OpenTestCenter Project

A pragmatic guide to teach QA Engineers how to build their own test and defect management system by documenting both the literal creation of such a system and the reasoning behind the decisions.

Part 1: Database Construction From Testing First Principles

1 Introduction

Hi, I'm Eric Mumford. I'm a QA Manager with 15 years experience in little and big companies. I am writing this series of guides in the OpenTestCenter project ultimately to help me straighten my conceptual grasp of QA, strengthen my tech skills, and increasing my ability to write and express myself. I'm writing in order to ultimately produce a valuable software product that interesting minds contribute to. I'm writing it to network with more people who can't stand poor quality technology, and can't help but teach and mentor and advise their QA teams (and even coworkers) whenever they can (and is appropriate); not because they have to, but because they can't live a life without improving other people while improving themselves. To me, that's real quality. That's good stuff.

2 Background

This part is probably boring and too long. You can skip if if you're not interested in getting to know Eric.

So let me tell you how I came about to writing this, and then I'll tell you what you will get out of it.

So, I've always loved creative writing. I love poetry (Whitman is a kindred spirit of mine,) I love fiction and nonfiction about 10%/90%, and I always wanted to be published. Of course I have spent the last 30 years scared to death of what I might write and what you might think about it, and recently something clicked where it has become more important to write than to be judged (or, self-judged, perhaps?) So that's one piece of how I came to be writing.

How I came to be writing about software: I love taking shit apart. I've blown up motherboards, breadboard circuits, my dad's Sony Trinitron TV speakers, an old CRT TV that I was turning into a dancing soundwave picture show, burned a dark line in my dad's '83 Honda Civic wiring a boombox battery box to the cigarette lighter with very thin lamp cord, and countless other fuck-ups. I'm clearly willing to risk embarrassment and failure in the name of discovery.

As a 10-yr old kid I learned BASIC on the TRS-80 and then the Apple II. I went to computer camp in the summer of 7th grade and learned Assembler. I took computer classes at RPI in the summer of 10th and 11th grade and learned unix, C, C++, LISP, and Pascal. I idolized Steve Wozniak for years (probably still do at a visceral level – have always been too scared to meet him. Not sure why, he's probably just a flight and a phone call for coffee away. Or cake, too, maybe. Who would turn down cake, seriously.) and anyway, as I entered RPI as a full time student I bought a Tahiti Blue 1979 MG Midget (look it up – and then keep in mind I'm 6'6" / 2M tall) and tore the engine down and built it back up new. Upon graduating I continued buying cars and fixing them or modifying them and selling them and moving on to the next adventure. (I've owned over 48 vehicles – buses, motorcycles, cars, trucks, and I built a working paddleboard and two electronically powered cata-rafts with my brother Scottie.) The software testing thing came about during my co-op, where I couldn't believe that I could get paid to break software and then have people fix it and give it back to me to break it again.

So the deeper reason I'm writing about software testing is that there are a lot of sucky books about software testing. I have read a lot of them. Many of them are written by many smart people. But smart people don't always write good books.

Having hired and fired a number of QA engineers and managers, and having reviewed and interviewed more QA Engineering candidates than I can keep track of, I am bothered – distinctly *bothered* – by the lack of QA theory and the spotty arrogance that many candidates have. You wouldn't believe the number of QA Engineering candidates that insist that certain testing is done in certain orders or certain ways – but have no idea why. They can't really relate stress to soak to scale to baseline. They can't draw out a framework for how they think about functional and non-functional testing and how they know how to best structure their time if they only have 10min to put a QA plan together. And, mind you, there's no right answer, I'm only testing for quality thinking and logical reasoning around testing, but the responses are very rarely at the bar for quality themselves.

I blame the rap music.

So who am I to fix everything? I'm not exactly smart – I make a lot of mistakes. Some of them stupid mistakes. (My 6th grade report card said, CARELESS MISTAKES. My mind wanders, sorry. I still have to stare at a restaurant check to figure out the tip. OK double the check total, then move the decimal over, then round down to the nearest dollar, that should be good.) Over the years though, by making mistakes in testing and by having my arrogance kept in check (and believe you me, my ego was huge) by owning up to the mistakes I've made, my QA theory is pretty sound. Still a few holes here and there, and my wording or reasoning isn't always as tight as I'd like it to be, but I think I have something to give to the world there: if only I can organize it into a sensible presentation.

My tech chops are rusty. I have to look up code on google more now than ever. I'm not writing SQL joins often enough, not adjusting stored procedure code often enough, not writing classes or methods often enough. I'm way out of practice on HTML4, and forget about HTML5, I have no idea. I'm out of touch with the differences between IE8 and Firefox whatever and Chrome. I've fallen behind the times and I wanted to build something fun and useful to people.

So anyway, I wanted to build something, maybe related to teaching and writing. The solution to that problem presented itself nicely as a thorn in

our sides for years – HP Freaking Quality Center. FQC. That thing is a high school quarterback juiced full of steroids that is about 10 years past his glory days. And back then it really was hot. But these days, it breaks when you patch it, getting useful support on the phone is like asking a three-year old to bang you on the larynx with a 3/8's ratchet, and using the thing is just slow and tedious. Now don't get me wrong, I'm not dinging HP. HP had to compete and they tried to keep adding features to QC. They put the sales guy in charge. But like Microsoft's decline, they found they couldn't be everything to everyone, and that's what HP's still trying to do. So we QA engineers suffer depending on this thing to manage our testing work for us. The torture is that the product, at its core, IS NOT THAT COMPLICATED. It stores requirements, tests, test results, and defects, and graphs some simple relationships between them and provides rudimentary reporting. I ask myself – why does a software product have to suck at that? Simplest way of answering that is to challenge myself to build my own. So now I knew what I wanted to build.

I like to write, I like software testing, and I knew what I wanted to build. I also wanted the information and instruction to be free to the world, at least this initial piece.

So, putting my background and motivation and problems I saw in the world into a solution design: write a book which teaches you all how to build your own defect and test management system by building my own, keeping track of what I do and where I screw up, commenting about the testing principles along the way that is driving the requirements for the very system we're building together, and then as icing on the cake USE the testing system we are building to test ITSELF as we go so you can then modify and apply your system to any consulting gig or project you take on in your own company.

THAT is cool man! I resonate with that. If you are interviewing for a job with me and you had done what I describe above, and you weren't a total boosch, I'd hire you immediately. You'd be able to converse with me about theory, about actual decisions you made and how they worked out or didn't work out and what you learned, and you'd pass my tech screen because you'd have worked with databases and algorithms and front ends and caching. If you love what you do and are passionate and interesting and have a sense of humor, you're in. It's not hard.

3 Goal

So that's my goal – to fill the world with better QA Engineers by making myself a better QA Engineer. To craft and form QA Engineers who know how to build their own defect and test management systems and embrace a set of principles (what I'll call “QA First Principles”) that drive its design.

So! When you are through with this series of narratives, I promise you that:

- you will have built your own defect management and test management system from open source technology,
- you will have learned QA theory and meta-concepts from me and my 15 years of making mistakes,
- you will maybe be more humble and open-minded to being incorrect or having poor beliefs or assumptions,
- you will be better at diagnosing problems to root causes,
- you will have a mental construct about how to determine what good testing looks like for any piece of software, even if it's not the same as mine,

- you will be hungry for more, and may be even interested in further books I write, and
- you may want to network with me and keep in touch.

