Get the Balance Right: Acceptance Test Driven Development, GUI Automation and Exploratory Testing

Michael Larsen
mkltesthead@gmail.com

Abstract

Depending on who you talk to, all of our testing challenges can be solved by doing some particular variant or flavor of “in vogue” software testing techniques. In the Agile community, Test Driven Development and Acceptance Test Driven Development are quite hot properties. So hot, in fact that some programmers and programming managers have declared that "Test is Dead" or, at least, is figuratively so. Others speak to the proliferation of frameworks and tools that allow for automated testing of the front end/GUI interface, and that this spells the future of testing. Others say that "active, sapient and human” exploratory testing is the truly effective method of performing software testing. Which group is right?

The answer is "all of them are right, and none of them are right". All three of these approaches, applied with maximum effort and efficiency as standalone initiatives, will not guarantee bug free code. Taken together, with a good understanding of where each area excels, where each area has deficiencies, and where each can leverage the strengths of each other, programmers and testers can bring a balance to development and testing efforts that are much more likely to find the issues that matter.

Biography

Michael Larsen is a senior tester with SideReel.com, a wholly owned subsidiary of Rovi Corporation, currently working at the Rovi site in San Francisco, California. Over the past two decades, he has been involved in software testing for products ranging from networking equipment to capacitance touch devices to Internet applications. Michael serves as a Director for the Association for Software Testing (AST) and is the Chair of the Education Special Interest Group. He actively teaches software testing through the Black Box Software Testing series of classes offered by AST. He is a black belt in the Miagi-do School of Software Testing, is the co-founder and primary facilitator for Weekend Testing Americas, and is the senior producer (and frequent commentator) for Software Test Professional’s “This Week in Software Testing” podcast.

Michael writes the TESTHEAD blog at http://mkltesthead.com/ and can be found on Twitter at @mkltesthead.

1 Introduction

It's been a little over ten years since the Extreme Programming (XP) community, and other parties interested in improving the craft of software development, came together in Salt Lake City and developed what would become the Agile Manifesto. There is no question that a seismic shift in the way that software is being written, tested, and delivered has come about because of that meeting. Many new terms have been added to our software development lexicon because of it. Pairing of developers, testers, and combinations of either are now common in software development teams.

The notion of a dedicated testing team as a standalone entity is changing, with self-organizing teams covering many disciplines that were, and still are, treated as separate in traditional development environments. With these changes, however, a philosophical debate has emerged. By embracing Test Driven Development (TDD), programmers write tests first to make sure that the code is able to meet the call of what is being designed. Programmers then make sure code, as it is written, continues to pass existing tests. New tests are created to define and inform new functionality. Extending this model to incorporate the business goals, as well as defining them earlier, the concept of Acceptance Test Driven Development (ATDD) has gained popularity. We can look at ATDD as being synonymous with the Steven Covey rule to "begin with the end in mind", and making sure that the functionality being developed actually meets the criteria being requested by the organization [Gärtner, 2012].

We have seen an explosion in the availability and the capabilities of robust, effective and, in many cases, free software testing tools. Gone are the days when testing tools were only available from commercial organizations that charge exorbitant fees for licenses and proprietary standards that would only work within their applications. Languages like Java, .NET and Ruby have allowed the growth and popularity of frameworks like FitNesse, Selenium, and Watir. Natural language tools such as Cucumber structure tests into readable specifications that anyone in the organization can read and, theoretically, create. The ability to create tests to automate the front end and user interface of applications is more prevalent than ever.

Over the past several years, I have seen a number of industry figures look to these trends and say that, because of the developments of these tools and technologies, as well as using Continuous Integration (CI) to manage and deploy builds, dedicated testers aren't needed any longer. Testing, as has been said in a number of different mediums, is dead [Savioa, 2011]. To counter that, many voices have also come out to say that the ability to apply reason and dynamic thought to problems, to use "sapient skills" and active exploration, and that thinking, actively engaged testers are needed now more than ever [Tomlins, 2011].

This paper looks to explore all three of these areas, explore the strengths and weaknesses of each, and show that we need all of them to work together to be effective. Additionally, I will offer some suggestions as to how we can leverage all three areas to drive our testing efforts so that we can deliver what is most important, which is real value to our customers and stakeholders.

2 TDD, What Is It?

Test Driven Development (TDD) is an ongoing process, where a programmer works on small pieces of functionality, and in the process of writing the code, create tests that determine if the code in question is doing what it is supposed to do. Kent Beck wrote the book “Test-Driven Development by Example” in 2003, and in its preface, he sets out the primary goals of TDD. In short, the goal of TDD is to "create clean code that works" [Beck, 2003]. The process for doing that is the core focus of TDD.

