to diagnose which change caused the bug, so fixing a bug found by the testing department takes longer, too.

##### *Slow Patches*

A corollary of slow releases is slow patches for bugs reported in the field. In many development environments, developers have to set up a full database and perform many manual steps to reproduce a bug. And they must reproduce the bug both to diagnose it and to confirm they have eliminated it. How much nicer if they had an easy way to script the system with the minimal conditions to reproduce the bug!

#### *Not Knowing When a Task is Done*

Almost everyone has experienced a project that was declared “done” and then continued for weeks or months afterward. With functional testing, the customer writes tests that exercise the business

process (represented in a use case, story or feature) scheduled for the current iteration. When the functional tests pass, the work is done.

### *Poorly Captured Requirements*

#### *2...1 Imprecise Requirements*

One of the reasons projects drag on after they are declared "done" is that the original requirements were imprecise. Verbal requirements do not provide enough detail for coding. Developers guess what the customer meant and call the project done. But if the developers guessed wrong, the code will have to be re-worked.

#### *2...2 Contradictory Requirements*

Many "done" projects get stuck in the testing phase because of bug cycles. An example of a simple cycle is that when bug A is fixed, bug B appears; and when bug B is fixed, bug A re-appears. But the cycle is rarely that obvious, especially if A and B are in different parts of the system or take a long list of manual steps to reproduce.

An automated test suite could quickly show that both bugs are never fixed at the same time. At that point, one might discover that A and B *cannot* both be true at the same time because they are contradictory. Functional tests help "test" our requirements for contradictions.

### *Outdated Requirements*

Finally, requirements are also often outdated. The longer running the project, the more likely that at least some of the requirements have fallen behind the code. Let us be frank—have any of us really had requirements that were 100% up-to-date after a year of development?

Outdated requirements can be more nefarious than no requirements. If there are no requirements, developers will try to extract them from the customer, the code, or the unit tests, all of which are likely to provide fairly up-to-date information. But outdated requirements are *mis*information. They can waste a lot of time by sending developers down the wrong track.

## Description

Functional testing is much more than automated acceptance tests; the set of tests can be considered "executable requirements." That is, they are requirements written by the customer (sometimes with the help of a developer depending on tool support) that can be run and either passed or failed.

Unit testing is often practiced with test-driven development. The developer writes a test for a case the code cannot yet handle. Because the case has not been implemented yet, the test fails, resulting in a red bar in the unit test GUI. Then the code to pass the test is written, which turns the bar green. Then the cycle is repeated in a red-green-red rhythm.

Functional tests take the red-green-red loop of unit testing to the level of red-green-red loops for adding new business functionality to the application. From that point of view, functional tests allow the developer to know when she is done with the task at hand as indicated by the customer. This reduces a large amount of effort where code is submitted to the customer or testing group only to be found lacking in functionality and be brought back into the development group.

A major—often uncited—contribution of functional testing is the improvement of the architecture of the system under test. Functional tests force business logic to be removed from the GUI and moved into the service layer, where the functional tests can exercise it. Functional tests also encourage modularity and the separation of subsystems, analogous to how unit tests force loose coupling between objects for testability. This idea is still new to us but we have found that it rings true with others with similar experiences.

Another major contribution of functional testing is that it tests an entire set of possible errors that is not addressed by unit testing. As any experienced object-oriented programmer knows, a significant part of the complexity of an object-oriented system is in the relationships between the objects. Functional testing exercises these complexities as unit testing cannot (and is not intended to). Software quality increases. And development can proceed at an even faster pace than unit testing enabled.

A fourth contribution of functional testing shows up more in the later stages of a project as it enters maintenance mode; bugs reported either by the testing team or the customer come in to the developer as a set of steps for reproduction. The immediate response for a developer when functional testing is available is to write a failing functional test to reproduce the steps. Then she digs in, finds the problem, writes a failing unit test, and fixes the problem, causing both the unit and functional tests to go green (most of the time). This technique, which is enabled by functional tests, catches the "false fixes" where the developer finds the bug, writes the unit test, and assumes the bug has been fixed when it truly is not.

Note that for all of these benefits, the functional test suite must be part of the continuous integration build. If functional tests are not in the build, they can easily become a liability instead of a benefit, a situation we describe in the smells below.

## Variations

### *Covering the Domain Only*

This paper focuses on functional tests that execute logic from the service layer through the domain layer all the way down to persistence. Not all functional tests must exercise all these layers; in fact Mugridge and Cunningham [3] argue for writing functional tests to exercise the domain logic only. Such tests are still useful, but they do not cover the subsystem boundaries, which are bug-prone. The domain-only approach is a viable alternative if running end-to-end tests within a developer check-in cycle is infeasible.

### *Functional Tests Written By Committee*

We argue that customers or analysts should write functional tests because they are in the best position to write requirements. However, testers and developers can join customers and analysts to co-write tests.

Testers bring their expertise in test-case development and help write requirements that cover the necessary details. Developers may be needed to help make the requirements executable depending on the tool. For example, the Framework for Integrated Tests (FIT) tool [4] requires developers to write fixtures before tests can execute. We have found that writing tests by committee usually happens primarily in the beginning stages of adoption of functional testing as analysts learn to think like a tester, and developers build their domain language. In later stages, writing tests by committee tapers off and the brunt of test authoring falls to the analysts with occasional help from others in the development group.