Now to me, THAT is worth doing. I could sleep better at night knowing I put that into the world. That is awesome. So that's what this book is for, ultimately for me to feel better about my life. So please, let us begin helping one another today and not tomorrow. So here we go.

4 Assumptions

I'm going to assume you're familiar with Agile development and testing practices. If you're not, it's fine, but review Agile/SCRUM quickly on Google to ensure you know what a Story and Acceptance Criteria are, and what the rationale behind the Fibonacci estimation points is. It should take all of 5min, and I'll be using that paradigm later in this series as a base for how software is being developed.

You should be able to understand the concept of version control such as svn or git. You should be able to install git on whatever platform you are using and be able to create your own repository, and push/pull files from a remote repository (such as the OpenTestCenter repository on github.) If you struggle with this, find a developer friend who can help you.

You will need to find access to a personal computer system that is yours to rebuild as you wish and an operating system you like which you can be administrator on. You should be able to partition your hard drive, update your own operating system, install Linux on a partition or a secondary hard drive on your computer, and generally download and install software. You should be adept at finding solutions to your own questions on google, stack overflow, and other forums.

As we go through building the test management application, my primary machine will be a MacBook Pro. While I will be making references to Apple commands, you should find it straightforward to find the similar commands on Windows or Linux. Later, I will be double-doing all of my steps on those operating systems and updating the documentation accordingly. I do not know how long it will take me to do so; in fact I am almost certain I cannot do it all on my own in a short period of time, and I look forward to leveraging the community with your comments and suggestions as you replicate my work.

5 Understand Fundamentals of Databases

A relational database can be understood at a basic level in order to build one. To build a great database, you should understand it at a deeper level.

I demand that all of my QA Engineers have a basic understanding of relational databases and basic SQL syntax. You have four choices: ignore me, go out on your own and figure out how to learn about relational databases, ask a trusted friend or coworker what you should read, or take my recommendation and get your hands on a copy of C.J.Date's An Introduction to Database Systems and read the first two chapters deeply. (How much of the rest of the book you read is up to you. I would recommend not going too much deeper

until you actually great your own database and have a chance to play around with it yourself.) It's a good book to have in your library.

http://www.amazon.com/gp/offer-listing/0201385902/ref=dp_olp_used?ie=UTF8&condition=used

Go ahead and read those two chapters now. Don't try and proceed unless you have a basic understanding of what a database is, tables, columns, rows, primary keys, foreign keys, and indices.

6 Plan our OpenTestCenter Database Solution

Our plan to build our OpenTestCenter data layer contains five steps, based in a by-and-large way on Ray Dalio's management principles.

1. Understand the purpose and function of our data layer. That is, understand that it will store our test and defect data, and all metadata (data about that data, such as status of bugs, owners of requirements, and how bugs are linked to the tests that found them.) All text above this line has contributed to that understanding. OK, done.
2. Plan your time so that you can read through this guide and create the database with me. You could skip to the head of the class and just run the database build SQL file, but if you are lacking the conceptual framework that went into that file, you've taken your eye off of the real prize.
3. Design a model for what good testing looks like. How do testers test well and how do they think about testing as software is delivered? We cover that in our next section.
4. Build the design as an ER diagram and forward engineer the design into a MySQL database using MySQL Workbench.
5. Manage ourselves to focus on our goal, delivering working software, and not getting too off-track on tangents.

7 Design a Model for Good Testing

Please allow me to begin this section with an anecdote. Trust me, you'll want one. This section is a doozy, and I can't figure out how to whittle it down, or even if I should.

7.1 Anecdote

So, one of my favorite movies is Star Wars. There is a scene that always makes me laugh and which I use in my career. It's the scene where Obi-Wan Kenobi is impressing upon Han Solo the need to “avoid any Imperial entanglements” on the Millennium Falcon's trip to Alderaan. Knowing the ubiquitous presence of the Empire, and the high likelihood of such “entanglements,” Han curls his lip in a half-exasperated expression and replies sarcastically,

> Well that's the real TRICK, isn't it?

Ha! It's one of my favorite lines to use in software test planning because as long as software is developed, there will be a manager that says “I'd really like to find all the bugs before shipping,” or “This effort is complex

and its important we meet time and budget restrictions." Well that's the real TRICK, isn't it?

It's almost like, "Thanks, you've just described my entire job and purpose for being here. Do you wish now to remind me to breathe and drink water to live? Do you want to follow me around and remind me when I open that snack-bag of Doritos that I'm putting no nourishment into my body but rather converting processed cheese-food directly into feces? Because that would be just as helpful as what you just said."

But no, it's best to be kind. But to dispel the viscera, I half-smile and deliver the Han Solo line to them, and they either smile or look confused.

I told you that story to tell you this: if everyone could just read a document and go out into the world and design a model for good testing, that document would be worth gold. This document probably won't enable you to do that; I wish I could write such inspiration. My goal is to get your conceptual framework about thinking about good testing a notch or two closer to fine. Maybe a novice has an insight into pursuing a software testing career after reading this, maybe a 20-year QA veteran connects a relationship that didn't click for him before. All of these are fine outcomes that would bring me joy.

This section is very important. It is the core of my entire effort with OpenTestCenter. This section occasionally goes into deep detail behind certain concepts. I try to keep it as brief as I can but I am certain it can be re-written with greater brevity and acuity. I have taken a long time talking about what I am going to talk about in hopes that the reader does not have unreasonable expectations about what they will walk away with.

We are going to design our database entities from the top down based on how a QA Engineer sees the world. So let's talk about how we'll design it before we design it.

7.2 Thinking about Good Design

Your test data, and therefore your schema which will hold the test and defect data and metadata for the OpenTestCenter application, is an expression of how you hold how these entities are related to one another. I can't tell you how many times engineers and managers prognosticate over what status fields should be, or the need to add more fields, when they have only a weak conceptual notion of what data they really want to measure and how they want to communicate the meaning they derive from that data. Our test management system requires that we think through how good testing is performed.

Your database will be an emergent property of your well-connected conceptual framework of how you think about the inter-relationships amongst the data required to store information about testing and defects.

The ontology, or relationship between entities, should by and large match your mental framework of how you anticipate your actual testing experience unfolding. I will offer a general framework below which is based on my experience in industry. It is my hope that my experience will give you a by and large useful model to start from in terms of the 80/20 of how you probably test software. As you go through the exercise you will likely find areas to tweak to match how you see your team working.

Warning: some of you might find this section very basic or obvious. I would encourage you to read through it all for content and not skip it. You may

find opportunities to give me feedback to hone it better (I'd appreciate it), mistakes I've made, or perhaps areas in which you thought you had a firm opinion and you didn't and now you have a better perspective as a result.

7.3 Visualize yourself in a Movie

I will use a Bridgewater "movie" technique here because I find it personally valuable. Step back and visualize yourself as an actor in a movie of your company delivering great software. The movie is you doing excellent testing in your work environment. See yourself working with yourself, with your team, and with other teams. See how the teams interact. See the sub-teams or larger teams acting as an organism. Let any cynicism or sarcasm or jadedness fade away for a moment. See the reality of what needs to be done in order to accomplish good testing, all things considered, without blame, seeing only the truth of the reality of how things get done where you work.

- Identify procedures and processes that must be carried out, identify who is responsible for them, and visualize the relationships between them and how the people are supposed to work.
- Clearly identify your responsibilities as QA manager/lead/engineer and how you are supposed to carry out those responsibilities.
- Visualize groups as entities.
- Identify people, procedures, note any technology used, dependencies, and how things are managed.