To reach this goal, the programmer creates tests first that are meant to focus the expectations of the software. Each test, as it is written, is first written to guarantee a failure (after all, the code associated with these tests doesn't exist yet). Once the test fails, the programmer then works to create the functionality that will make the defined test pass. After doing this, the programmer(s) then examines the code they have created and clean it up. Repetitive steps can be placed into functions or methods. This approach is encapsulated as "Don't Repeat Yourself" (DRY) and is formally called "refactoring". It is meant to make sure that the functions and procedures created answer the functionality needs, and that the code in question has a minimum of replication and duplication [Beck, 2003].

After cleaning up the code, the programmer runs the tests again, to make sure that the tests still work and that the cleanup/refactoring has not introduced new issues or caused the tests to fail. This process is used throughout the entire software development cycle. Each function, new feature, and new user story is driven by creating tests first, and then writing code that makes it possible for all cumulative tests to pass. If tests fail, then the process repeats itself until all tests pass (and all code, whether newly or previously written) passes all of the tests.

2.1 ATDD, how does it differ from TDD?

Acceptance Test Driven Development, while it may share many of the same words in its name, has a different focus from Test Driven Development. From One of Microsoft's teams in Israel, they describe the difference as “Unit tests (TDD) is about building the *Code Right*. FitNesse (ATDD) is about building the *Right Code*” [ArnonA, 2011].

TDD places the focus on functions, classes and methods. The goal is to make sure that the code that is written meets the specific goal of that piece of functionality, or for those given functions and methods, and that the resulting code is as clean as possible. By contrast, ATDD places the focus on User Stories. Stories are developed to explain the desired functionality, not from the perspective of functions and methods, but from the expectations of the customers. User stories can cover variations of functionality. It can mean anything from a new interaction screen, enhanced text, and improvements in performance, usability or other aspects that describe a feature as a user would see it.

TDD is used by programmers, and focuses on performing tests at the unit level (typically written in the same language as the application being written). ATDD is geared towards the members of the team that are not necessarily programmers. ATDD uses tools that allow a more natural language to present the requirements. The requirements scripts will look like typical manual tests, but they map to underlying automation code. This allows an automation framework to run our acceptance tests as actual executable steps. Various tool can be used (Cucumber, FitNesse, etc.) that allow testers to verify that, indeed, the programmers are building the Right Code.

Here’s a very basic scenario example, a user logging into Twitter:

```
Scenario: Twitter user signs in through Twitter connect button on sign in page
  Given I connect to twitter from the sign in page
  When I log into twitter with good credentials
  Then I go to the user private profile page
  And I can confirm my twitter credentials
```

The syntax above is Gherkin, which is coupled with Cucumber to allow the system to map the scenario statements to step definitions, which in turn map to actual commands in the programming language being used. In my environment, this would be Ruby, but Cucumber can interact with other languages and frameworks as well, including Java and .NET [De Florinier, Adzic, 2010]. We will see this same example get expanded when we discuss automating the Graphical User Interface (GUI) later.

2.2 Why are TDD and ATDD not Test Techniques?

For all of the talk about tests being a core part of both TDD and ATDD, we need to be clear; neither TDD nor ATDD are about testing. TDD and ATDD are design tools. They are used to help ensure that the software being developed meets the needs of the project. They help keep the project goals in focus. They help to ensure that the code developed does not get overly heavy or tries to accomplish too much. They help to ensure that "dead areas" of code do not develop, or can be removed. They arguably help make sure that programmers create more error free code. Every one of these is a win and a positive development, but they are not performing active testing.

What they are doing is repeatedly checking to make sure that the tests that are built into the system continue to work, and they help make sure that the code that has been written still meets the criteria of the design. What these tools do not do is perform active and dynamic testing of code, in the way that a focused, inquisitive, engaged and potentially devious human being can. Testing is the art and the act of asking questions of a product, and then developing and devising new and more inventive questions based on the answers we receive. Neither TDD nor ATDD do that. What's more, TDD and ATDD cannot test for questions developers never thought to ask.

2.3 Where do TDD and ATDD Excel?

Both TDD and ATDD are excellent ways to provide a disciplined approach to developing cleaner software. TDD and ATDD act as a brake on "cowboy code". It prevents software being developed that requires concentrated debugging days, weeks or even months after code was written. It also makes it possible for programmers who have not worked on a project to see the code in context with the tests that have been written.

TDD and ATDD both provide an advantage when it comes to creating Continuous Integration environments. Continuous Integration is the ability, whenever a new component or method has been committed to a project, to automatically run the tests associated with that module, rebuild the rest of the application, and run the tests associated with all of the features in the software application. These checks are performed repeatedly to ensure that new functionality works as the original design intended. Running these tests every time that the software changes keeps the code clean, as well as helps keep issues based on dependencies to a minimum.