### *Functional Tests Written With Unit Testing Tool*

Some teams write their functional tests with a unit testing tool such as NUnit or JUnit. Using an xUnit testing tool covers code adequately but loses involvement from customers and analysts, since the tests are now coded in a language that they can neither write nor read. It becomes the developer's job to translate the requirements into these tests. The status of the tests as passing or failing is also not visible to either the customer or testing group.

We consider functional tests in xUnit to be rather hobbled because of the exclusive focus on coverage. These tests are indeed better than no functional tests but could be considered a smell.

### *Functional Tests Within a Traditional Development Environment*

Our experience with functional testing is within an agile development environment, but there is no reason it cannot be used on non-agile projects. The key point is that the functional tests must be run at a frequency that matches the developer check-in cycle. That way, the source of failing tests can be identified. All of the benefits of agile functional testing are achieved, just at a slower cycle time because there is no continuous integration build. When done in this environment, the emphasis on speed of running tests is reduced because the check-in cycles are typically much longer.

## Benefits

Whereas forces push us toward a pattern, benefits pull us. Forces describe a problem that the pattern will solve. In contrast, we obtain the benefits even if we do not currently have any problems.

### *Development Team Has More Confidence*

There is a definite sense of confidence that developers acquire when there is a solid test framework that they rely upon. Unit testing and TDD have gone a long way in making developers more confident of their code. This is not merely a "warm-fuzzy" feeling (which is always good for morale), but enables faster development because developers change what needs to be changed via refactoring. Functional tests take this confidence up a notch or two above and beyond unit testing. They also improve the confidence of the customers/analysts and testers because they have a direct relationship to the requirements and regression tests. They know a green test is a non-ambiguous indication that the related scenario is *working*.

### *Robust Tests*

Service-driven functional tests skip the GUI and focus on business logic. Business logic tends to be fairly stable, and so the tests don't have to change much. In contrast, automated tests that hit GUI elements break when GUI elements are re-arranged.

### *Errors and Bugs are Reproducible Quickly*

Once a bug is found, a functional test is written, and that bug doesn't come back to haunt us. A unit test should also be written around the buggy code, of course, but when developers first begin investigating a bug, they don't know where to write the unit test because they don't know which unit caused the problem. But they (hopefully!) know which use case caused the problem, so they should be able to write a functional test immediately. By writing tests as soon as bugs are discovered, we eliminate the bug-fix-break thrashing that happens when systems become brittle.

We have found that when a system moves from initial development to production that the amount of time spent developing new functionality decreases. With a functional testing framework at hand the "business language" has already been built and it becomes very straight-forward (more than for unit testing) to build a functional test that exactly reproduces the error based on the bug report. This allows the developer to have an executable reproduction of the bug that can be used for digging into the code repeatedly without having to keep setting up the environment "just so".

### *Testers Have Time to Be More Pro-Active*

If "Slow Manual Testing" is a reason to try functional testing, then quick automated testing is a benefit. The consequence is that testers are relieved of much of the day-to-day burden of manual testing of the main business rules. Instead, testers have more time to be pro-active, collaboratively helping developers design more testable code, rather than waiting to "clean up" at the end of an iteration.

### *When a Task Is "Done" is Visible for All*

Recall that without functional testing, we are driven by the force of "Not knowing when a task is done." Using functional testing does help us know when a task is done, but it's more than just that. Functional testing makes progress visible to the entire development team—customer, analyst, developer, tester, and manager. At any point in time all passing (and failing) tests can be viewed. With a little effort business value produced at a functional level can be analyzed for management needs.

### *Better Design, Better Architecture*

Functional testing drives better layer and subsystem separation. Consider the layers of a multi-tier architecture. Since the functional tests execute through the service layer, every bit of business logic that has found its way into the presentation layer must either be duplicated in the test fixture or pulled into the service layer. We explore this point in more detail in section 4.1.

Similarly, consider the subsystems of the system—the modules with functional responsibility, such as a module for tax calculations. As we show in section 4.2, any tax logic that has leaked out of the tax module will be duplicated in the test fixture unless it is moved into the tax module. Functional tests help solidify the responsibilities of a subsystem.

### *Analysts Think Through Requirements in Greater Detail*

Analysts think through requirements in greater detail to achieve the descriptions needed to write a test. For example, an analyst might state that textboxes should be disabled whenever they are not needed. But when he writes a functional test for this requirement, he is forced to get explicit about which conditions cause which textboxes—or really their representations in the underlying service layer—to be disabled.

### *Improved Customer-Developer Communication*

The concrete examples codified in the functional tests are not sufficient to specify requirements. Customers would not know how to create such detail by themselves, anyway. Instead, it is the collaboration between customers and developers that helps flesh out requirements for both of them.

On the whole, functional testing with requirements specification can improve communication between developers and customers. Over time, the discussions of the functional tests help the team develop a common vocabulary and a common vision for the system [4]. Examples of the development of such collaboration can be found in Mugridge and Cunningham's recent book [5].

## When to Use It

There are several tool requirements when it comes to functional testing. Only use functional testing if you are able to make it part of your build process. On agile development projects this means that it must be part of the continuous integration build. On more traditional projects, the functional test suite must be run within the granularity of a typical check-in cycle.