7.4 Identify a MECE list of How Things Work

This is the last subsection of this section. It's also the longest and most important, so heads up!

Build out a MECE (Mutually exclusive, Collectively exhaustive) list of the above. I'll do that in real time with you to both prompt you to think about specific aspects of your movie, and to help you carve out accurate descriptions of what you probably already have visualized in your head but are struggling to get down on paper.

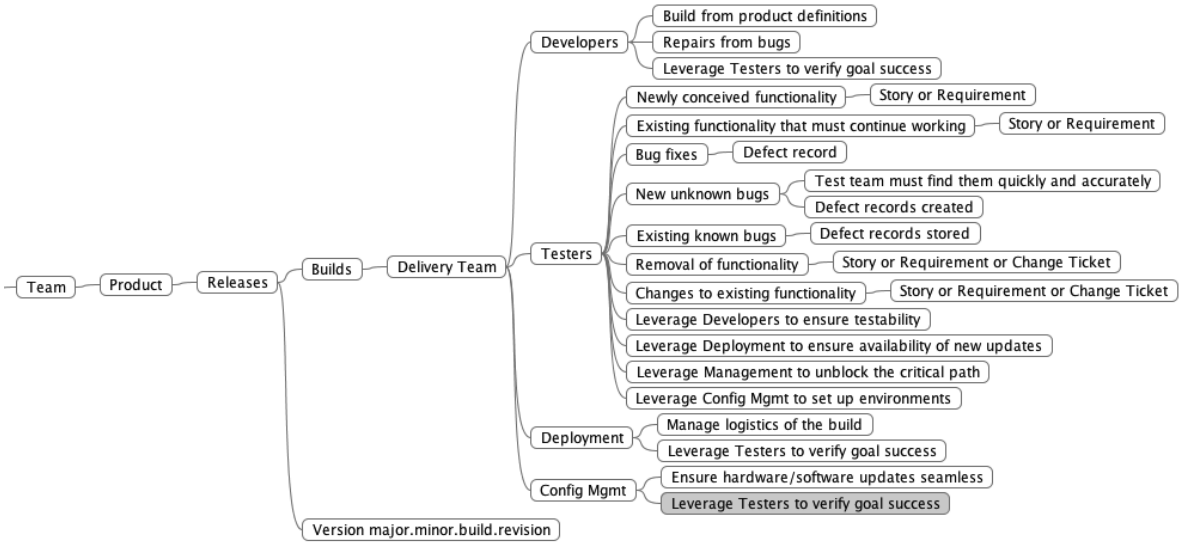
I'll boldface important key words and concepts we will be building up (sometimes quite literally using those exact names in our DB design.)

Here's what NOT to do. Do NOT step back and look at this analytically, specifying how things SHOULD work. Don't begin by saying "Well we need to center everything around Requirements, and Tests, and Defects, so clearly those are the three main database tables, and I will design everything around that." Other open source and commercial tools did that, and the are convoluted and difficult to deal with.

Here's what TO do. Use your knowledge of how human beings FLOW and EXCHANGE WORK in creating software to build out your MECE list. Follow me.

- Start at the very top. There is probably a **Department** and or **Business Unit** you are working in and a **Team** within that unit. That team will work on one or more **Product**s.
- Each Product will probably have a Product Manager that is setting goals for certain functionality to be made available to users on a timeline identified by **Releases.**
- Each iterative delivery of a Release will be delivered to QA in a **Build** with unique identifiers (perhaps *major.minor[.build[.revision]]*)
- Each Build will contain a set of **Existing** and/or **New Functions** that should already work that probably map to **Story Acceptance Criteria** or

Requirements, probably also
- a set of **Changes** to functionality, and
- a set of **Bug Fixes**.



It is probably the developer's and configuration managers responsibility to document these changes and publish them. The QA team validates that this process is good and producing value.

That describes the top-down "what" of everything QA will need to understand. If you are a graphical thinker, you should be able to draw an upside-down pyramid and by and large make sense of how the above flow reasonably follows from Department and Product down to the individual Builds delivered to QA.

So the above description really sets the stage, right, so we can continue in our first-person action play of how testing will happen:

OK, so when QA receives a build with a known Delta Payload, they need to create (either mentally or on paper) a set of "right-sized" test sets which contain a "minimal set of maximum value" tests to exercise the payload. Remember those phrases because they are going to be vital to making you stand out from other QA Engineers who are going to just test everything they can and waste time randomly clicking around to see if they catch something opportunistically. We will talk about that later.

So back to receiving a build. What are your priorities? Your priorities may have a bit of a wiggle from this framework but you will likely find a by and large match.

Your "Step Zero" is to ensure that developers have a reasonable set of Unit tests automatically green lighting the build (and failing unit tests will reject the build back to them without it hitting QA.) Almost no developer practices test-driven development. It's not fun. It takes longer to get results. It's not as easy to experiment. There are hundreds of excuses to not use test-driven development. It takes a short time to save to buy a Hyundai. It takes a bit longer to buy a Honda, a bit longer yet to buy a BMW, and quite some time to buy a Ferrari. Time spent front-loading good design pays off in the final product. There is not right or wrong amount of test-driven development. If you want to ship a Hyundai, skimp on the unit testing and hire a Engineering Manager who can't be bothered with such

things. If you want to ship a Honda or BMW, look closely at unit test coverage and ask honestly and openly (usually with the entire team is best) if their design methods are going to likely result in what you want to ship to your users. Your product manager will be interested in why it will probably take a little longer (or maybe it doesn't have to.) It would be good for you to read up on what test-driven development really is – versus what some people think it is ("Developers doing QA's job" or "QA writing unit tests" is a misconception and not what test-driven development is.)

Our testing world is based around our **Tests**. Our Tests build up our cumulative knowledge of a system designed to be reused by testers that come after us.

Your very first priority as a QA Engineer is

- running the automated **Regression Test Suite** to see if anything that worked before is broken now.

If you don't have one, your first priority is writing one. In writing the regression suite, you'll be forced to dig into the product and talk to product owners about what is important and speak with users to see how they use the product.

That is your first priority because you don't want to start a race with a handicap that you didn't have in the last race. Put in QA terms, you do not want to *pollute your baseline.* Your regression suite is at worst a list of hundreds of tests you need to click through manually, and optimally a bank of automated scripts that passed the last known good build. The regression suite tests the functions that should already work and should always be the first thing kicked off when a build is delivered and should always pass, or *green light*.

Ideally, a regression suite is conceptually composed of two primary parts:

1. a set of unit tests that developers write based on their implementation which execute at build time, and
2. a set of automated tests authored and managed by QA that are written based on QA's understanding of the users' needs and likely use of the product.

It is important that both regression suites be implemented because they test different assumptions.

Unit tests serve to test the algorithmic correctness of the developers' implementation.

An automated QA Regression Suite serves to test the assumptions of the developers who designed and wrote the implementation against QA's understanding of what the user wants or, sometimes more importantly, should want.

Note these are often at different levels of disambiguation. Also note that QA often knows what the user wants without the user having to know. QA may fight for a 2 second SLA between a database call and a result calculation noted in the debug log because they know that that particular calculation is on the critical path to a response to the user. Development may be tempted to shrug off demands to get their algorithm under 4 seconds because they are under the total 5 second SLA for the user response, but QA needs to be able to fight back with cold hard facts.