Both TDD and ATDD are well adapted for Agile software development environments. They are well suited for short iteration cycles and the ability to work on small subsets of functionality in an iterative manner. Each directs the programmer to focus their attention performing "just enough design" and "just enough development". It eliminates the need for having to do all of the planning up front. The ability to focus on small areas of the code also has the benefit that refactoring can be done on a regular basis with each module as they are developed.

Another advantage is that the entire product team can handle the design and development work. The customer or their advocate also participates. TDD is typically done at the component level. ATDD allows for a broader range of technologies to be used. If we write our TDD code with Ruby, to perform ATDD, it is common to set up an environment where tools such as RSpec and Cucumber are utilized. Cucumber uses a language called Gherkin that allows anyone on the team to craft tests by creating Scenarios. Those scenarios and their statements are mapped to underlying commands. These underlying commands are written using Ruby, and they allow for the actual automated steps to be run. The big benefit from this is that teams can work together to develop tests. Even team members that do not have programming experience can understand the flow of tests and create their own scenarios.

2.4 What are TDD and ATDD's Deficiencies?

Both TDD and ATDD can give those who use them unrealistic expectations. The system of TDD and ATDD is not a magic bullet that will guarantee clean code that is bug free and without issues. Neither TDD nor ATDD replace good design or coding skills. The programmer is ultimately in control, and thus the framing will only be as good as the overall skill that the developer has [Tchepack, 2003]. There is also a tendency to over think or over apply these tools. When a change to a table format is needed for a web page, is it really essential to write a failing test first before making a change to the table structure?

Additionally, for many who are new to the ideas, the common approach is to "follow all of the rules" as they have been laid out and follow them all the time. Of course, context matters, and these tools are really only as good as the current context allows them to be.

The biggest challenge that is still left unanswered by TDD and ATDD is that, while they are focused on having tests run to determine that the code is being written the right way, and that the right code for the right purpose is being met, there is a limit to how much a handful of acceptance tests and unit tests can go in determining if the software being written is really going to behave well when a real human being starts interacting with it [Dalke, 2009]. While TDD and ATDD can tell you if an object exists, or if a page element is visible, it can't tell you if the load time is lagging, or if the order of elements appearing in the time frame it does will be frustrating to an individual user. It also doesn't say if the objects that are appearing are doing so in the correct context of the application, doing so at the right time, in the right place, when they are actually needed.

Additionally, TDD and ATDD do not answer aesthetic issues, those things that, while technically correct, actually require a human to determine if they appear in a way that is pleasing or in a way that is helpful.

3 What is Front End GUI Automation?

The Graphical User Interface (GUI) is how most people interact with computers and mobile devices today. Yes, there are some who are familiar with how to use the command line tools that come with various operating systems. The majority of activities, especially on the web and with mobile devices, are performed using the GUI. Unlike programs that start and stop from the command line, the GUI runs all the time, and programs are written to be always available and “waiting” for new input or direction.

To simulate the actions of users interacting with a GUI, there has been a proliferation of tools that help programmers and testers simulate the actions of users. This simulation activity is broadly referred to as Front End GUI Automation.

There are many reasons to do this type of automation. The most common is to follow standard workflows (logging in, navigating to pages, clicking on links, filling in forms, etc.). These can be as lightweight as a simple form filling tool (such as Texter) all the way up to full feature web page and application automation tools such as Selenium/WebDriver, FitNesse, TestComplete, and others.

3.1 How Does GUI Automation Differ From TDD/ATDD?

GUI automation is often used to help with constructing acceptance test cases and using those cases to check and demonstrate that the acceptance criteria has been met.

Expanding on the example mentioned in section 2, we use Cucumber to represent an acceptance test:

```
Scenario: Twitter user signs in through Twitter connect button on sign in page
  Given I connect to twitter from the sign in page
  When I log into twitter with good credentials
  Then I go to the user private profile page
  And I can confirm my twitter credentials
```

The Cucumber statements are an abstraction of what is actually happening in the code. Our GUI automation tool would then take these statements and map them to the appropriate tools that would make the statements work on the application or browser.