If you cannot run your test suite within the normal check-in cycle time, you may find that your tests are noisy and often failing because they cannot keep up with the current build (more detail in section 0). For functional tests realistically to be part of the build, the functional test suite should not take more than 20 minutes to run (as a rule of thumb for agile projects). To achieve this, the following strategies have been found helpful:

- Database where test set is present and refreshable/loadable within an acceptable time. That means we have to actively keep a snapshot to support our suite of tests.
- Tests can use transactions and rollback at the end of the test instead of committing (usually 5-10 times faster than a committed transaction).
- Distribute functional tests on separate machines every time one machine's run takes too long.

Finally, you are ready to introduce functional tests if you have the attitude that testing is a primary development practice and not a secondary practice that can be dropped in a crunch or if it requires a large effort. Functional testing does not come free, and we will see below in section 3 that the cost of cutting corners is very expensive.

## How to Use Functional Testing

Functional testing is much more than just testing. It is also about communication between developers, analysts, and testers. It is about understanding the requirements, the business domain, and your system as a solution addressing business problems.

Jim Shore states, "In the same way that test-driven development, when done well, facilitates thinking about design, [functional testing] done well facilitates thinking about the domain. This thinking happens at the requirements level and at the design level" [6]. Ultimately functional tests become a domain-level language spoken among the various members of the development team. So as you embark on functional tests, be sure to focus on communication of requirements and building up of the domain language. In fact, *Functional Tests Written By Committee* in section 0 is an excellent way to start off.

We would add that service-driven functional testing also facilitate thinking about system architecture. You simply can't put much logic in your GUI if you have to run your functional tests without the GUI!

Functional testing is also very tool sensitive. If the tools are not up-to-par in speed and feedback then functional tests lose much of their benefit. Once you have the right tools, you need to know how to use them. Functional tests should iteratively cover use cases, one thin scenario slice at a time.

1. Choose one specific example of a path through a business process—e.g. one scenario through a use

case—to test at a time. Keep the scenario "slice" thin and deep. That is, test a small set of functions at a time and run it from the service layer all the way to the database. We would recommend selecting a high-risk slice first, e.g. replicating a recent bug, so that team members care about the outcome.

2. Minimize the amount of data in your database snapshot used for your testing. Remember, the smaller the database, the faster the refresh and the actions that are performed in the database.

3. Mock out external systems whenever possible for speed and independence. A good example would be mocking out an external credit card authorization service for an e-commerce application.

4. However, you may want to include a few tests that interface "high risk" external systems that could cause (or already have caused) your system to fail if you misunderstand their API. The tests can then help document the API.

5. Whenever a functional test strip gets too "thick"—e.g. if it includes more than one scenario—separate it into different tests.

### When Not To Use It—Are you ready for Functional Testing?

The long and short of it is this: don't use functional tests if you are not willing to put the effort to write the tests. This may turn out to be a non-trivial effort—there are definite costs. So if you are not willing to do *all* of the following, then maybe functional testing is not appropriate at this time:

- Introduce a technique to determine what coding modifications have broken a build. We recommend that you make functional testing part of the continuous build, but if not then at least have a functional testing cop. This is discussed in detail in section 0.
- Modify your existing system for testing. Most systems built without functional testing in mind will need modifications. Many of these modifications are not simple and may involve architectural changes. Section 5 discusses *architectural smells* that will require these types of changes to enable useful functional testing.

### Suggested Adoption Strategy

Like almost everything in agile development, functional testing should be adopted iteratively. Be careful that you keep "people" ahead of "process." That is, iterate to get developers and customers trained and have them build a few functional tests. Then, after the team has a few working functional tests that are part of the build, ask them for feedback on the tools and processes. Improve your tools and processes until the developers and customers are happy with functional testing. Then iteratively expand the practice to the team.

When functional tests are not part of the build, they can cause much more harm than good and may not catch on or ever be useful. We have seen this happen and it is not a pretty sight.

Adding functional tests to a legacy system—i.e. one that does not already have functional tests—can be challenging because the architecture might not allow excluding the GUI or testing a single use case scenario at a time. You also may have re-architect some of your system to speed up the functional tests enough to be part of the continuous build. Functional tests can initially be added for new features or to reproduce bugs, with supporting unit tests added for the implementing code. As we describe below, we do not recommend adding functional tests without unit tests.

During the transition to functional tests, it can help to assign a developer the role of "Functional Test Cop." The cop's job is to track down the developers who break the functional tests, help them see why their code broke the test and help them fix the problem. See the narrative in section 0 for more detail on this role.

## 3. EXAMPLE OF FUNCTIONAL TESTING WITH FIT

Those new to functional testing might feel frustrated at the general nature of the pattern description. How exactly does it work? For the purpose of helping you decide whether functional testing is right for your team, we provide a brief taste of doing functional testing with a tool we like, Framework for Integrated Tests, or FIT for short. Our example is extremely simplified. Mugridge and Cunninham [5] describe how to write more complicated functional tests with FIT. But we hope this simple example will illustrate our claims about functional testing's benefits.