As you might imagine, a number of interesting corner cases and odd bugs are

far more likely to pop up in test, as opposed to finding them in production when users are dissatisfied, when these two levels of assumptive checks are implemented. Do not make the mistake of having just a set of unit tests or just a set of QA regression tests and presume you can cut corners and not do the other. If you choose to cut this corner, know that you are putting the build quality at risk (this risk may be acceptable at certain stages of a product's lifecycle.)

I'm probably overemphasizing now but another way of putting the importance between unit tests and a QA regression suite is that a regression suite allows sets of eyes from another perspective, another direction of core motivation, to target the particular functionality with a fresh test vector of critical thinking, suspicion, even intuition. QA, after all, is rewarded with breaking the software, putting on the brakes. Developers are rewarded with shipping software, stepping on the gas.

(Did he just say "Fresh test vector"!?) You bet I did. Scrumptious.

Alright, so your second priority once you have verified that the new stuff didn't break the old stuff is not testing new functionality, but

- testing the **Bug Fix Set**. Go through the open bugs, verify the fix, test around the bugs, and set the bug status to Verified or Closed (we'll talk through a reasonable bug fix lifecycle later.)

It is usually more important to determine the correctness and succinctness of developers bug fixes before you test new functionality. Here is my reasoning, which you can agree with or disagree with.

A developer is usually going to be motivated to fix bugs they caused quickly. She is going to have a vested interest in knowing whether the fix broke anything else – developers often rely on QA for this and do not test around their fix (in some theaters of team environments this is acceptable practice, in others less so.) Human memory has a steep drop off. A developer will soon forget how he or she fixed the particular bug if you let too much time go by. *You will often have less than a day to get bug fixes tested before the cost of context switching the developer back to return to an old bug becomes very high.*

Context switching is extraordinarily expensive. I wish I had studies to quote numbers; I do not. I am using my experience having been on both sides of this issue (taking too long to test, and working with developers who are painfully waiting on QA.) Developers will have to mentally pause their momentum, switch back to remembering how they fixed the bug, open that branch, get back into the code, solve the problem, implement the solution, and then switch back to their original work and try to pick up their momentum again. It is tedious to everybody if QA cannot test bug fixes fast.

If you are doing the math, you are probably wide-eyed thinking, well if this has to be done fast, and regression testing is the most important thing to get done first, then it seems pretty important to have an accurate, fast regression suite targeted at the high value areas of the system! You've just formulated the exact expected value equation of automating regression tests versus leaving them as manual tests "because automation is too expensive/hard."

Trust QA Engineers that trust existing regression suites... but verify. As a QA Manager I have been guilty of being too quick to say "I'm going to assume this regression suite is a pile of garbage until I can see it proven to cover what it needs to." Really, that statement says the same thing as "Trust but

verify," but it sets a negative tone and can pull the energy of the team down. So take whatever tack you feel is best here, but by all means trust only in the truth you can verify yourself and don't take anyone's word that tests you didn't write are supposedly providing some sort of coverage. The more emotionally charged a tester insists that tests provide coverage, the *less likely* that coverage exists in reality. The more cool and calm a tester is willing to prove out his automation, the *more likely* that coverage exists in reality. Be careful about culture barriers here. I personally made the mistake of confusing engineers from India communicating in high tempo, higher volume ways as emotionally charged (because when I'm emotionally charged, I talk faster and louder.) The truth of the matter was, they were not reacting emotionally, just talking faster and louder. It didn't mean anything, but I didn't realize that yet. It sometimes takes a few months to really get to know people's nuances of communicating, especially when they come from other cultures. Good QA Managers are patient and persistent with this, and put a solid few months into evaluating their team when they first come on board.

The smartest test engineers will tell you explicitly, "I wrote these. I think they work and can show you very easily. Absolutely do not trust them. Work them through and find holes." Smart developers will communicate about their bug fixes similarly. "I fixed this. Break it." The smartest engineers have little to no ego. They have no ego because they know that if they are always looking for problems, they write the best code.

In conclusion, trusting automated regression suites without understanding what they cover *will always fail you eventually* and let a bug through to production when you get lazy. Trusting without verifying is a time-bomb decision. Do yourself, your QA team, your project team, your company, and your career a favor and work personally to ensure an accurate, well-targeted, well-maintained (targets always move and priorities always shift!), efficient regression suite in place.

Stay limber, stay open-minded, keep your eye on the prize: ensuring quality of the system as targets move and priorities shift. Adjust, adjust, adjust. In systems theory, the most flexible element of a system is the one in control. Always have QA be in control – it's the right place for QA to be, but you have to earn it. That is within your grasp, all systems require only a requisite ability to learn and persistence to get into the underlying technology to build an understanding.

OK, so your third priority after the automated unit tests green light the build, you've run the automated regression suite, and you've manually gone through and verified bug fixes is finally to play with the exciting shiny stuff:

- test the **New Functionality** in the build. Ideally you have already worked with the product manager and release manager to know ahead of time what new functionality was going to be included so that you could perform an analysis of the new stuff and figure out what tests to write.

Here is one way to think well about testing. Almost all QA Engineering candidates make a real mess of this question when I ask them what their thought process or mental framework is for ensuring they don't miss anything when they are testing something new.

At the top you have the functionality being implemented, suppose it's a 5-point story with a few **Acceptance Criteria**. The acceptance criteria must be independent from one another and testable / measurable. (Sometimes stories

that don't involve a front-end change or are purely performance increases require developers to put in a DEBUG writes to the application log, such as time spent in a certain function, to make behind the scenes acceptance criteria visible to QA. Kindly remind them of this when they balk about measurability. If you consistently enforce mandatory measurement as a part of doneness, developers will stop complaining and just get used to writing DEBUG log writes in their functions because they know QA will be looking there and they are not wasting their time. Developers hate wasting their time, so don't waste their time.)

For each Story's acceptance criteria, run down this tree:

A) How can you test this thing positively and negatively (think of reasonable and edge case/crazy ways to break it to see how it behaves)? I think of this as a **Test Mode**. Brainstorm these cases into your test tool and mark them **positive** or **negative**. This is useful because it will produce test metrics that can be very interesting down the road. Indicating positive or negative is a piece of metadata that you will want to capture to show your product manager about a month or two after you are testing. It will begin to show a momentum or heat map in the product in terms of what is standing up to what sorts of testing. You'll see, just bear with me. You can always stop doing it later if you're not satisfied.

B) For every A, is the business impact deep? Is the usage of that scenario heavy? If so, is it valuable and cost-effective to automate it? If so, mark it a **priority** 1 and *automate it directly into the automated regression suite* (creating a branch for this automation build in QA's source control.) If yes and automation is too complex/impossible/costly, *escalate this to the QA Manager* and mark the test a priority 1 and write a manual test, linked to the story's acceptance criteria, to be included in a smoketest. If no, write a manual test and mark the test priority a 2 to be included in a regression test. If it's an edge condition, mark the priority a 3 to be included in a full system test. You might be starting to visualize QA metrics down the road that use test priority as a way of generating a heat map for a product manager. You're right, they will. They will also serve you – you will be able to very quickly pull queries from your test tool as to what tests should be a part of a smoke test, a regression suite, and a full functional test. You will want these three different test options as a sort of menu to offer your product manager and engineering team when they ask how long it will take to test a certain part of the application if they make a change. You wouldn't believe how many QA engineers use their intuition on this, and are wrong, or untrusted because they often finish way too late or way too early. *Using a test priority rating is a way for QA to uphold their responsibility of knowing their own test bed and how it maps to the different nuances of the system under test. It is not a nice to have; it is a responsible tool that a QA engineer puts in her pocket for use whenever needed.* I will prove this to you later as we build the tool and we begin to build up some test data and some test metrics that demonstrate this in real life. You won't have to take my word for it because you'll see it with your own eyes.