The below statements are groupings of selector statements. This technique is helpful when we want to group technical and specific statements so that they are not part of the actual acceptance test verbiage:

```
Given /^I connect to twitter from the sign in page$/ do
  step %Q|I am on the user sign in page|
  step %Q|I click "Connect your Twitter account" within ".login-container"|
end

Given /^I log into twitter with good credentials$/ do
  step %Q|I fill in "username_or_email" with "mytest497tw" within ".sign-in"|
  step %Q|I fill in "password" with "S4p3rl0g1n!" within ".sign-in"|
  step %Q|I click "Sign In" within ".buttons"|
end

Given /^I can confirm my twitter credentials$/ do
  step %Q|I should see "srtest101tw" within ".login"|
end
```

Each of these statements are mapped in a step definition to actual code that is run (in the examples below, the code is Ruby and we are utilizing Capybara to make the calls to the browser):

```
Given /^(?:|I )am on (.+)$/ do |page_name|
  visit path_to(page_name)
end

When /^(?:|I )click "(^")"(?: within "(^")")?$/ do |link, selector|
  with_scope(selector) do
    click_link(link)
  end
end

When /^(?:|I )fill in "(^")" with "(^")"(?: within "(^"*)")?$/ do |field,
value, selector|
   with_scope(selector) do
    fill_in(field, :with => value)
  end
end

When /^(?:|I )go to (.+)$/ do |page_name|
  visit path_to(page_name)
end

Then /^(?:|I )should see "(^")"(?: within "(^")")?$/ do |text, selector|
  with_scope(selector) do
    if page.respond_to? :should
      page.should have_content(text)
    else
      assert page.has_content?(text)
    end
  end
end
```

These are steps that allow the system to send commands to a browser and walk through them so that the login process can be completed and then verify that the user is seeing elements that give them some satisfaction that the steps performed their job.

3.2 Where Does GUI Automation Excel?

GUI automation can be a tremendous blessing when dealing with repetitious steps that need to be performed for set up and take down of test environments. It's also helpful when certain states need to be created and prepared for testing. GUI automation can be a positive aspect and an important part of Acceptance Test verification and checking to see if the workflow as requested behaves appropriately.

GUI Automation also acts as a proxy for human interaction. For test steps that require a significant amount of interaction and verification, where components are loading and are visible in an application or on a web page, GUI automation helps the tester get to the point in the program where they can then examine interactions directly.

3.3 What is GUI Automation's Deficiencies?

As Corey Goldberg aptly describes, the GUI tends to be the most fragile layer of an application. It also tends to be the layer where a lot of automated test infrastructure is built [Goldberg, 2008].

There are a number of areas where GUI automation falls short of its intended goal. Some of this has to do with the way that testing tools have been marketed and are frequently used, especially by those who are inexperienced with test automation. There has been a great deal of rhetoric over the years regarding GUI Test Automation Tools and the way in which they are meant to be used. We have been promised test tools that provide record and playback simplicity. Suffice it to say that, in many cases, these tools have been oversold. While they do indeed record and playback, usually this creates overly specific scripts where the slightest change in the application will break the script. This creates a tremendous overhead of script maintenance or, often, the need to throw the scripts away entirely and start over again.

Even when the user can automate user interface interactions, there is a limit to how much can be realistically done. The basic steps for the acceptance test can be performed, following along the course of standard workflows and covering the steps that could be considered the "happy path", with some additional error handling for a few side cases. Automating all user scenarios, of course, would be out of the question, or even a small percentage of them. The return on investment would be low for the amount of work that would be required [Venkatakrishnan, 2009].

Another challenge that we face is that, while a tool will recognize if an element is on the page, it will likely not recognize if the context of that element has changed. An example; what if a user is checking to see if all of the links for a particular page are present? The test checks for the links, and if they are present, then the test passes. But what if, on that same page, the Cascading Style Sheet (CSS) is not loaded, and all of the styles for the page are, subsequently, not loaded? To the GUI automation program, the tests will pass, but a user would see the page and say "wait a minute, this doesn't look right!"

Also, If the appearance, location, text or description of the object changes, human beings have the ability to adapt to change much more quickly than we have the ability to reprogram the tests.

Finally, while front end GUI automation may be effective in helping to find issues the first time they are created, the odds are that these same tests will likely not find many, if any, issues beyond when they were first created.

4 What is Exploratory Testing?

Exploratory testing comes up in many discussions, but there seems to be confusion as to what it actually is.

One of my favorite definitions of exploratory testing is that it is “scientific thinking in real-time” [Bach, 2012] It's also the opportunity to put test design and test execution together. Exploratory testing is the opportunity to ask a program or application questions. Based on the answers we receive, we can then go any number of directions, and seek new or interesting answers based on the answers to those previous questions.

Exploratory testing is best described as simultaneous test design and test execution. To put it simply, the tests that we did before inform the tests that we will do now, and those tests will inform the tests we will perform later. Being able to have that flexibility and that opportunity to examine different paths is at the heart of exploratory testing. In addition, there are many implicit aspects of how we do things that, when exploited, fall into the realm of exploratory testing as well. Such things as varying the speed that we type, adding or reducing time to perform button clicks on the page, or getting more creative by opening up or closing down system resources so that our system has lots of memory or very little, lots of disk space or very little, or causing the CPU to run at a high percentage. Each of these can tell us interesting things about the software that we are testing.