### A Simple, Fully Explained Example

FIT tests are expressed in table form. The simplest type of table has several test input columns on the left and a test output column on the right, with each row representing one test. (Such a table uses what FIT calls "column fixtures.") To add a new test, a user adds a row. He then types the input data and expected output data for that row. When FIT is executed, it takes the input data, feeds them into the function under test, and compares the output of the function to the expected output. It colors the output cell of each test green if the actual output matched the expected output, and otherwise colors the cell red and inserts the actual value below the expected value.

Below is a simple example of a FIT table for a payroll calculator. The first row just tells FIT which code to execute, which we will describe more shortly. The second row is the column headers. Each remaining row represents a test, so there are three tests in this table. The column headers indicate the column type. Plain headers indicate inputs while headers followed by open and closed parentheses "()" indicate outputs. So in these tests, Hours and Wage are inputs while Pay is an output.

| Accounting.Fixture.PayrollCalculator | | |
|---|---|---|
| Hours | Wage | Pay() |
| 10 | $20 | $200 |
| 40 | $10 | $400 |

| | | |
|---|---|---|
| 45 | $10 | $475 |

When FIT is executed on the payroll calculator table, the first two tests pass, so their cells in the Pay() column turn green, but the third test fails and turns red, as shown below. (If you have a black and white printout, the green should be light shading, and the red should be dark shading with bold text.) This third test is the specification of a new feature. An overtime rule will be added to the payroll calculator in this iteration so that hours above 40 will be applied with 1.5 times the wage rate. A developer now has not just a description of the overtime rule but an example. When the developer is done coding, she can execute FIT again, and the third row will turn green if she implemented it correctly.

| Accounting.Fixture.PayrollCalculator | | |
|---|---|---|
| Hours | Wage | Pay() |
| 10 | $20 | $200 |
| 40 | $10 | $400 |
| 45 | $10 | **$475** ***expected*** **$450** ***actual*** |

FIT tests are designed to be very expressive, so that non-technical people can write and read the tables. Furthermore, since these tables can be embedded in HTML or ordinary Microsoft Word documents, users can provide context to the tests. For example, for the payroll calculator, the table could be preceded by a verbal description "Demonstrates pay calculation. The first two tests just multiply hours by wages while the third test demonstrates that 1.5 * wages are used for overtime hours." FIT knows to skip over everything but the tables.

In order to keep the test interface simple and impervious to code refactoring, developers usually create an object—called a *test fixture*—that translates between the test's interface and the code's interface. For the payroll calculator, the test fixture would have settable public member variables called Hours and Wage. Then it would have a method called Pay() that does any necessary set-up, calls into the appropriate real objects in the system under test, and returns the calculated pay.

## A More Realistic Example

Now, if we are going to do end-to-end testing of a reasonably complex system, there will be far more complicated test scenarios than the one above. They may involve configuration parameters for the set-up, connecting to databases and so on. Rest assured that FIT can do it all, and our example was the simplest possible. FIT tests can extend over multiple tables, some of which do essential set-up for other tables. FIT also offers many types of pre-built fixtures that let you break out of the basic NxM grid we demonstrated. For a complete description, we refer you again to Mugridge and Cunningham [5].

Despite this extra complexity, non-experts can easily understand the tables. To prove our point, we present a more complex example of FIT tests for a shopping cart application. The cells in green represent passed tests. We bet you can understand the requirements even without knowing how exactly the fixtures work or are implemented.

Load inventory to be used for tests and confirm 5 items loaded.

| Fit.ActionFixture | | |
|---|---|---|
| Start | com.valtech.service.tests.ItemInventoryFixture | |
| Enter | Inventory | ./src/com/valtech/post/service/tests/inventory |
| Check | Total items | 5 |

Make sure the items have the correct UPC, descriptions and prices.

| com.valtech.service.tests.ItemInventoryDisplayFixture | | |
|---|---|---|
| UPC | Description | Price |
| 2458 | Chocolate | 0.75 |
| 1244 | Cola | 0.99 |
| 3214 | Milk | 2.34 |
| 8743 | Eggs | 1.89 |
| 0987 | Olives | 3.15 |

Confirm you can select an item and change its description.

| Fit.ActionFixture | | |
|---|---|---|
| Start | com.valtech.service.tests.ItemInventoryFixture | |
| Enter | Select | 2458 |
| Check | Description | Chocolate |
| Enter | Description | Dark Chocolate |
| Check | Description | Dark Chocolate |

Confirm you can add a new item.

| Fit.ActionFixture | | |
|---|---|---|
| Start | com.valtech.service.tests.ItemInventoryFixture | |
| Enter | Add item | 1112 |
| Enter | Description | Honey |
| Enter | Price | 5.60 |
| Check | Total items | 11 |

These examples demonstrate that with an expressive testing tool, developers and customers can collaborate on writing tests. The concrete examples in each test are a meeting ground for domain knowledge and code functionality, understandable by everyone on the team. Furthermore, the tables can be executed to ensure that what they claim should be true really is currently true—which is why they are called "living documentation" or "executable requirements."

So if functional testing is so easy, why isn't everyone doing it? Well, there are a number of pitfalls that we describe next, including how to avoid them.

# 4. IMPLEMENTATION SMELLS

Your first attempt at functional testing might encounter problems. We've encountered two broad classes of functional testing problems. The first class involves the implementation of the functional tests themselves; the second is related to the (un)suitability of our system under test.