C) Think about how this story acceptance criteria should behave in terms of a response time or SLA. A service-level agreement is both a legal term (not what I mean here) and a metaphor (what I mean here) of the agreement the engineering team has with the user, or even within the engineering team, as to a reasonable and acceptable speed at which the

thing will do what it needs to do.) These tests are often called "Performance Testing" but I don't like to use that phrase because it's ambiguous. Instead, I like to use the specific test type names below to indicate which kind of performance testing you are doing because the *goals* of the testing are different. It also expresses confidence that you know what you're doing and why rather than randomly probing a field for mines.

1. **Baseline**: How fast should the thing finish under reasonable conditions
2. **Soak**: (Duration at expected load.) How fast should the thing operate under reasonable conditions over a long period of time (think smartly about what long period means. It could mean 10 seconds or six hours. Try to keep the median value here under six hours so that your total test turnaround for a build is within a work day.)
3. **Scale**: (Duration at stretch load.) If the demand for this module goes up 2x or 3x, or the user load goes up 2x or 3x (often not the same thing, a user load increase of 2x could mean a module load increase of 20x in some cases such as cache refreshes on busy web pages) what happens to the SLA of the module? Often engineering will ask that you (QA) find out the reality before they specify what reasonable is.
4. **Stress**: (As long as is required at ever increasing load until system breakage.) If you increase the transaction load against this module or module set in a step function to infinite load, what part of the system breaks and in what way?

D) How well do you know the product? Unless you have more than 6-12months testing a product, have your tests **Peer Reviewed**. Add metadata to the test that indicates who reviewed your tests for accuracy, value, and completeness, and when. When your skill and confidence builds around a product, you can hide this metadata. It's useful sometimes, especially for new products or new engineers on the team.

8 Install MySQL and connect Workbench

Never thought we'd get here, eh? After 13 pages of reflection, we're installing a database and creating something real. Come on, that's not too bad, right?

8.1 We'll need a database to store and structure information about what we're testing and how its going. A reasonably popular one seems to be MySQL, so let's pick that and install the latest version on your box (at the time of writing I'm using MySQL 5.5.27)

8.2 Set up the admin account and database manager account.

8.3 Ensure that you've set up MySQL Workbench product. It may have been bundled with your download.

8.4 Check to make sure MySQL is running on your box (I was toying with other installs and fidgeting with updates and had rebooted while writing this, and MySQL for whatever reason had not restarted upon reboot.)

8.5 Install MySQL Workbench if it wasn't installed with your MySQL installation

8.6 Select New Server Instance

8.7 Name your Connection something simple, I will use "TestDS"

8.8 Leave all the defaults alone for now. MySQL Workbench should be able to create its connection to your MySQL instance by prompting you for a password to "root". Enter in the password you used when you installed it. It will say,

Connecting to MySQL server localhost...
Connected.
MySQL server version is 5.5.27
MySQL server architecture is i386
MySQL server OS is osx10.6

8.9 Continue through to specify the installation type for your target OS. Finally, name the instance profile (I would just leave the default and click through.)

8.10 Workbench is now set up with a connection to your MySQL database server.

9 Begin your ER Diagram by creating Tables

An ER diagram is an entity relationship diagram, which you know because you read the first two chapters of Date's book like I suggested. While an ER diagram has specific symbology, we will be using only the most basic concepts of tables and foreign keys.

Create a new EER model using Workbench. This will create the database entities visually, and it will draw out relationships and dependencies for you as you specify the Table structure. It's cool, you'll like it.

We will create entities -- Tables -- based on our discussion above. All Tables must have unique names within the Schema. You should find that statement intuitive if you've read Date.

9.1 Click Add Diagram

9.2 Click the Table icon or press "T". Place a Table into the left hand side of the diagram. Look at you, you're making Tables! Hot-dog!

9.3 Type in Department for the Table name. There, you have an empty table called Department. Let's keep going and just build out a group of tables first.

When we create tables I will specify them in all caps with 10 equal signs underlining them to review what we created.

```
DEPARTMENT
==========
(empty)  # Note, I'll make comments after hashtags
```

Begin to build a data schema based on our conversation thus far. Begin at the top and work down – conceptually and literally. Start in the upper left of the Workbench document and create table boxes for BusinessUnit, Team, Product. Those are the largest containers that map where in the organization the work is being done.

```
BUSINESSUNIT
==========
(empty)


TEAM
==========
(empty)


PRODUCT
```

==========
(empty)

Then under product, create tables Release, Build, Functionality. Then create Payload, and make two columns under Payload, containing ChangeSet, Story, AcceptanceCriterion, BugFix, and Bug. Don't worry too much about the exact way the information will connect, we'll be crafting that soon.

RELEASE
==========
(empty)

BUILD
==========
(empty)

FUNCTIONALITY
==========
(empty)

PAYLOAD
==========
(empty)

CHANGESET
==========
(empty)

STORY
==========
(empty)

ACCEPTANCECRITERION
==========
(empty)

BUGFIX
==========
(empty)

BUG
==========
(empty)

Finally, create the Test table.

TEST
==========

(empty)

Note that I have chosen to label all table names in the singular. This is common in some industries and not in others. You may call your tables anything you wish using whatever format or nomenclature you like. My goal here is to give you a baseline understanding of how a test and defect management schema can be be built in a way that can probably represent 80% of all testing projects out there. You can then carve out a schema that suits your particular needs.

Whatever method you choose, I only ask that you be consistent with it throughout your schema.

10 Create Columns and Restrict Values with Foreign Keys

Table columns define the shape and size of the data that a table may contain and depend on. Table column names must be unique within the Table. Foreign Key (FK) names must be unique within the entire schema. It is a common error to lost track of what you named a foreign key that referenced a UserID 18 tables ago, so it is good practice to use a consistent paradigm when specifying the name of FK.

10.1 Workbench is cool in that it fills in details and makes assumptions for you as you begin to define what columns a table should have. For example, create the column and Primary Key (PK) for our Test table, *TestID*. Workbench assumes the first column you create for a new table is both the PK and therefore Non Nullable (NN), and in this (and most following) case(s) both of those assumptions are correct.

```
TEST
==========
TestID      (INT) PK NN
```

So, why not simply name the column, "ID"? Is "TestID" not redundant, as the table name clearly says "TEST"? Play it forward: later on we would end up with a lot of tables with the column "ID" and it might get confusing as we are debugging SQL statements to figure out which ID belongs to which table (your eyes begin to glaze over when you stare at SQL too long and that's when mistakes can happen.) So its best to be as specific – and wisely brief* – as you can. Let's stick with "TestID". Assign it to be an INT data type, as all of our IDs will be. Workbench will automatically assume it is the Primary Key (PK) and is Non-Null (NN) – meaning, the database will not accept a new or modified (updated) entry in this row unless that column has a specified value. Further, click the UQ box. This forces the value of the TestID to be unique. These are handy constraints to be able to specify. This is probably intuitive to most of you; for the rest of you, keep reading and it will probably become obvious why this is wise.

```
TEST
==========
TestID      (INT) PK NN UQ
```

*Rant: What I mean by "wisely brief" is that I have noticed, especially with engineers from India but perhaps with engineers elsewhere also,

that there is an almost obsessive compulsion to abbreviate as much as possible-- even when such indulgence makes it difficult or impossible for other people following you to inherit your work. For example, Test cases are TCs. Requirements are RQs. Emails are Mails (Is the E that much of a savings!?) Quality Center becomes QC, Clearquest becomes CQ, and before you know it you have engineers rattling on about the TCs in the QC linked to RQs in the CQ that aren't showing in QC and project managers are pulling their hair out trying to keep up (that example is a real example by the way.) STOP THE MADNESS. This may be a cultural phenomenon or it may be my oversensitivity to mindless attempts at time saving without thought given to common sense. So, let's endeavor to be brief, but wisely so.

