the andrea group

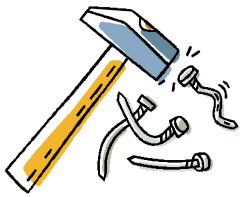
jennitta@theandreagroup.ca www.theandreagroup.ca

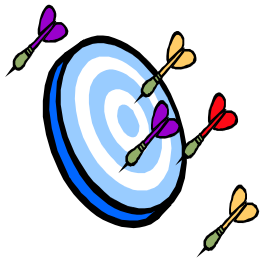
Envisioning the Next Generation FTDD* Tools

***Functional Test Driven Development**

This presentation was made on Tuesday, February 19 as part of the SQE web presentation on Envisioning the Next Generation of Functional Test Tools (http://www.sqe.com/Webseminars/Archive.aspx)

What's wrong with this generation?





Tool Users:
- not picking appropriate tool
- not using tool appropriately

Tool Developers:
- not building what is needed
- feature / tool silos

It's exciting to see automated functional testing tools flourish in the last few years. I can remember starting on my first test driven development project in 2000, and using junit for both functional and unit tests because there wasn't a whole lot of choice at the time. On the next few projects I discovered several other tools that were basically variations on the same junit theme. Then FIT came along and got people thinking outside of the box. We soon had whole new generation of tools that recognize that functional tests are fundamentally different from unit tests. Interesting features and syntax have been popping up all over the place.

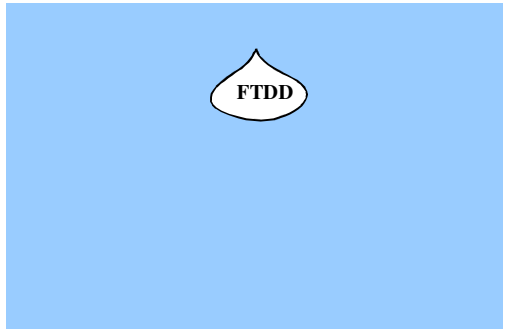
… so what's the problem … why complain about "this generation" ….

Well I personally see teams continue to struggle … their functional tests are not effective requirements specifications … the tests have become a maintenance nightmare and a bottleneck to progress …. And ultimately the tests are being abandoned. This isn't just a tool issue, but often the problems start when a team does not pick the right tool for their context.

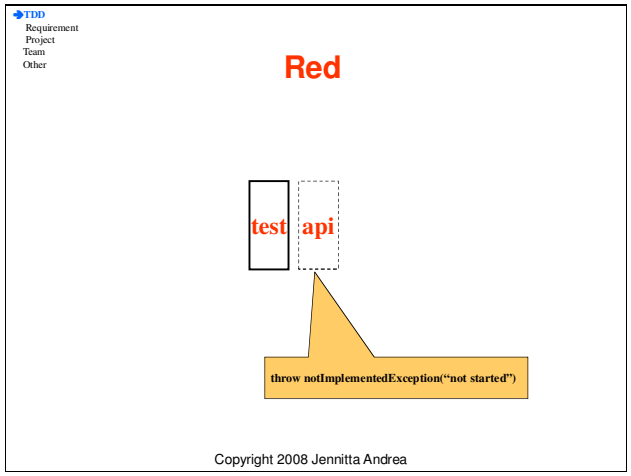
Many different innovations are scattered through various tools. As excited as I am to see these great ideas … it has become frustrating because I wanted them all … in ONE TOOL. The problem we have today is not lack of innovation, but it is in feature silos and lack of integration.

In most cases new 'wiz-bang' features are part of a tool that does not even support the core requirements for functional test driven development. That's why I'd like to take a step back and perform a business process analysis of functional test driven development in the hopes that it will help teams make better tool choices, and help tool vendors to make better tools.

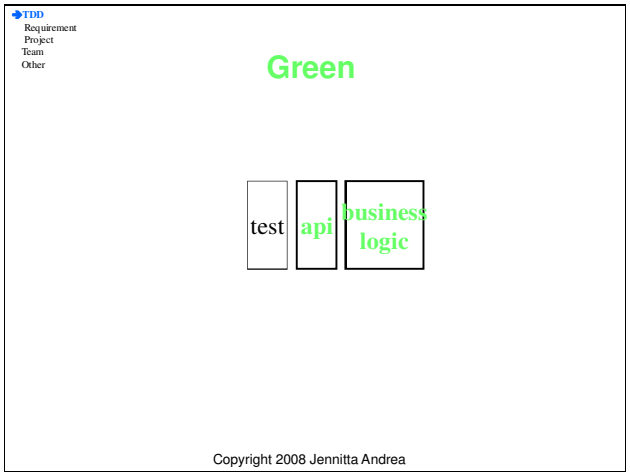
TDD: Red-Green-Refactor



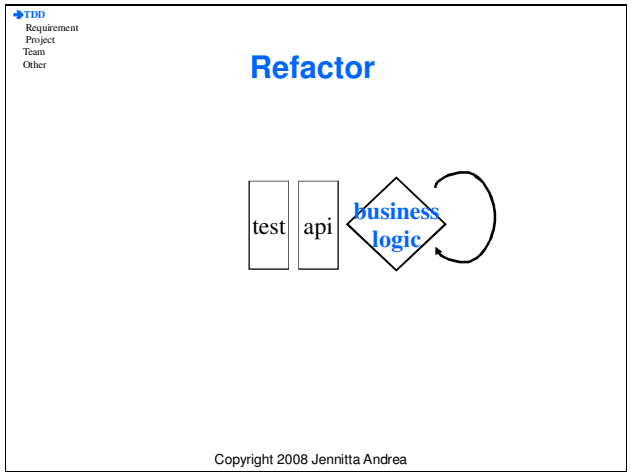
Let's start our business process analysis with a quick review of test-driven-development. This process is referred to as Red-Green-Refactor.