Exploratory testing is not random, nor is it undefined. It does differ from "scripted testing" (of which automated testing is an example) in that the concepts are defined, but they are not completely pre-defined or run in a rigid sequence. Exploratory testing allows for a development of test ideas that are interesting, and again, are often based on the idea that they will be executed based on the information that previous tests provide answers for (or additionally having not provided answers).

Each time I run a series of tests, and if those tests are run all the time in the exact same way, then that is the epitome of scripted testing. The tests are re-run the same way, every time, without deviation. Suppose I take that same set of scripted tests (using my Cucumber test suite as an example), and I randomize the order in which the tests are run? If the tests are written where there are no contingencies on other tests, then there should be no difference. But often, I see that there is. Why? Because I discover that there are areas where dependencies or state conditions do exist. Randomizing the tests with the goal of seeing if we can uncover some unintended dependencies is an example of exploratory testing; we are conducting a controlled set of tests, but we are doing so in a way that asks different questions each time, and through the process, we evaluate the results to see if we are getting the expected clean runs, or if we learn something new from differing the approach.

4.1 What Makes Exploratory Testing Different From Automated Testing?

Automated testing, for the most part, is a sequence of steps that allows the tester to get to point in the program and run an assertion to see if a condition is met, or if an element is present on a page. It might be helpful if we were to change the terms used. Instead of saying "automated testing", I prefer to use the term "computer aided testing". By using those words as a framework, Exploratory Testing can, and often does, use elements of test automation to help accomplish its goals. The difference is that, instead of creating a mechanical process for running a sequence of steps, we instead look to get as much understanding from the application that we can, and that understanding can help us make further decisions about where we should aim our efforts. Scripted tests that do not deviate from their defined course do not give us that advantage.

Below are some examples of where automation excels and thing automation cannot do. I should point out that, even though automation cannot specifically do the things that are listed in the “Cannot” column, they can add to the ability of helping a tester make decisions based on the information they can provide.

Automation Can:	**Automation Cannot:**
Generate test data to be used in forms or as variable values.	Create curiosity.
Parse the output of a program and use it as input	Make sapient (actively thinking) decisions.

to another program.	
Create a log of actions and transactions.	Invent a new idea or an approach based on the output of a sequence of tests.
Alert if an assertion is met or not met.	Notice something unexpected unless we have already defined what is unexpected.
Search for patterns in output, or help to reveal patterns we did not know about.	Make a judgment call as to the value or importance of a piece of functionality.

Figure 1: A comparison of what we can and cannot do with automation [Bolton, 2010].

4.2 Where Does Exploratory Testing Excel?

Exploratory Testing allows the tester the ability to choose the sequence of steps that, at a given time and in a given context, can provide answers about the state or condition of a product. These tests allow for a very open and dynamic way of questioning an application. If we ask the same questions in the same way all the time, the odds of finding out something new and potentially interesting are much more limited. Imagination and creativity are what opens up the possibilities, as well as being able to determine which tests might be redundant or of lesser interest.

When I take my first look at an application or project, often there is little in the way of formalized documentation. Because of this, Exploratory Testing can be very helpful. Since I have had little interaction with a product at this stage, I am free to look at it and see if I can "figure out" what it should be doing.

Let’s go back and look at our original acceptance test:

```
Scenario: Twitter user signs in through Twitter connect button on sign in page
  Given I connect to twitter from the sign in page
  When I log into twitter with good credentials
  Then I go to the user private profile page
  And I can confirm my twitter credentials
```

We have looked at how we could automate this process so that we can determine if the acceptance criteria is met. The acceptance criteria and the steps to meet it are straightforward. At the same time, we can also ask a number of different questions, many of which will fall outside of the actual story and acceptance criteria:

* Is there a login interface?
* Does it provide for proper error handling if I can't log in?
* Does it give me feedback to let me know that I have successfully logged in?
* Can I trick the system into letting me log in with improper credentials?
* Does it present me with information that could help me guess a login without actually having one?
* Are there ways that I can find out how to configure the application without explicit instructions?
* Can I figure out what a standard workflow with the application might be without explicit documentation explaining it?
* Where else could I use this functionality? Is it just on the login page, or can I find this control in other places? Do the designers even intend to have other places where I can access this control?

All of these examples, when applied, are where Exploratory Testing offers great strength. Each transaction could present me with something unexpected.

Exploratory testing is at the core of TDD and ATDD, at least as far as the initial understanding of what the programmer wants to do to implement the functionality. The idea of proposing a failing test first, and then creating code that meets the acceptance criteria, requires that the programmer give open consideration to what the application needs to do, and determine which of several avenues they will consider to implement the feature(s). Beyond initial development, the first time I create a series of automated test scripts, there is a great deal of learning, fiddling around with the application, and walking down various paths to determine if I have a sufficient number of checks and assertions to create a robust series of steps to make for a meaningful test case.