10.2 We spoke about the intelligence we could build later in test metrics if we could discern how much positive and negative testing were finding more defects. In order to calculate that information later, we need to provide an outside container for that data and then relate Test rows to that container.

Create a new table called TestMode. Create a column TestModeID, type INT, PK, NN. Create a column Name, type VARCHAR(45). Create a column Description if you'd like, type VARCHAR(255). This will allow us to create whatever kinds of testing modes we want and map those modes to our tests through the creation of FKs.

```
TESTMODE
==========
TestModeID  (INT) PK NN UQ
Name        (VARCHAR45) NN    # Force a name to be entered
Description (VARHCHAR255)     # Optional, possibly superfluous
```

10.3 Create the FK relationship between TEST and TESTMODE tables:

Let's return now to the Test table, where we will connect the container of shaping data in TESTMODE (I say shaping because the data in TestMode, eg “Positive” or “Negative”, will provide a shape to the row in the Test table) to our growing definition of how we shape a Test, or said more succinctly, a row of TEST data.

First, create a column in TEST to hold the chosen test mode value. TestModeID will suffice. Remember, even though we used that same column name in another table TESTMODE, column names must only be unique within the table, not within the database system as FKs must be.

```
TEST
==========
TestID      (INT) PK NN UQ
TestModeID  (INT) UQ          # No reason to make this NN
```

Click on the Foreign Keys tab of the Test- Table tab in Workbench. Click on the “click to edit” text to create a FK. There are three basic steps to the initial definition of a FK.

Think the name of the FK through smartly. I mentioned earlier that you want to use unique names for FKs, so you want to name them in a way that you are confident you did not use that name before.

There are several ways to accomplish this. I will explain my way, which may not be the best way, and use my way consistently through this project.

Name the FK "TestMode_for_Test". Shaping table, the word "for" separated by underscores, and the Table containing the FK. It reads sensibly and promises to be reasonably unique.

The second part of identifying the FK is the Referenced Table. Select "TestMode".

The third and final part of identifying the FK is the details of what column the FK should map to in the foreign table. Choose TestModeID to TestModeID. Fini!

```
TEST
==========
TestID      (INT) PK NN UQ
TestModeID  (INT) UQ FK=TestMode_for_Test->TESTMODE.TestModeID
                                                  # Col.Row
```

10.4 Create a table PRIORITY.

Now that we've seen how TESTMODE will be able to supply a controlled metadata set for the Positive, Negative, Edge, and any other modes of testing you feel you need to track, let's do the same for Priority, TestType, and Step. Each test will have those as mandatory elements.

Create a table PRIORITY. For whatever Priority values you want, you'll need that unique ID again which we'll call PriorityID (INT), and give it a Name (VARHCHAR(45)) and a Value (INT).

```
PRIORITY
==========
PriorityID  (INT) PK NN UQ
Name        (VARCHAR45) # Give it a name if you wish, or just call
                        # it by its value
Value       (INT)       # Numeric value enables future quant metrics
```

A note about using both a Name and Value in addition to a unique ID for Priority: With this configuration, you can prioritize tests as 1, 2, 3, or as High, Medium, Low, or any combination of words and numbers. I'm going to suggest a simple 1, 2, 3 to begin my framework. Your framework may be more simple and more clever and I encourage you to do what you think is right for you.

Note that PRIORITY has no foreign keys, because its definitions are independent of others, like the noble Honey-Badger. Priority table don't care. Note however that updating any existing rows will change possibly hundreds or thousands of tests referencing that priority ID. This is the responsibility of the application layer to honor these contracts and ensure the user really wants to make such sweeping changes (and in fact there are reasonable scenarios in which they may.)

We'll certainly create a row in TEST for priority. It's not the only place we'll do this for a test, though. Think about smoke-testing, regression testing, performance testing – with all the different test

sets that this test might fall into, ought we preserve the freedom for the tester to assign a different priority to a test within those test sets? I can imagine scenarios where that freedom would be useful.

So let's build the FK relationship but remember we're not done with mapping Priority and that we'll see this table again.

```
TEST
==========
TestID      (INT) PK NN UQ
TestModeID  (INT) UQ FK=TestMode_for_Test->TESTMODE.TestModeID
PriorityID  (INT) FK=Priority_for_Test->PRIORITY.PriorityID
```

10.5 Create a table PASSVALUE.

You may know this table as the artist previously known as State, or Execution State, or Status, or... I can't keep track of those words that seem to start to mean the same thing. So what I prefer to do is keep track of the Pass or Fail execution value of a test is to just call it a Pass Value.

Pass Value generally has two purposes in the real world. I want to know what the test did last time, and I want to know how it's doing now. Status, State, those are too meta concepts for what we want to know, so let's simplify and agree to call it a PassValue. This way you can come up with your own Pass Values. Failed, Passed, Not Run are probably the base set. You might come up with others you need.

```
PASSVALUE
==========
PassValueID (INT) PK NN UQ
Name        (VARCHAR45)       # Passed, Failed, No Run etc.
```

10.6 Create a table STEP.

Tests contain test steps as their basic anatomy. STEP must contain our unique ID, StepID (INT). Each step has an order it needs to be followed in, so we'll account for that with a LocalOrderID (INT). We'll need a test Condition (VARCHAR), provide for an Expected Value (VARCHAR), a container to hold its CurrentPassValue (INT), and a container to hold its LastPassValue (INT). Think of these values as convenient information we'll want to hold about the particular steps of all tests for “As is” or “Snapshot” reporting.

Build-level or Release-Level reporting will be aggregated from tables we have yet to create that will contain cached aggregations.

Note I'm making Expected Value nullable. While QA philosophy demands that each condition carry with it an expected value, the reality of “street testing” is that most testers are so damn busy and pressed to the edge that they are looking for any way to work more efficiently. If they can name a test step in 4 words or less and skip the expected value (because it's intuitive what should happen), they'll leave the field blank. So, don't force your philosophy on people, allow testers to optimize on their own. Create a flexible schema that provides for both scenarios.

```
STEP
==========
StepID      (INT) PK NN UQ    # Systemwide unique identifier
LocalOrderID(INT) NN          # Must have an order 1,2,3,...
Condition   (VARHCHAR255) NN  # Must enter a name for the step
ExpValue    (VARCHAR255)      # Nullable as per above
CurrentPassValue  (INT)
LastPassValue     (INT)FK=Last_PassValue_of_Step->PASSVALUE.PassValueID
ParentTest        (INT) NN FK=Test_which_step_belongs_to->TEST.TestID
                            # A test step may not be orphaned
```

10.7 Build out test execution schema.

Let's think about how we'll execute tests.

Earlier, we began talking about testing by reflecting on the importance of a Regression Suite. If you recall, we agreed that a regression suite at its conceptual level is a group of (hopefully not too many) manual and (hopefully mostly) automated tests that we've determined are useful to run as soon as we've received a new build (or a change of some sort is introduced, such as a payload delta consisting of purely bug fixes.)