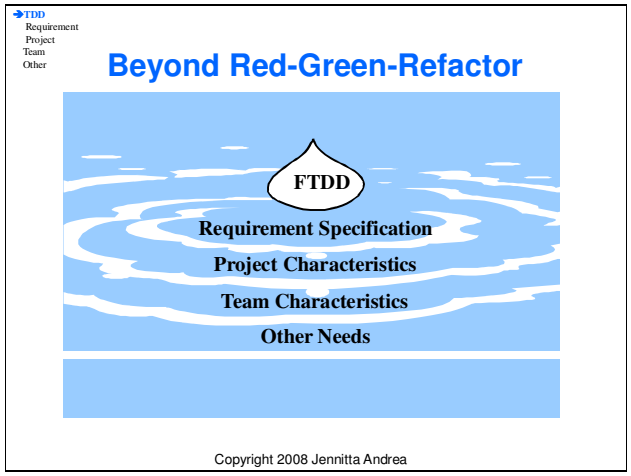
- The Red part of the cycle refers to creating tests that fail intentionally.
- The cycle starts with writing automated tests that describe a new or enhanced capability. While the tests are being developed a minimal skeleton of the API is created to enable tests to compile. Emphasis is on correctness and completeness of the tests.
- Once the tests are finished, they are run to ensure that they fail due to intentionally missing system code.



- The Green part of the cycle refers to getting the tests to pass.
- The focus shifts from the tests to the system code. The API and underlying business logic are implemented just enough to cause the tests to pass.
- While the code is being developed, the tests are run frequently to gauge progress; we're done when the tests are green.

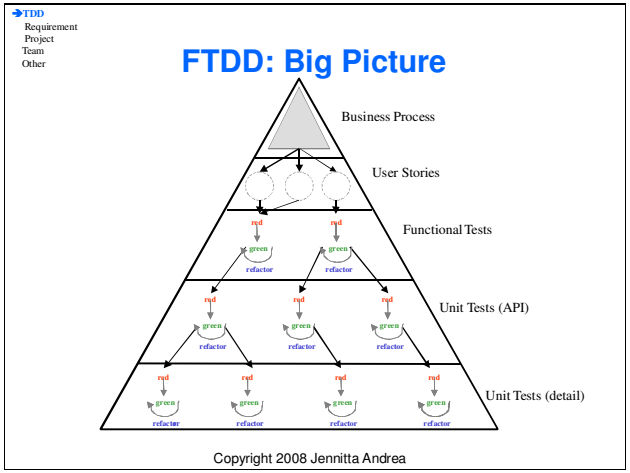


The term refactoring refers to a disciplined step-by-step process of cleaning up the code while preserving the original behavior. Refactoring should occur only when the system is in a known state of stability, such as when all of the tests are passing. The test suite is re-run after every micro-refactoring step to ensure this stability is preserved.



The Red-Green-Refactor cycle is at the heart of FTDD, but teams must go beyond that simple view in order to experience the full potential. Tools must also go beyond Red-Green-Factor, and move toward supporting the full business process:

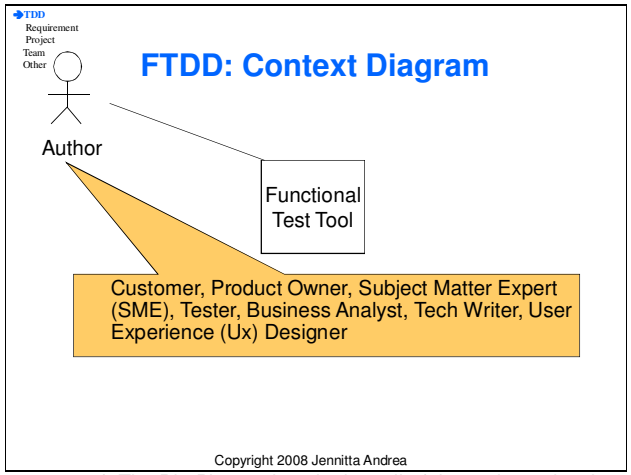
- Functional tests act as concrete examples of business rules, and thus are a key requirements artifact
- Each project and team are unique … one tool will not fit all situations … and on the flip side any-old tool will not necessarily be a good fit for a situation.
- Functional Test Driven Development goes beyond the core development project team. We can make significant contributions to other processes and roles within the big picture.



- Red-Green-Refactor is the TDD heartbeat.
- The full TDD cycle builds of multiple layers of inter-connected tests, developed using the same rhythm, and driven by the desire to support a business process.
- To assist in planning development iterations, the system features that support the business process are decomposed into lightweight descriptions, called user stories. Each story is developed in a test-first fashion using the same red-green-refactor cycle.
- **Red**: Functional tests describe the business process and rules, and generally span the end-to-end workflow of a user goal. The side effect of creating the functional tests is the system API needed to support it.