Even with a number of checks and collective true or false statements, there is no substitution for the human brain to interpret the results and determine if they are meaningful or if they actually provide a real value to the development process. These checks allow us a jumping off point to look in other places, since the tedium of setting up the environment, or providing the data necessary to get us to a significant state, does not have to be manually run each time. One of my favorite tools to use when I am creating Cucumber scripts is to use a statement that is simply called "And let me see that":

```
And /^let me see that$/ do
  puts "PAUSED - Press Enter to continue: #{$1}\a\a\a"
  puts ""
  $stdin.gets
end
```

Programmers are familiar with this kind of a construct; it’s referred to as a “break point” in code, where the program will run until it reaches that point, and then it will stop.

I use this often in my tests to help me make sure that I am able to get to a point that I am expecting to and determine if the conditions are right to perform additional steps, to ensure that elements are where they should be, but mostly it is used to give me a place where I can let the automated steps stop and hand control of the system to me. Frequently, this option allows me to then go and poke at any areas that might interest me. Again, automation cannot substitute for initial curiosity. It can only recreate the steps I tell it to do, and those steps are usually determined after I have had a chance to poke around and explore the system. Once I have done so, I can then go back to the script, hit the Enter key, and the script will pick up right where it left off, either to run the remaining steps in its defined sequence, or to tell me if the changes I have made or the areas I have explored will now produce an error because I have left the application in a state that the script does not know how to handle.

The key is that I have the ability to jump off and consider avenues that the script itself may not yet be coded for. Once I have added additional steps, then they become part of the series of pre-determined paths that I have created [Bach, 2006].

4.3 What are Exploratory Testing's Deficiencies?

The most direct deficiency with Exploratory Testing is also its greatest strength. Exploration is most important when treading on new ground. After I have gone and explored and mapped those areas, I am less likely to find new revelations by going over the same ground. This follows what Boris Beizer calls the "pesticide paradox"; by repeatedly running the same tests, we will not only not find new bugs, but the bugs that are left will be more resistant to the tests we do perform [Beizer, 1990]. Thus Exploratory Testing requires that we constantly be looking for new ways to explore areas we have already been over before.

Another issue is that, because most of the steps I perform require that I be actively engaged in examining new paths, there is simply not enough time to go through all possibilities. Exploration is labor and time intensive. There is a tradeoff; freedom to explore vs. time to perform that exploration. The answer then frequently becomes "why not just automate the steps I already know about"? While this will help run many tests faster, it also restricts the avenues I can drive the application.

Perhaps the biggest challenge with Exploratory Testing is the fact that sentient humans do it. Sentient humans also have a challenge that computers do not have; sentient humans get bored. If we consistently run through a variety of exploratory tests, over time, we may naturally start to fall into a routine. While we tend to think that we are exploring, we are actually treading the same ground, doing the same steps we always have. We start to do things by rote, even unconsciously [Bolton, 2010].

5 Can We Create a Balance?

Every team and every product is a little bit different. As I'm sure you have seen from the examples given previously, none of these steps will stand on their own as a be all and end all for delivering quality software. They are all interdependent, and to speak of doing Test Driven Development, Acceptance Test Driven Development, Front End GUI Automation and Exploratory Testing as separate activities is simply not true. TDD and ATDD, as well as front end GUI automation, are activities that we perform. Exploratory Testing is a mindset and an approach that we use when we perform those activities.

Exploratory Testing can be as manual or as automated as we choose to make it. Test Driven Development and Acceptance Test Driven Development are, by their very nature, exploratory endeavors, at least at first. Developing and defining the acceptance criteria for stories that drive development are likewise exploratory. Front End GUI Automation starts from the desire to capture and use the discoveries we make when we are examining the requirements and how to implement them. In all of these steps, testing is at the center of the activities being performed, and testers very much have a role in them being performed.

When the "testing is dead" conversations started in 2011, it was not to say that there would be no more testing, it was to say that testing needed to be a part of an entire process of quality and improvement, starting at the earliest stages of design and development. The idea of testing as a standalone, separate, after the fact endeavor where a group of people would take a product and shape it into quality, separate from its initial development and design, is indeed dead (or dying) *and that paradigm deserves to die*. Testers cannot "bake in" quality after the fact. We can identify problems and report on them. We can explain issues with design and requirements, but as a standalone group we cannot shape the quality of a product by ourselves. Agile teams over the past decade have come to this realization, and the idea of testing as an end of the line process is largely history for these teams. Instead, exploration, automation, and testing all come much earlier in the process, and involve not just testers, but everyone on the product team.

With that, here are some ways that we can leverage the "testing everywhere" idea, and help develop a balance for all of these goals, and see them as interdependent and not as separate and standalone activities.