At the highest level, and this is the right level to think at when designing a data schema, any test suite is a group of tests that (in theory) already exist because you were notified by development that this work was going on and you had time and foresight to author tests before getting the payload. You have the tests selected by priority and included in a set, which has an execution order within the set.

A tester will need a mechanism to map tests to BOTH test suites, and then a mechanism to map suites to the particular container (which we will call sets) of suites that we wish to manage (in this case Regression).

Suites are nothing more than the “last known good” set of tests that comprised a major test pass, such as a smoke test or full functional test. Sets are the actual groups of tests that get executed – in reality, right, you test a smoketest “plus” a few other tests or “minus” a group of tests you have high confidence will pass because the payload doesn't touch that group of functionality.

Separating suites from sets saves the tester time. It allows the tester to choose from a palette of suites, while at the same time preserving the history of what the tester actually tested. A good application layer will allow the tester easy access to these diffs, and a good data layer will provide the flexibility to store this data without forcing the tester to write complicated joins against the back end. We are designing such flexibility here.

Do you see the value of defining suites and sets as separate and distinct from one another? If you have a logically-connected argument that this is wasteful and why, please contact me. If it just flusters you, have a beer and continue on.

So first let's create the mapping mechanism that allows us to group tests into a suite. Create a table TESTSUITEMAP.

```
TESTSUITEMAP
==========
SuiteID     (INT) PK NN FK=Suite_containing_Test->SUITE.SuiteID
TestID      (INT) FK=Test_in_Suite->TEST.TestID
TestPriorityInSuite (INT) FK=Priority_in_Suite->PRIORITY.PriorityID
```

Through this table, many tests can be mapped to many suites, each with a different priority for that Suite-Test relationship. Querying this mapping table on a single TestID will yield all the Suites it is in (and the priority in each Suite if you wish,) and querying on a single SuiteID will yield the list of Tests that are in it (and the priority of each test as defined in that Suite, if you wish.)

The concept of suites and sets, each with independent (but not necessarily different) priorities, is a *new concept that shouldn't be new.* Any tester who has tested software knows that priorities can and will often change per build, not per release and certainly not per product, and that the power of defining what is tested and what is not must lay with the tester, not with the limitations of the test tool.

Current test tools assume somehow that all tests are going to look the same for all test sets that you create for a project. That's a silly assumption to make. Our system empowers the tester because it's designed to support the way testers actually test, instead of the smoky backroom philosophy of how testing should occur.

I expect some pushback around suites and sets, so I'll offer another cheap metaphor.

Just like a hotel “suite” consists of one or more different sized rooms packaged together as a suite, a test suite consists of one or more possibly different, possibly same-sized tests.

A “set” of hotel rooms may consist of single rooms, double rooms, and suites all grouped together. The “set” is what is actually booked, whereas the “suite” is how its organized in inventory. Both are useful to the property management system because one provides valuable insight into organizing availability, and the other provides valuable reference in booking real rooms.

Likewise, a test set may consist of individual tests or suites of tests, plus or minus groups of other dependent or independent tests as the test engineer specifies. Hotels book out sets of rooms and suites for different purposes – weddings, reunions, special events. Testers create sets of tests for different purposes – regression testing, functional testing, one-off specialty testing, smoke testing, whatever. Ultimately, we want our system to flex to whatever testing the tester wants to imagine.

I'll say this again because it's important. A test set is a set of tests and/or suites that a tester wants to run based on her *judgement at a given point in time*. That judgment may be right or wrong but our system must support it regardless. QA is as much about art as it is about the scientific method. A test suite is a group of tests *grouped by their congruence to utility* in testing a system in an efficient, logical way as judged and prioritized by the tester.

Let's go through another example to drive this home. Imagine a smoketest suite grouped by functional positive and functional negative high priority 1 tests.

Suppose:

1. You are testing a high traffic web site that allows for anonymous sage and authenticated user interaction.
2. A build you receive contains major changes to the login functionality. Your smoketest suite contains high priority tests across all areas. You consider what testing will be a good expected value payback.
3. For your smoketest set for that build, you want to create a test set that contains your smoketest Suite for all authenticated areas, fewer unauthenticated areas (as that code path is by and large unaffected by changes in this build) -- and also all of the priority 2 and 3 tests functional positive tests around the Login module.
4. I hope you can see that in this case where your actual smoketest execution varies from your smoketest suite, you would not want to pollute your smoke test suite by changing it based on the build you receive. That is ill-advised and short-sighted because you want to preserve the thinking and evaluation that went into identifying your *smoketest baseline*. By preserving the freedom to identify a smoketest test set for this build, and not polluting your source smoketest suite, you permit yourself to have available the same tools for the next build as you do with this, using your good testing judgment each time in creating a reasonable smoke test set.

This conceptual separation of a smoketest *suite* from a smoketest test *set* is a concept that many QA engineers cannot grasp, and certainly a concept that most QA tools do not support. I must inspire you to maintain smoke tests and regression test suites separate from actual sets of tests to execute. You must differentiate between the value proposal of tests sets and test suites.

10.8 Create a table SETTYPE.

This will be a lookup table to hold the different kinds of test execution sets you create, easy enough. I originally called this Test Type, but that was a bad decision because a single test can be a baseline test, it can be part of a scalability test, and a handful of different performance tests. So, the type is really a decoration for an execution level concept, not an authoring level concept.

```
SETTYPE
==========
SetTypeID   (INT) PK NN
Name        (VARCHAR45)
```

Another way to think of suites is as convenient containers of “stuff that should have passed the last known good build.”

Our traditional upstanding citizen exemplifying this quality is our lowly Smoketest, which should be a group of high priority tests that do not change that often. This group of tests may increase slightly with

each build, but should not in theory decrease as the codebase grows (unless a story specifies the removal of a large functional block from the system.) Generally, smoketests slowly grow as functionality increases. They may grow geometrically at first, but eventually their growth trims off to a more linear feel. At first, the smoketest might increase by 50 or 100 tests per build, but 15 builds later, the smoketest goes up by maybe 5-9 tests per build.

Test suites should, in theory anyway, be a gold master of a well-defined group of tests – you might think of them as specifying the minimum "good idea" set of tests. When in doubt, run the whole thing. When informed, run the smart whittled down set.

10.9 At long last, create the SET table.
The SET table is the key table that will contain the data about our test execution data – but not the test execution data itself, that will be in our mapping table.

```
SET
==========
SetID       (INT) PK NN UQ
Name        (VARCHAR45) NN    # Force the name
Description (VARCHAR55)
AuthorID    (INT) NN          # Force an author
SetTypeID   (INT) NN          # Force the set type
RP          (INT)
CreatedOn   (DATETIME) NN     # Force the creation timestamp
LastRun     (DATETIME)
```

10.10 Create TESTSETMAP
SET needs a mapping table for it to contain the references to the tests – not the tests themselves – that may be executed and recorded as passed failed or something else.

```
TESTSETMAP
==========
SetID             (INT) PK NN UQ    # Tell me what set
#     FK=Set_containing_Test->SET.SetID
TestID            (INT) NN          # contains which tests
#     FK=Test_in_Set->TEST.TestID
TestPriorityInSet (INT)             # of what priority
#     FK=TestPriorityInSet_to_Priority->PRIORITY.PriorityId
ExecutionOrder    (INT) NN          # and in what run order
```

10.11 After all this talk about sets and suites it should come as a glorious relief to create the table SUITE. SUITE will look very much like SET with the difference that a test suite will map directly to a particular product.

```
SUITE
==========
SuiteID     (INT) PK NN UQ
ProductID   (INT) FK=Product_for_Suite->PRODUCT.ProductId
Name        (VARCHAR45) NN          # Gotta name the thing
Description (VARCHAR 45)
CreatedOn   (DATETIME)
```

```
LastRun     (DATETIME)
AuthorID    (INT) NN FK=Author_of_Suite->USER.UserID
RP          (INT) NN FK=RP_for_Suite->USER.UserID
```

We care less about when the Suite was last run, as it should have been run inside a SET, which we've already accounted for.