We describe these problems in terms of "smells," which are early warning signals that the development process needs to be "refactored" [7]. In this section, we consider smells of poor implementation and offer the techniques that can alleviate them.

## Little (or No) Accountability for Broken Tests

If there is no accountability for broken tests, then they don't get fixed. In general there is no accountability if it is difficult to tell whose code change broke the test. We have found that this usually happens when the test-run cycle is significantly slower than the check-in cycle of developers; that is, if several developers have checked in their code since the last time the tests were run, it is difficult to determine whose changes broke the tests.

### *Solution: Functional Tests In Continuous Build*

We *strongly* recommend including functional tests in the continuous build. Inclusion in the continuous build was also recommended in Gandhi et al.'s experience report [8]. In a traditional development environment without a continuous build, the functional tests should be run after every check-in. Another variation is to use a "functional test cop" as described in "" Slow Tests Removed From Build Stay Broken" below. Remember, the goal is to identify the check-in that broke the tests.

### *Technical Tips for Speed*

In order to get functional tests into the continuous build, the tests must be made fast enough. First, the team must make a commitment to functional testing as a primary development practice instead of a secondary one. When it is not an option to drop the tests, then teams find creative solutions. The main thing is to speed up the running of the functional tests so they can be run effectively by developers on their local machines before checking in. Effective strategies we have found are:

- *Functional Tests on Separate Machines:* By grouping tests into related suites then each suite can easily be run on its own machine. This effectively parallelizes the test suite and can give a speed increase proportional to the number of machines used.
- *Functional Tests Rollback Database Transaction:* This is a very simple but effective idea – don't commit your database transactions if you are testing end-to-end. We have seen this practice emerge independently on different projects and this usually gives about an order of magnitude increase in speed.
- *Functional Tests Refactored to Thinner Slices:* By testing a small scenario within each test instead of several scenarios (or even all scenarios) for a use case we get a finer granularity for splitting up tests. We have also found that larger tests tend to have more redundancy – breaking them up allows for faster individual tests.
- *Functional Tests Grouped By Business Area:* Grouping functional tests by business area allows a developer to test the subset of relevant tests on their machine without running the full suite. This allows for a faster red-green-red test loop and will keep a test suite from slowing the pace of development.

Note that having independent database sandboxes for each functional test run is a prerequisite for the above advice. If two functional tests run against the same database, one may report an incorrect "failure" because of interactions with the data inserted by the other test.

### *Related Smell: Confidence in Functional Tests is Lost*

Leaving tests broken takes away from much of the value of the functional test suite as a "safety net" that prevents bugs from entering the build in the first place. The tests aren't catching the bugs and helping us keep the code in working order as we would expect. Without this safety net, confidence in the

tests is lost. Test writing is reduced, and in the more serious cases they are deleted and finally dropped as a whole.

### *Narrative: Slow Tests Removed From Build Stay Broken*

The context of the following example is from a large leasing application after one year of practicing XP with a 50-person development team consisting of about 30 developers, 7 analysts, 8 testers and management. The code base was over 500,000 lines of executable code and the technology was J2EE with EJB 1.0.

When we first started implementing functional tests we weren't quite sure how much value they would have, but we had a very smart and experienced consultant advising us to do so. We knew we were missing inter-object testing and our xUnit tests were testing unit and more increasingly "integration" tests by testing systems of objects together. We had greatly reduced the errors found by the testers in QA, but there were still many getting through. Also, we had several cases of the developer saying they were "done," but when his code was reviewed, there was either missing or incorrect functionality even though the unit tests passed. So those were the driving factors to implement functional testing.

But functional tests were slow and the build went from 20 minutes to 50 minutes. We decoupled the functional tests from the build and their time shot up from 50 minutes to 120+ minutes over the next few months. Now every 4 or 5 builds, one set of functional tests would be run, and we didn't know who exactly broke the test. Several check-ins had happened and everyone *knew* the failure wasn't caused by their code. The tests would break and stay broken for over a week, and frequently we needed someone to step up and be a "hero" to clean up those stupid tests! Sometimes (ok many times) we thought they were more trouble than they were worth.

Thankfully, we didn't drop them. I don't remember who, but someone on the team stepped up and proposed that we have a coded functional test (CFT) cop. This person had the painful job of watching the CFTs and fixing them when they broke. Of course this was a pain, and one cop got tired of it and dug into the CFTs to try to make them faster. With a few solutions such as *Functional Tests on Separate Machines* and *Functional Tests Roll Back Database Transaction* and *Functional Tests Refactored to Thinner Slices* (described in the section above) the CFTs were running in less than 20 minutes and brought back into the build.

Surprisingly the functional tests stopped being broken because developers could run them effectively on their local machines before checking in. Even if they missed something, the CFT was run with every build, so broken unit tests were immediately fixed because it was (almost always) obvious who the culprit was.

## Small Code Changes Break Many Tests

When many tests fail, one normally assumes that a big code change must have been checked in. However, if only a small change caused many failures, then there must be a large amount of overlap of the tests.

### *Solution: Each Test Focused on One Thin Slice*

When each test focuses on one thin slice of functionality and does not overlap much with other tests, then it's more likely that only one or two tests break when a bug is introduced. It is much easier to diagnose why a thin test failed. Thus, writing tests to exercise one thin slice of functionality in one major system provides the best feedback on that example of a business process.