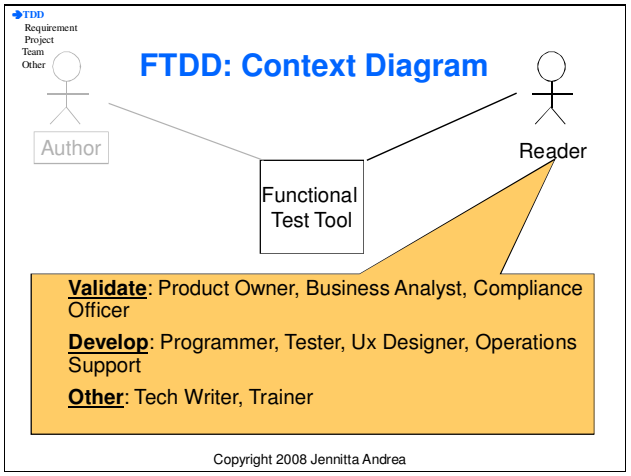
- ***Green:*** Development of the system API starts a chain reaction of unit TDD cycles. The unit tests are typically developed using different testing tools, and have a much finer grain of focus.

Functional tests don't exist in isolation. They are linked to business processes, user stories, unit tests, and ultimately to the underlying system code.

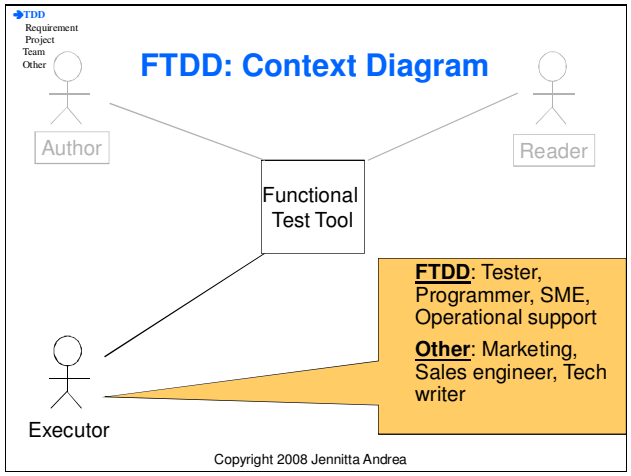


Another aspect of "The Big Picture" is to look at all of the project roles that will interact with the functional test tool. We need to ensure the goals of all of the different roles are met by the tool.

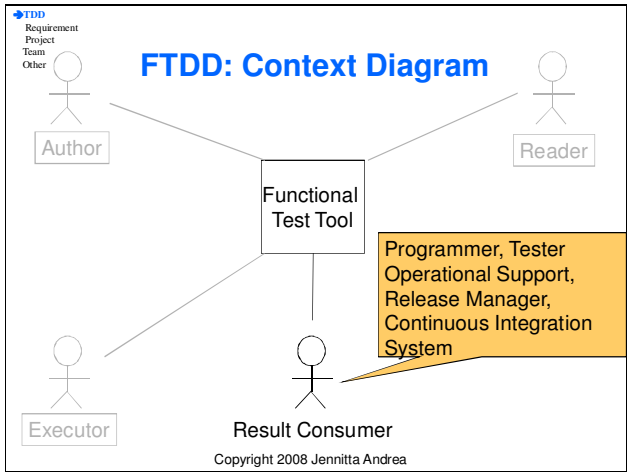
- The author creates concrete examples of the business rules and business workflow in the form of functional tests.
- Descriptions of agile processes often associate this to the "Customer" role. In reality, authoring often happens through collaboration between multiple roles, including: Customer, Product owner, Tester, Subject Matter Expert, Business analyst, Tech writer, User Experience Designer.



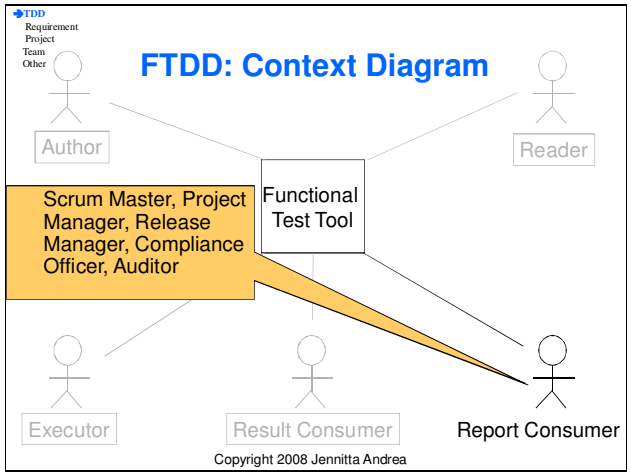
- Because they are specifications (not just "tests") … a wide variety of different roles need to read them. A person must be able to comprehend functional tests quickly to gain a clear, unambiguous understanding of system capabilities.
- Roles with subject matter expertise read the tests to **validate** the requirement specification is correct and complete.
- Development-oriented roles read the tests to **guide their work** (Testers read a test to automate the test; programmers read a test to develop correct system code; operations support people read a test to fix or enhance a specific part of the system.)
- Tech writers and Trainers can use tests as **"examples"** within their supporting deliverables.