5.1 TDD and ATDD are Exploratory

From the first time a story is presented and criterion is established, a development team explores the options available to them. The customers provide us with their goal and their hopes for what the software will do. Developers and testers weigh in on what these requirements mean and how they will be implemented. Rather than testing being seen as an after-effect, it is part of the initial development effort. Rather than saying "here's what we will do and, oh yeah, we will test this all after we finish", testing comes first. By putting the emphasis on testing and making sure that the piece of functionality under development is designed with testing in mind, code can be written to make sure that it meets the criteria it is being designed for.

Requirements do not develop in a vacuum. Instead, they are discussed and the respective stories are fleshed out by examining and understanding the goals of the customer. Testers have an opportunity at this early stage to consider the customer's requirements and offer input on ways that the product can be tested. While programmers consider tests at a unit level for the modules they are programming, testers can also help guide and provide consideration for testability in areas the programmers might not initially consider. This process takes into account different factors depending on the context in which the new code is to operate. Exploration is vital to this process. In many cases, functionality is vague and amorphous early on. Exploring the approaches, asking a lot of "what if" questions, and actually thinking out the ways that the software needs to work, how it might fail, and how we can effectively and helpfully alert the user to errors, is paramount to its success [Pier and Kaner, 2012].

5.2 Focus on Delivering Business Value

As a tester on a development team, we need to perform many tasks. While finding issues or examining acceptance criteria is an important aspect of this, the key area we need to focus on needs to be "is what I am doing helping to deliver real value to my customers"? To that end, does it make sense to do a lot of up-front automation for requirements that may or may not stay intact? Do we want to focus on the business value and how the requirements or goals of our customers can be met? If we put the business value first, that will guide us in our testing efforts and the approaches that we use [Crispin, 2008].

5.3 Automation Does Not Have to Be Permanent

The unit tests that are created to help guide TDD and ATDD are part of the process of writing software, and subsequently, those tests are part of the code base early on. Later automation efforts, such as those that lead to developing integration and regression tests, may have several iterations. Some automation will make sense early on but not be of much help later in the process. Not all automation is equal, and not all automation should be approached as though it will be a permanent part of the testing process. There needs to be room for "throwaway automation”, steps that can be used to help explore different avenues.

Consider the difference between freight or passenger trains and taxi cabs. Freight and passenger trains can move a lot of people and cargo, but they are limited to the rails the trains can ride on. In this case, the rails are a permanent infrastructure, much like many of our dedicated and most important automated tests. By contrast, a taxi cab can be called to appear anywhere and take us anywhere (granted, for a price) and then we can go off and explore as we see fit. Likewise, we can use less permanent automation to do the same thing when we test. Like a taxi cab, we can use this temporary automation to take us to points of interest in the software, and then jump off to follow leads and see where they go. Some of these will be dead ends, and some will help identify key areas and workflows that are important to the success of a project. Be willing to experiment and see which steps will work and get you to where you need to be.

5.4 Exploration is a Mindset

As stated previously, Exploratory Testing does not live outside of automated testing. For many, Exploratory Testing is seen as being a manual process, one in which a tester is actively engaged in looking at the system and considering where to go with each question that they ask. The questioning aspect and the ability to adapt to the information gleaned from the questions asked sits at the core of Exploratory Testing. Automation and Computer Aided Testing can be, and often is, central to getting to the point to successfully explore those avenues.

Much as paddling upstream to explore a river's tributaries requires a well-stocked canoe (and possibly a team of rowers), having a team of automated tasks that allow a tester to get where they want to go inside of an application, and then allowing them to step off and look around, can be tremendously helpful. Again, Exploratory Testing is not a technique. It is a way of thinking about questions and answers. It is a dynamic exchange, with a give and take based on the information received. The approach can be entirely manual, or it can have many automated steps. The key is to know how to get where you want to go to look deeper.

5.5 Utilize Personas to Help Visualize Your Goals

One of the most effective tools I have found when it comes to exploration of an application or a feature is to put myself into the shoes of as many potential customers of our product as I can. At SideReel, that means understanding the interests, desires and ways of interaction of many different groups of people. The interests of a 16-22 year old girl, and the ways that they interact with the devices they use to discover and watch television show and learn about them, are different than the ways middle-aged men might. Also, the shows and the content that appeals to one group often do not appeal to another group.

Relying too heavily on one group of customers may help see what they like or desire in a product, but may also mask what others would like to see. Using personas helps us get inside of these customers word views, and gives us a good understanding of what they effectively want to do and see. Personas are a key ingredient to effective exploration. Not only to they help us frame our questions, they also help us develop entirely new questions based on a variety of contexts.