### *Related Smells*

If your functional tests cover too much ground, you may notice these smells:

- Many test fixtures must be used in a single test
- Developers get frustrated with updating many tests for small code changes

### *Narrative: Trying to Test Everything*

We experienced the smell of small code changes breaking many tests on a project of about 15 developers who had developed a code base over two years (though it was integrated with a larger, 10-year-old code base). At that point, the team decided to add functional tests, beginning with the code they were currently working on, called project B. They thought it would be best to test with all real objects (rather than mock objects) in order to maximize the test coverage for each functional test.

The team spent a month setting up their first functional test. This set up included writing a *test fixture* for each class, which is code that mediates between a test specification (e.g. a FIT table) and the appropriate object in the system under test [5]. Since many parts of the system were "upstream" of the code they were working on, they had to write many fixtures before they could reach the part of the system that they intended to test. The result was then when anyone made a code change "upstream" of project B, all of the tests for project B failed and had to be updated. Developers became extremely frustrated with the burden of test maintenance.

One solution is to mock out parts of the system that are not the focus of your current test. We can use mock objects as we do with unit tests, and for functional testing we can also mock subsystems. Mocks mean you don't have to write "real" fixtures for everything upstream.

Similar principles are echoed in Mugridge and Cunningham's book [5], which advises teams to "avoid over-commitment to details that are not essential to the specific business rule…[and] focus on only one business issue, so that it is less vulnerable to change" (p. 156).

## Functional Tests Try—and Fail—to Catch Unit-Level Bugs

If functional testing does not reduce the bugs found by your testing group and customers, the problem may be that the bugs are at the wrong level for functional tests.

### *Solution: Unit Tests Support Functional Tests*

Functional tests are *not* a replacement for unit tests, even if the coverage statistics look high. Unit tests support functional tests by exercising the code most likely to break, even if it is buried deep in otherwise inaccessible parts of the system under test. Use unit tests for unit-level bugs and functional tests for interaction bugs.

### *Related Smells*

If you use functional tests without unit tests, you may experience these smells:

- It's hard to diagnose failed tests
- Test fixtures work around known issues rather than diagnosing and fixing them

### *Narrative: Pathological Functional Tests*

The previously mentioned project with 15 developers had a cluster of three or four classes that was repeatedly the source of bug reports. The classes already had unit tests, so the team tried to reduce the bug count with functional tests. But the developers writing the test fixtures coded around the buggy classes so that they could get their use case for the functional test done. For example, the developers discovered that their fixture had to call "Save" twice to get the object saved properly.

Why didn't the developers fix the "Save" method? They explained that saving was only a small, initial part of their use case, and their usage did not go deep enough into the code for them to diagnose the problem. So the bugs were not getting fixed.

Finally, the team assigned two developers to refactor the module and improve its unit test coverage. They quickly discovered that the unit tests were inadequate because they were some of the first unit tests the team had ever attempted to write. After a month of work, the module was cleaned up. It was no longer the source of bug reports. The functional test fixture could call "Save" only once. But it was the unit tests, not the functional tests, that ensured this basic functionality.

### *Unit Testing Complements Functional Testing*

Unit tests make sure the units are working properly; functional tests make sure the units interact properly. It is very difficult to use a test of interactions to improve the units themselves. If basic functionality is buggy, focus on refactoring and unit testing the individual classes. If the units are solid but don't interact correctly, use functional tests. We need both kinds of tests.

A commonly cited reason for adopting agile development techniques is the increased communication between the developer and customer to *really* solve the problem and use iteration and feedback to come up with a good solution. Well, unit testing does not address this issue at all and functional testing greatly improves this communication. Asking, "Which testing is more important" is equivalent to asking, "Are requirements quality or code quality more important?" You cannot drop either—you *must* have both for a successful software system.

With that said, let us provide detail on how unit testing is more powerful than its coverage numbers would suggest.

### *Unit Tests Cover Important Code Paths*

Unit tests exercise the most important code paths more easily than functional tests can. Imagine two classes, A and B, each with 5 code paths, $A_1$ through $A_5$ and $B_1$ through $B_5$. Consider writing unit tests for the two classes. $A_4$ and $A_5$ are a getter and setter respectively, so we don't write unit tests for them. We write one test for each other code path for a total of 8 tests. A code path coverage metric would tell us we have 80% coverage. But because we did white box testing, we know we covered the 80% that was most likely to break.

Now consider functionally testing the two classes. Let's assume class A is called before class B and that it's easy to set up three of the tests: Test 1 exercises $A_1$ followed by $B_1$, Test 2 exercises $A_2$ followed by $B_2$, and Test 3 exercises $A_3$ followed by $B_3$. All three tests incidentally exercise the getter $A_4$ and setter $A_5$. With just these three functional tests, we again have 80% coverage.

Unfortunately, the functional tests have failed to exercise code paths $B_4$ and $B_5$. These code paths are triggered by exceptional circumstances that are difficult to set up in a functional test. For example, $B_4$ could deal with a division by zero that results when certain combinations of values are produced by class A, and $B_5$ could handle an exception thrown by a resource. So the functional tests' 80% coverage does not include the code that is most likely to break. Instead, functional tests tend to focus on the "main success scenarios" of the use cases. That's helpful

coverage, of course. But it is unit tests that ferret out the most common bugs.