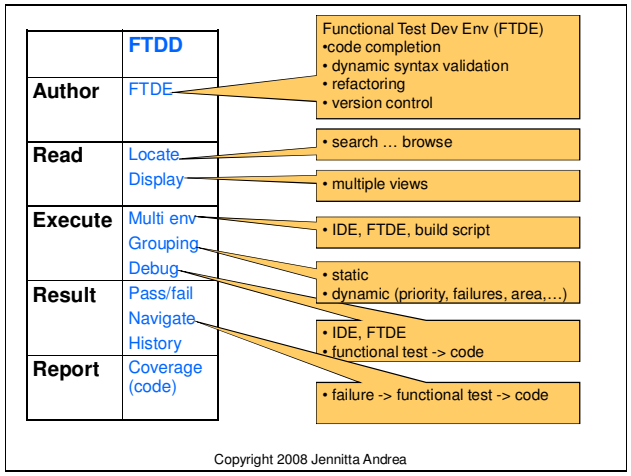
- Many different roles execute the tests at different times, and for different reasons:
- Testers run tests in the functional test development environment as they create the functional test during the "red" part of the cycle
- Developers run tests in their development environment as they create system code during the "green" part of the cycle
- Subject matter experts run tests from a desktop tool (such as a browser) to sign off on functional testing for the system.
  Other roles can use automated functional tests to enhance their work. For example, marketing/sales could run tests before they give a demo to set up demo data, and to ensure the system is stable (avoid the "law of demos" where the system always crashes)



- The Result Consumer is interested in the fine-grained details about whether an individual test passed or failed (and why it failed and where it failed)
- They help answer the questions: am I finished yet, have I broken anything else, it the system stable enough to promote to other testing environments.

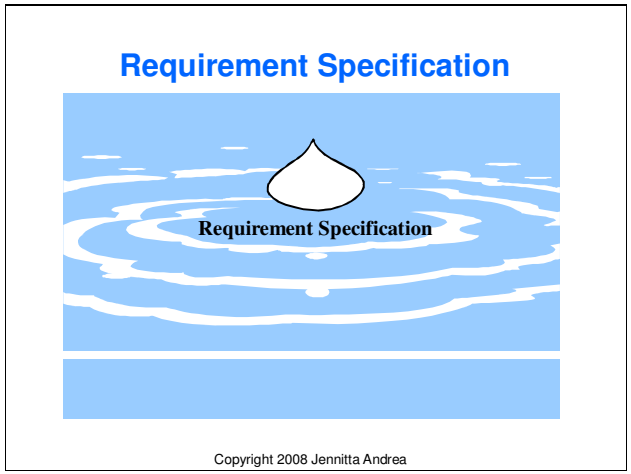


Report Consumers are interested in aggregated test results in order to track progress and ultimately make a go/no-go decision … how many tests were run, how many are passing, have all the defects been fixed, what areas are covered by tests



- There are a lot of **core features** required just to support the simple notion of 'red-green-refactor'. These should be considered **MUST HAVE**. Some highlights ….
- First and foremost, a powerful functional test development environment is essential **(and this seems to be missing from pretty much every tool of the current generation).**
  - At the very minimum, a FTDE must support all of the powerful features provided by software development tools (like IntelliJ and Eclipse) … code completion, intellisense, refactoring.
  - Full support for version control features are also a must-have (the test should not be considered 'binary').
  - The tags for the functional tests must be synchronized with those for the system code.
- Tests must be well organized and easy to find through browsing or searching.
- We must be able to execute the tests from a number different environments (testers = functional test development environment, developers = integrated development environment, automated build = command line).

- We need to be able to arbitrarily group tests into test suites. It is also important to be able to have the tool support dynamic grouping based on things like test priority, last-run status, functional area, etc.
- Full debugging features are required from any environment that the test is run from (integrated development environment, functional test development environment, browser, …)
  Important to be able to navigate from test failures through to functional tests and all the way through to system code



We must always remember that we are creating **requirement specifications**. This notion raises the bar very high for our discipline and approach to developing them.

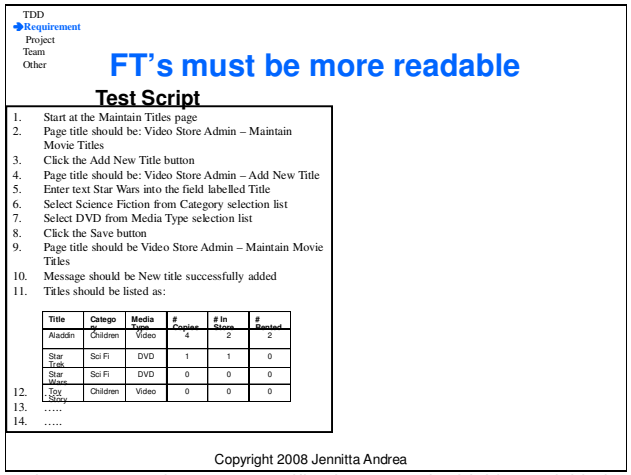
Compared to production code, executable requirements (functional tests) must be

- maintained as long or longer,
- more readable, easier to write,
- more correct,
- more easily and safely maintained, and
- more locatable.