5.6 Use Session Based Techniques to Keep Things Fresh

Testing and having 100% regression coverage is not something that is going to be achieved overnight. It takes time to examine requirements, develop robust tests that are not flaky, and getting all of the tests to behave in a way that is independent of each other takes time. Also, there is a tendency to grow weary when addressing too many things at once. Developing small charters and executing on them in set periods of time can help focus testing efforts on areas that are most important, and allow for a variety of approaches to examining the product. By keeping our focus on a small sub system, or by using a particular persona, and doing so within a specific context, we can keep our testing fresh, our eyes and minds focused, and we can work on what's most important, which is delivering the best product we can to our customers.

6 Conclusion

For many organizations that have adopted Agile and Test Driven techniques, testing is very much alive and well. It happens at every level of product development and delivery. For many, a high amount of automation exists, and for some, that automation is developing as the project matures. The avenues of exploration are many, and the possible questions to ask are as limitless as the imaginations of the team members that are working on a project. Quality is no longer the end tasks of a group of testers whose mission is to pound quality into a product, it's an integrated series of processes that are owned by and practiced by the entire team.

Test Driven Development focuses on writing correct and clean code. Acceptance Test Driven Development focuses on making sure the functionality being delivered makes good on the goals and promises made to our customers, and that the value they are expecting to receive is present. Automation happens at many levels and in many interactions, at the unit, system and user facing levels. Some tests will be end to end and require little human interactions. Other will be a way to get to a certain destination and allow tester the ability to jump off and have a look around interesting areas. Exploration happens at all of these levels, and is an approach and a mindset, not a methodology. For software to be written in a way that is effective, meets the needs of customers, and has a high degree of quality, all of these components need to work together and be considered.

References

Gärtner, M. (2012) “Acceptance Test Driven Development: By Example”, Addison-Wesley

Savioa, A., (2011), “Test is Dead”, http://www.youtube.com/watch?v=X1jWe5rOu3g, Electronically retrieved on July 24, 2012

Tomlins, M., 2011, "Test Is Dead, Long Live Test", http://mtomlins.blogspot.com/2011/12/test-is-dead-they-say-long-live-test.html, Electronically retrieved on July 24, 2012

Beck, K. (2003). “Test-Driven Development: By Example”, Addison-Wesley.

ArnonA, 2011, "Comparing TDD With ATDD", http://blogs.microsoft.co.il/blogs/arnona/archive/2011/02/14/comparing-tdd-with-atdd.aspx, Electronically retrieved on July 24, 2012

Tchepak, D., 2011, "Why TDD is Hard and What To Do", http://davesquared.net/2011/03/why-learning-tdd-is-hard-and-what-to-do.html, Electronically retrieved on July 24, 2012

Dalke, A., 2009, "Problems With TDD", http://www.dalkescientific.com/writings/diary/archive/2009/12/29/problems_with_tdd.html, Electronically retrieved on July 24, 2012

De Florinier, D., Adzic, G., 2010, “The Secret Ninja Cucumber Scrolls”, http://cuke4ninja.com/download-pdf.php, Electronically retrieved on July 24, 2012

Goldberg, C., 2008, "GUI Automation Considered Harmful", http://coreygoldberg.blogspot.com/2008/09/gui-automated-considered-harmful.html, Electronically retrieved on July 24, 2012

Venkatakrishnan, S., "Problems with GUI Automation Testing", http://developer-in-test.blogspot.com/2008/09/problems-with-gui-automation-testing.html, Electronically retrieved on July 24, 2012

Bach, J., "What is E.T.?", http://www.satisfice.com/articles/what_is_et.shtml, Electronically retrieved on July 24, 2012

Bolton, M., 2010, "Can Exploratory Testing be Automated?" http://www.developsense.com/blog/2010/09/can-exploratory-testing-be-automated/, Electronically retrieved on July 24, 2012

Bach, J., 2006, “Manual Tests cannot Be Automated”, http://www.satisfice.com/blog/archives/58, Electronically retrieved on July 24, 2012

Beizer, B. “Software Testing Techniques”, Second edition. 1990, Intl Thomson Computer Press

Bolton, M, 2010, “Exploratory Testing and Interviews”, http://www.developsense.com/blog/2010/04/exploratory-testing-and-interviews/, Electronically retrieved on July 24, 2012

Crispin, L., 2009, "Beautiful Testing", O'Reilly Press, Excerpt from http://lisacrispin.com/downloads/Beautiful_Testing_ch15.pdf, Electronically retrieved on July 24, 2012

Pier, K., Kaner, C., 2012, "Episode #105: Exploratory vs Automation, Part 1", This Week in Software Testing, Software Test Professionals, http://www.softwaretestpro.com/Item/5604/TWiST-105---Exploratory-vs-Automation-Part-I/podcast, Electronically retrieved on July 24, 2012