Furthermore, as the code base grows, it becomes harder for functional tests to cover code that is many classes deep into the system. The functional test has to provide the input to A that leads B to output something to C that causes D to throw an exception so the test can make sure E handles the exception correctly. It's much easier to just write a unit test for E.

### Our Testing Tool is in the Foreground

An immature functional testing tool can lead developers to spend more time getting the tool to work right than they spend on understanding the domain and specifying the tests with customers. Of course, it's important that developers are trained in the functional testing tool, and there will be some start-up costs when they first start using the tool. But if the tool is the root of the problem, you will notice functional testing smells:

- It takes a long time to write tests and test fixtures; the team spends more time on fixtures than test specification
- It's hard to diagnose incorrect test fixtures
- Developers and customers complain about functional testing

#### *Solution: Don't Rebuild the Wheel – Use a Mature Tool*

We recommend starting functional testing with an established tool that has a track record of providing good feedback for customers and developers. Framework for Integrated Tests, called FIT for short, is an example of a widely used tool that provides good feedback [5]. Teams may already have their own tools, of course. But if the tool is taking over your testing, you may want to reconsider.

#### *Narrative: Changing Tools*

Recall the 15-developer team who spent a month writing their first functional test. This team was using a home-grown functional testing tool. The tool had a number of advanced features, but it did not provide good feedback when a test was incorrectly specified or fixturized: it was common to get a null reference exception somewhere deep in the tool code. Customers simply could not diagnose the test output. Developers had to attach a debugger and step through the test. They spent significantly more time in the debugger than collaborating with customers to write tests. Both developers and customers complained about working on functional tests.

This team is now in the process of switching to FIT. The very same developers who complained about functional testing are now clamoring to be the first ones to try the new tool.

A good tool lets you focus on the domain and the requirements; the tool itself "fades into the background" [6]. If the tool is in the foreground, you need a better tool.

## 5. ARCHITECTURAL SMELLS

If you are using good tools and techniques and it's still hard to write functional tests, then the root problem may be your system's architecture. In particular, if your test fixtures contain business logic, rather than merely translating test specifications into method calls, then you will want to consider the smells below. We also consider a smell when it is hard for a functional test to run through a single, complete use case.

Functional tests help push business logic into the correct layer (in a tiered architecture) and the correct functional module. When business logic has found its way into the wrong place, functional tests expose the misplacement.

### Fixtures Contain Business Logic to Mirror GUI Work

If you find yourself writing fixtures that must perform business logic so that they mirror what is done in the GUI, you may have an architecture smell. A common cause of such duplicated business logic is the use of a canonical three-tiered architecture having presentation, domain, and persistence layers. Such architecture does not always succeed in keeping business logic away from the presentation layer. In fact, it is very common for GUIs in this setup to contain "control" logic.

For example, a simple GUI to transfer money from one account to another (*account1, account2*) often does the following in the GUI:

(1) *Account1.withdraw($100)*

(2) *Account2.deposit($100)*

This is simple logic, but it is *business* logic and not view logic. So, if your fixture for the *transfer(account1, account2)* function has this logic in it, then you have code duplication with the UI (which is bad), and you have uncovered business logic in the presentation layer (which is worse).

#### *Solution: Service Layer Gets Control Logic*

When you encounter this type of problem, the solution is to pull out the duplicate code in a common place. That place is the service layer [1], which lies between the presentation and domain layers and contains control logic. In this way, functional tests help in proper separation of business and presentation logic and encourage a new logical layer to hold control logic.

#### *Narrative: Building Up Fixtures For Functional Testing*

This story is one from the 50-person J2EE leasing application. As stated earlier, we introduced functional testing after we had gained experience with XP as an agile development methodology. Building our initial functional tests took a large amount of work upfront because we had to build a fixture for every single test. Moreover, we discovered as we started building these fixtures that there was a significant amount of business logic that had seeped into our GUI even though we had both a domain and service layer. The first developers

working on these tests had not only to build the fixtures but also to understand the UIs in detail so that they could refactor them and pull out *all* the business logic into the service layer.

We took two full iterations with a five-person team to do a set of large refactorings for the entire presentation layer. We then had a design session to explain the problem with the old ways of doing things and how they were not testable to the rest of the group. Finally, for the next few iterations, whenever someone was to write their first fixture, they would pair-program with one of the team who did the large refactorings. Over a period of three to four months, we had made several large refactorings to the presentation layer and solidified the boundary between presentation and service layers. We had also reached critical mass with the number of fixtures present so that other developers began to feel comfortable writing test fixtures easily.

## Fixture for a Module Contains Business Logic That Belongs in the Module

There is another way that business logic can turn up in a test fixture—when a functional module fails to contain all the business logic that belongs in it. An example can best illustrate this point.

Let us assume that one of our subsystems is a tax module that is responsible for doing all tax-related calculations. Before introducing functional testing, we wrote this module and believed we had good functional separation. Unfortunately, over the development of our project not everyone using the tax module was completely familiar with it, so some "pre-calculation" was made outside of the tax module depending on special tax-exempt days. This functionality should have been in the tax module; in a sense, the tax module's boundary was breached.