You may find some of these statements rather startling because on Agile projects the focus is to remove all excess waste and deliver working code. So why would we consider functional tests to be more important than production code? The short answer is that the functional tests are the concrete requirements

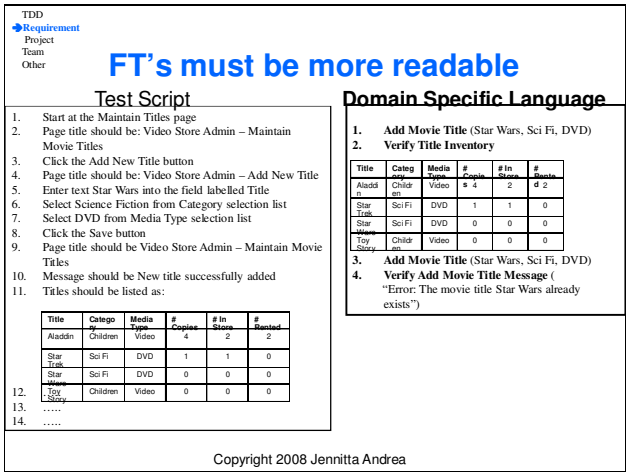
specification that drives our development … if the tests are wrong (or out of date, or difficult to understand) … then our system will be wrong.

I'll touch on a couple of these key points in the remainder of the presentation.



Functional tests as requirements specifications must be highly readable because so many different roles read them. Readability is especially important because non-technical subject matter experts are relied upon to validate correctness and completeness … these specifications will drive what is built in the system.
I've come to realize that the most unfortunate thing about the term test-driven-development is that it has the word TEST in it.
The sample shown on the screen is a traditional detailed functional test script. I wonder how many of you out there can you identify the business rule being expressed here? Would you sign off that it is correct and complete?

… I'd be careful if I were you … It actually has several very serious omissions and flaws … but you can't see the forest for the trees.
Even teams using state-of-the-art tools like FIT and WATIR are still producing specifications like this.



- A test script can be made significantly more readable by creating a Domain Specific (Testing) Language … DSTL (e.g. line 1 is equivalent to line 1 – 8 of the test script).
- The key to producing a good DSTL is to ensure the vocabulary of the specification is declarative (focuses on WHAT, not HOW), and is expressed as business domain goals and real-world objects instead of tactical user interaction steps.
- Although this is a significant improvement, the reader must still piece together the business rule from these high level statements.

TDD
➔Requirement
Project
Team
Other

FT's must be more readable

Test Script

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

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

12. .....
13. .....
14. .....

Domain Specific Language

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

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

3. Add Movie Title (Star Wars, Sci Fi, DVD)
4. Verify Add Movie Title Message ("Error: The movie title Star Wars already exists")

Declarative, Behavior Driven

1. **Given** inventory contains:Star Wars, scifi, DVD
2. **When** add movie title: Star Wars, sci fi, DVD
3. **Then** inventory unchanged
4. **And** message: "Error: The movie title Star Wars already exists")

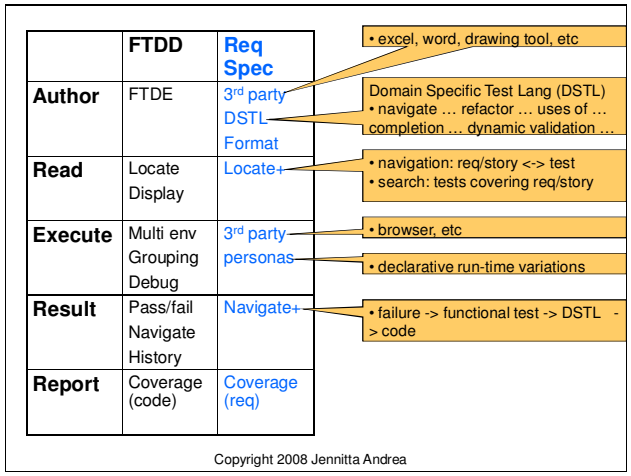
Copyright 2008 Jennitta Andrea

Adding some simple structuring statements to a DSTL helps to express the business rule more clearly.

- The given…when…then style shown here is suggested by Behavior Driven Development.
- Brian Marick's use of the term "example" is also very useful because it helps to remind us to keep the specifications concrete, succinct, and unambiguous.

Specifications like this can be captured in many different formats:

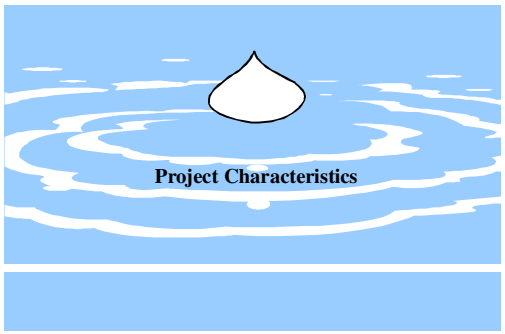
- Tabular
- Textual
- Graphical (many variations of this … including various UML notations.)
- Storyboard of workflow
- Wireframe of UI.



Copyright 2008 Jennitta Andrea

- Again, features to support functional tests as requirement specifications should be considered **MUST HAVE**. Some highlights ….
- Rich support for a Domain Specific Testing Language is essential. This includes having the FTDE command completion, syntax validation, refactoring, and navigation capabilities extended to the DSTL elements.
- The "look and feel" (that is the format and semantics) of the specification language should be accessible to the variety of non-technical participants. These participants should not be forced to use developer-specific tools or languages.
- Remember, functional tests act as the concrete examples within the overall requirement specification. It is necessary to tie these examples into a bigger picture description provided by diagrams, descriptions, workflows, etc, as well as the planning strategy that drive out development (user stories). Navigation and dynamic searching to and from these other artifacts is required.
- Persona modeling helps ensure the system design meets the needs of a wide variety of user types (for example expert vs novice users, or local vs distributed users). The same core test can be executed in a variety of different ways driven by persona-specific characteristics.

Project Characteristics



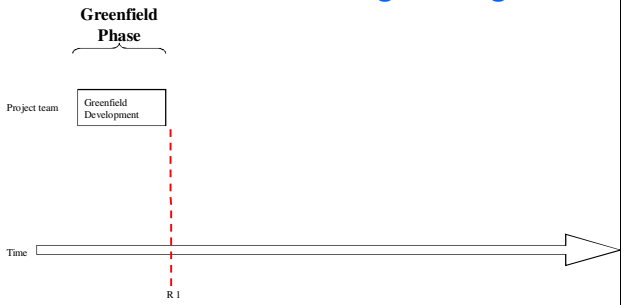
Each project is unique, and the approach to FTDD must be adapted to fit the characteristics of the project:

- Criticality (what happens if something goes wrong … inconvenience vs loss of money vs injury or loss of life)
- Complexity of the domain
- How domain is best expressed (formulas, tabular, graphical, … take your cues from what the subject matter experts draw on the white board when explaining things).
- Characteristics of the system (distributed, concurrent, etc)

In addition, we must apply long-term thinking to our approach to functional test driven development because our functional tests must live as long as the system does! This is a new perspective on things, and has huge implications for tool support.

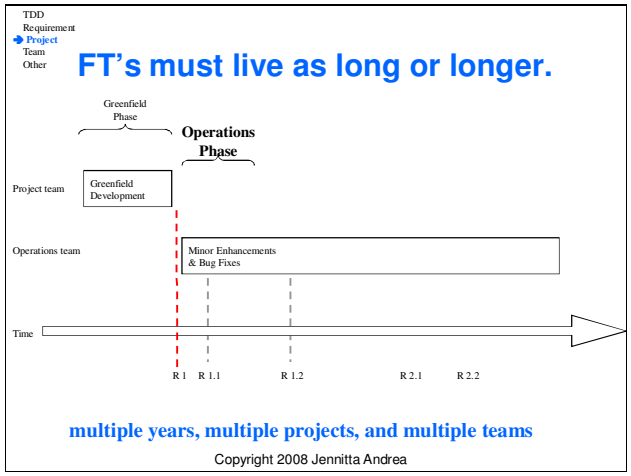
TDD
Requirement
Project
Team
Other

FT's must live as long or longer.

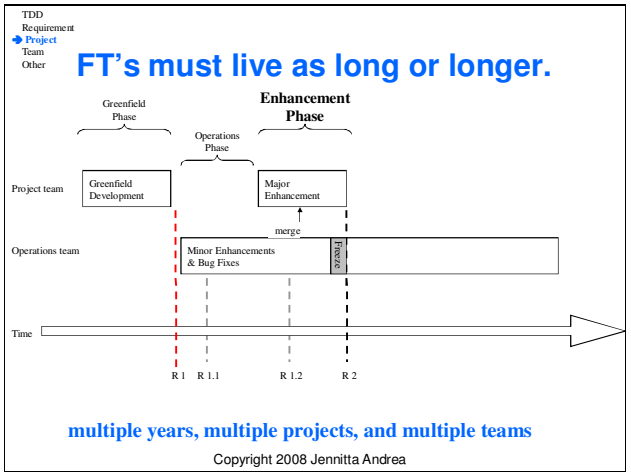


multiple years, multiple projects, and multiple teams

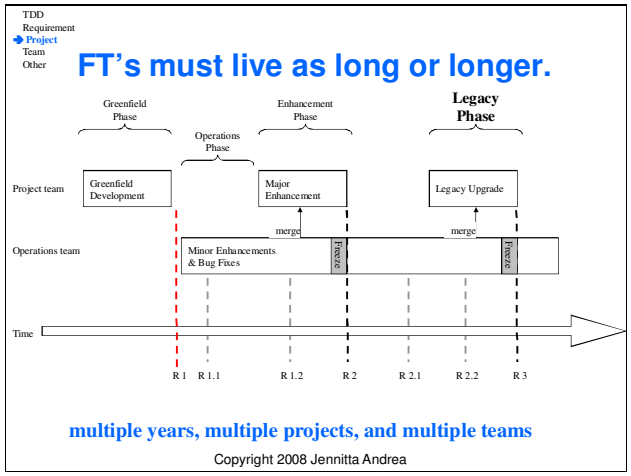
- As I said earlier, functional test driven development goes beyond the red-green-refactor cycle, which is focused on a single piece of functionality. Indeed, functional test driven development is a discipline that affects the whole lifecycle of an application, often spanning multiple years, multiple projects, and multiple teams.
- Let's trace the typical lifecycle of a single application.
- We start with a green-field project developed in a test driven fashion by a dedicated project team.