When functional tests were written for the tax module, we would find that the fixture code had to perform the "pre-calculation" that depended on the tax-exempt days. At that point, a responsible developer would notice the duplication and refactor the calculation into the tax module and out of the fixture and the non-tax-module code.

We have found that functional testing frequently solidified the boundaries and responsibilities of our subsystems. Our functional tests help us focus our modules.

## Functional Tests Difficult To Run Through a Single, Complete Use Case

Legacy systems—that is, systems that were not designed with functional testing—can be especially difficult to test. Sometimes they do not let you easily run through a single example of a business process. This is a very difficult smell to eradicate, and the solution depends on the architecture.

In some cases, the source of the problem is that a module assumes that multiple use cases are run simultaneously. When you try to isolate a single use case, you discover you still have to perform the set up for all the other use cases or the system crashes. We provide an example of this situation below. We highly encourage you to listen to your tests—if they are hard to write, then they are indicating a larger problem.

### *Narrative: An Executable Calculator*

The project with 15 developers mentioned earlier had an architecture that made some of its primary uses cases difficult to test. The system used C# for presentation; this code allowed the user to enter input data and view output data. The system used C++ for the main business logic and calculations. However, what made the system tricky was that the main medium of inter-language communication was the database. The C++ was an executable that accepted a handful command line parameters; it read hundreds of additional inputs from the database and wrote its outputs to the database.

To execute a single business process in such a system, we had to set up a fairly complete database with a lot of extraneous information that did not matter for the process we wanted to test. After this set-up, we could enter the one record we wished to test through the service layer of C#. Then we would fire off the C++ executable, which would perform far more calculations than we actually needed for our test. Finally, we would check the results in the service layer of the C# output screen.

Because testing one use case was so burdensome, the team tried shortened use cases. They wrote functional tests that entered the data in the input screen's service layer and then confirmed that the values were saved correctly to the database. These tests failed to exercise the most important business logic of the system, so the analysts were not very interested in whether they passed or failed. After all, these tests rarely found bugs that really mattered. After a few months, both customers and developers resented the functional tests as a waste of time.

To make this architecture more amenable to functional testing, we would have had to convert the C++ code into a library (e.g. a dll). Then we would have exposed individual methods so that the calculator would not always process everything in one batch. Then a test could set up just the data needed in C#, call a handful of C++ library methods, presumably through a new service layer, and confirm the results in C# again.

These architecture changes would not have merely made the code more testable; they would have made it more agile. Clients later requested real-time updating of the calculations as new input data became available throughout the day. If the C++ code had been a library that could fire off single requests, adding real-time updating would have been a snap. As it is, the system is not expected to offer real-time updating for years.

## 6. CONCLUSION

Functional testing is a practice that can have great benefits to the development process as a whole. When done properly, it increases the communication between analysts, developers and testers. The progress of the entire project is objectively visible at any point in time to management by examining the passing (and failing) functional tests. Eventually, the speed of development increases because well-communicated requirements result in less re-work. The tests also drive a more modular architecture with subsystems that have clear responsibilities.

However, functional testing is not free. A significant investment must be made to *get it right*. Cutting corners can cause myriad problems that we have outlined in the smells sections. If the smells are not addressed, the costs of functional testing can outweigh the benefits.

So we recommend that you evaluate your current environment to determine whether functional testing addresses your needs and provides useful benefits. Then, take a careful look at the costs to functional testing as indicated in the *When Not To Use It* section to make sure that you are willing to make the commitment. And if you adopt functional testing, pay attention for smells so you can catch problems early.

With the right techniques, we have seen developers and customers get excited about functional testing. They enjoy learning about the domain and its requirements in a deep way. And they take great pride in the high-quality software that results, on time and within budget. Functional testing is a pattern that works.

## 7. ACKNOWLEDGMENTS

We would like to thank Robert Osborne, Steve Sparks, Jason C. Yip, and two anonymous reviewers for thoughtful comments and suggestions. Furthermore, we would like to thank the PLoP 2006 workshop members for reviewing this work yet again and providing useful feedback these are Ralph Johnson, Paul Arumi, David Garcia, Jason Yip, Hesham Sadawi, Leon Welicki, D. Bellibia, Dirk Riehle, and Paddy Fagan.

## 8. REFERENCES

[1] Fowler, M., *Patterns of Enterprise Application Architecture.* Pearson Education, Boston, MA, 2003.

[2] Highsmith, J., *Agile Software Development Ecosystems*, Pearson Education, Boston, MA, 2002.

[3] Marick, Brian. "Bypassing the GUI." In *Software Testing and Quality Engineering*, (September / October, 2002), 41-47.

[4] Evans, E. *Domain Driven Design: Tackling Complexity in the Heart of Software.* Addison Wesley, 2003.

[5] Mugridge, R., and Cunningham, W. *FIT for Developing Software: Framework for Integrated Tests.* Pearson Education, Upper Saddle River, NJ, 2005.

[6] Jim Shore, "A Vision For Fit," http://www.jamesshore.com/Blog/A-Vision-For-Fit.html

[7] Elssamadisy, A., and Schalliol, G. "Recognizing and Responding to 'Bad Smells' in Extreme Programming." *ICSE 2002*, pp. 617-622.

[8] Gandhi, P., Haugen, N., Hill, M., Watt, R. "Creating a Living Specification Document with FIT," http://www.agile2005.org/XR22.pdf