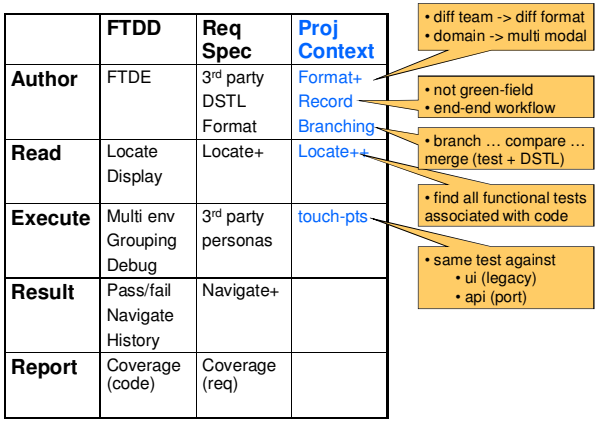
- Once the application is released into production (R
- 1), the responsibility for the code and the automated tests is transitioned to an operations team.
- We expect that the tests will be tweaked when the operations team fixes defects or develop minor enhancements to the system (R1.1, R1.2).
- What happens in reality is that the tests typically 'die' after the hand-off … for many different reasons
    - The operations team is not trained in test driven development
    - The operations team does not buy into test driven development … they don't understand the value, and thus don't run/maintain the tests
    - The tests weren't written with them in mind:
        - The operations team is VERY busy (supporting many different applications), so if it takes too long/too much effort to find all of the related tests, then things will fall through the cracks
        - if it takes too long to read/understand/maintain the tests, then they will take the path of least resistance to fixing a production problem.



- When major enhancements are required that are too large for the operations team to tackle, a new project team is formed.
- The application code is branched into two parallel test-driven-development streams (one for the enhancement and the other for operations). Their work will be merged together prior to the production release (R2).
- Advanced support for merging branched functional tests is critical
    - Firstly, we just need to be able to merge the tests together (some test syntaxes and technologies don't facilitate this very well)
    - More importantly, we need tool support to help people make the right decisions about how to merge the tests together to come out with the intended semantics … we have to make sure our safety net doesn't have any holes in it!

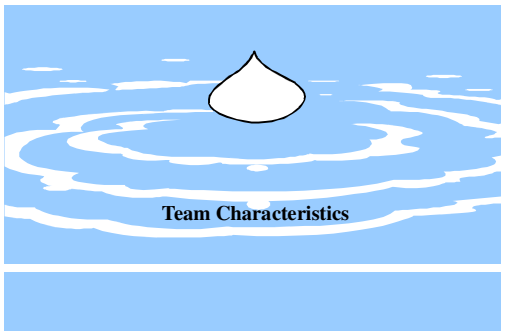


- Over time, the application becomes out-dated (legacy) and may need to be ported to a different technology.
- Again, a separate project team undertakes this major endeavor, performing test-driven porting to build a new application using the existing legacy tests as the specification.
- An effective strategy for test-driven-porting is to have a single functional test run against the old system (as the baseline … this validates the test), and the new system (as validation that we've built the same thing).
- This creates the need for a single test to be able to run against two different tools, technologies, and potentially touch points.



- The multiple handoffs of the tests between different teams over time drive the need for some advanced features.
- These features can be considered to be **NICE TO HAVE**, and certainly don't apply to every situation (but when you need them, you want to have all of the ones you need together in one tool).
- The format of the test should be driven by how to best express the concepts within the domain, and the type of the application.
  - For example, a data-oriented application may be best expressed in a tabular format, while a workflow oriented application may be best expressed using UML activity diagrams and decision trees.
  - Pushing this concept to the limit … some tests may need to be multi-modal … text, graphics, tables all in the same test.
- Making the tests easy to find is key to enabling operations support people to continue to use FTDD in their hectic world. Often they will be working backwards from an area in the system code that is broken. They should be able to find all functional tests that are associated with this area of the code in order take a TDD approach to fixing the problem
- Remember from our earlier statements … functional tests must be easier to maintain than production code. You have to make sure your safety net has high availability, even when the test suite is being refactored or modified. Branching and merging functional tests and the associated DSTL is a very important capability
- To support test-driven porting of legacy systems, we need the ability to run the same test against multiple different touch points, and multiple different applications. This implies run-time configurability of the touch point.

Team Characteristics



Layering on top of the uniqueness of projects is the fact that each team is also unique, consisting of:

- Different levels of domain experience
- Individual style of learning drives preference for format of functional test
- Different mix of roles and skill sets
- Different levels of stability within the team members

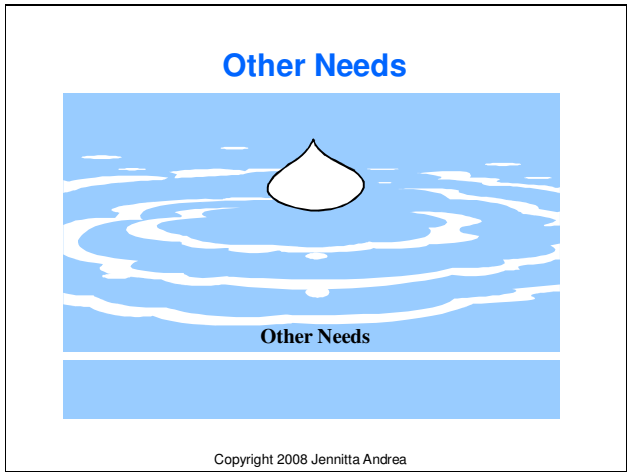
	FTDD	Req Spec	Proj Context	Team Context
Author	FTDE	3rd party DSTL Format	Format+ Record Branching	Format++
Read	Locate Display	Locate+	Locate++	Format++ MultiDetail
Execute	Multi env Grouping Debug	3rd party personas	touch-pts	
Result	Pass/fail Navigate History	Navigate+		
Report	Coverage (code)	Coverage (req)		

user specific
• tabular
• textual
• graphical
• wire frame

Diff fmt than original

Inline DSTL

- Because wide variety of different people will read and write a test over it's lifetime, we need to have a great deal of flexibility and configurability within a tool related to the format of the test.
  - If we separate the format of the test from its content, then the test can be read/written in the format that best suits the person at that point in time.
  - Being able to view the test effectively from a variety of different levels of detail also is important to support user variability (some need the overview, some need the low level details).

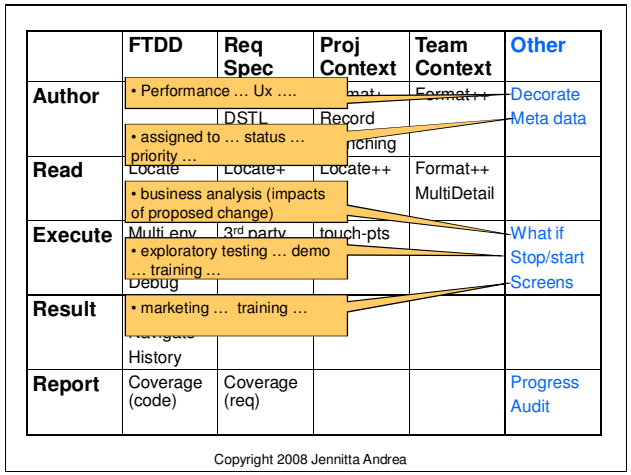


There are many opportunities to enrich the software development process, as well as the business process that the software is intended to support:

- Management
- Training
- Marketing
- etc

We can also adorn the functional tests to support other types of testing:

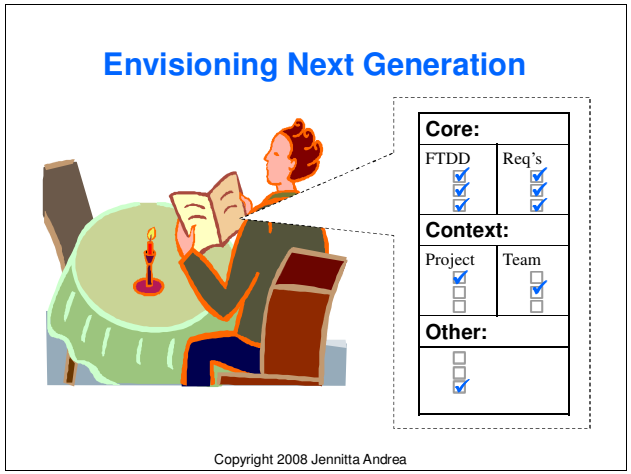
- Exploratory
- Performance
- Usability
- etc



- Functional tests can be decorated with other attributes so that the tests can do double-duty: for example they can also act as performance tests. We will need to be able to specify whether it is being run as a functional test (thus turn on all assertions) or a performance test (thus turn down assertions and turn on the timings … and keep a record of the timings).
- Enabling other roles to run the tests and to start and stop them at will will help to support getting test data set up and getting the system into a particular state. Exploratory testing, demos, training, etc can all benefit from this.

	FTDD	Req Spec	Proj Context	Team Context	Other
Author	FTDE	3rd party DSTL Format	Format+ Record Branching	Format++	Decorate Meta data
Read	Locate Display	Locate+	Locate++	Format++ MultiDetail	
Execute	Multi env Grouping Debug	3rd party personas	touch-pts		What if Stop/start Screens
Result	Pass/fail Navigate History	Navigate+			
Report	Coverage (code)	Coverage (req)			Progress Audit

Here's the full summary just for reference … NOTE: there's still work to do to fill this out (look for workshops at agile conferences and specific workshops like the Agile Alliance Functional Test Tool Workshop)



This vision of the next generation isn't calling for a lot of 'wow' or 'wizbang' features to be sprinkled around. Instead, the message is that we need to take a

deep breath, and possibly step back a little bit so that we can see the big picture of the complex functional test driven development process.

A priority should be to ensure that the essential core capabilities are well supported. Then the next step is to build a coherent mix of features that fully supports a particular project and team context.
There are many ways forward

- enhance existing tools a step at a time
- cooperatively build a single framework that is configurable with plug/play elements
- cooperatively build multiple tools that can work with a single generic test specification
- ….

The key to moving forward is through understanding the big picture and breaking down walls between tools.

the andrea group

jennitta@theandreagroup.ca www.theandreagroup.ca

"Envisioning the Next Generation Functional Testing Tools"	A call to improve the state of the art of functional test driven development by: reflecting on where we are now, describing processes and scenarios within the full application lifecycle, and painting a vision for the next generation functional testing tools(IEEE Software, May 2007)
"Brushing Up On Functional Test Effectiveness"	Explains how to make a functional test into an effective requirements specification by refactoring a test script into a declarative, succinct, autonomous, sufficient, and locatable specification (Better Software, November/December 2005)
"Some Assembly Required: Piecing Together the Truth About TDD"	Teams that naively adopt test driven development frequently run into serious problems. This article applies a magnifying glass to the TDD fine print and suggests paths to safely navigate some key landmines (Better Software, January 2008)
Agile Alliance Functional Test Tool Program (aa-ftt)	http://tech.groups.yahoo.com/group/aa-ftt/ Agile Alliance Functional Test Tool Program (aaftt) yahoo discussion group. The first workshop was held in October 2007. Videos of demos and lightening talks can be found at http://video.google.com/videosearch?q=AAFTT

Here is just a quick reference list to some articles that provide more details about the topics covered within this presentation. There is also a link to the agile alliance functional testing tools workshop resources, from which many of the ideas in this presentation were also drawn.