If we've got our framework in integrity, we are now connected interdependently within Tests, Test Suites, and Test Sets.

11 Everything else.

11.1 History

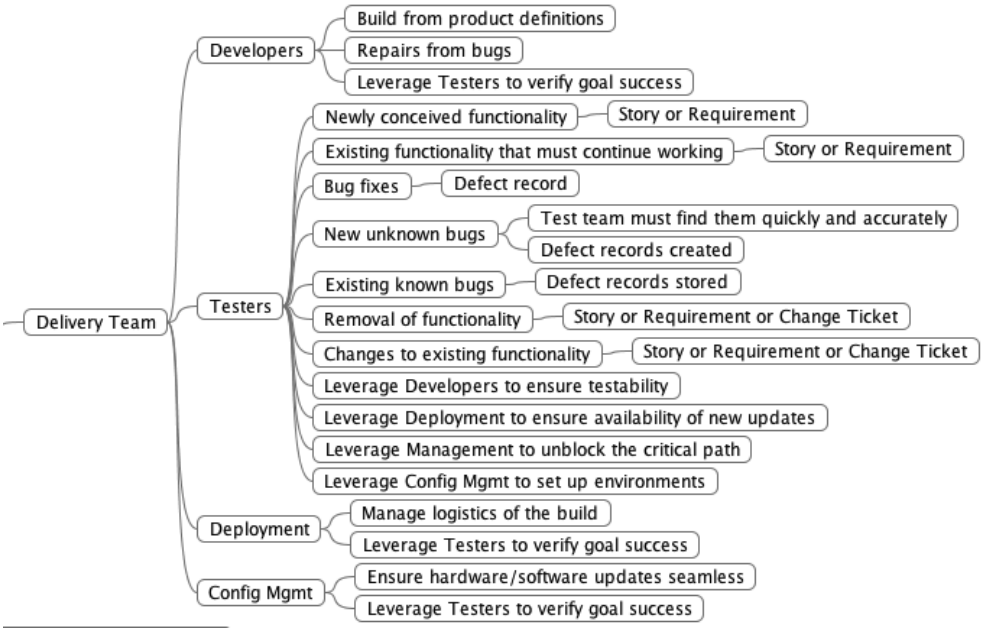
OK! So, the basic conceptual framework for our test assets is now wired together in a basic way. Cool. Our groundwork requires further supporting tables that uphold the principle notions of software testing.

Thinking about these data fields and their dependencies strikes an important recognition – how are we going to keep track of what changes were made to our tests? We absolutely need a history table. And it's going to be a big table, so later on we'll want to limit it and set it to roll over, or control its size in some way.

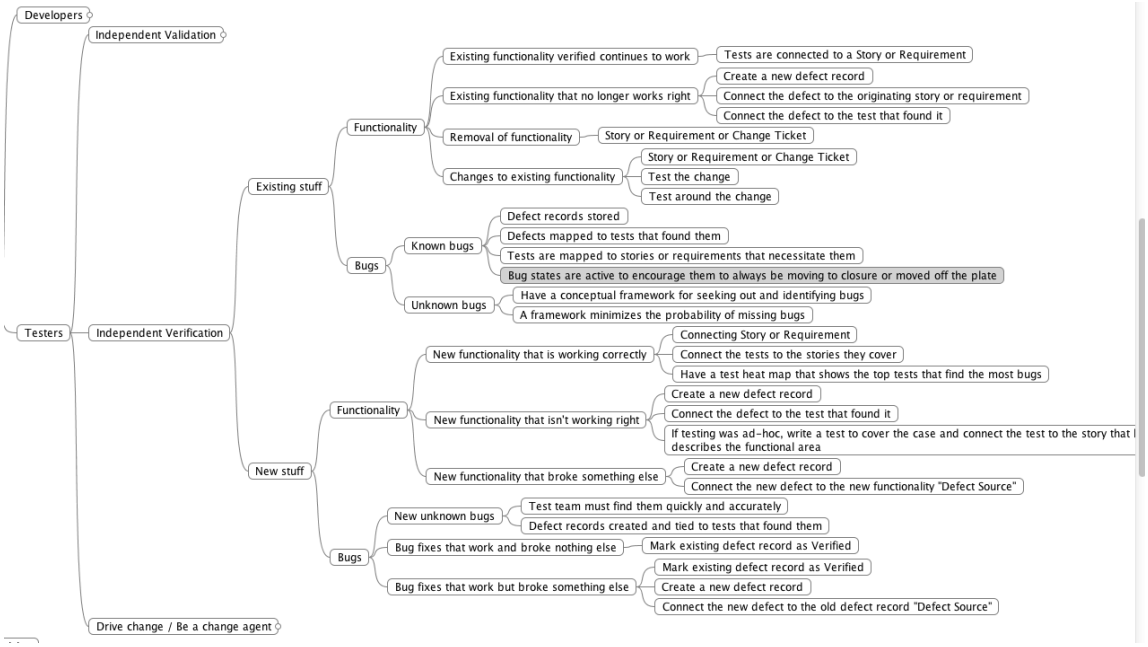
```
HISTORY
==========
TransactionID     (INT)       PK NN UQ
CreatedOn         (DATETIME)  NN
ErrorCode         (INT)
LongDesc          (VARCHAR255)
ShortDesc         (VARCHAR45) NN
UserID            (INT)       FK=UserID_of_xaction->USER.UserID
```

11.2 Define what we will be ready to accept from the development organization.

Lets examine the mind map I offered earlier and zoom in on the Build. We will construct relationships between these entities now.



Breaking the testing function out further, we have:



It is useful to have a schema that allows for maximum flexibility of receiving handoffs, versus a schema that contains minimalist concepts such as "a build contains functions and defects." While those concepts are often true, the life of a real tester may involve builds, patch builds, developers submitting artifacts late, or anything else you can dream up.

Designing a schema to handle the worst case scenario will enable a tool to adjust quickly to exceptions – versus a tool that forces a minimalist schema, which will force the tester to implement temporary and often haphazard workarounds.

So, the system I'm visualizing goes as such, from the top down, based on the flow of the person who has the responsibility (the RP, again borrowed directly from Ray Dalio's management principles):

A PRODUCT, which belongs to an RP in a BUSINESSUNIT in a DEPARTMENT, is owned by an RP.

This PRODUCT will be built by a team of developers and testers using some sort of process. Whatever that process is, that PRODUCT will be incrementally built by applying subsequent RELEASES that iteratively construct the data and assemblies necessary to complete the release as defined by the product manager and as managed by the project manager and engineering manager, and as signed off by the testing manager as whole and complete and correct.

Each RELEASE must have an RP and will probably be delivered by a SUBTEAM or a TEAM, whether that team is 7 people or 1 person, and there will be responsible parties identified for test sets and bugs.

Each RELEASE will be delivered to QA through a series of BUILDS. These builds may contain new, changed, or deleted sections of the application data, metadata, database, the database management software or DBMS, the application layer, caching systems, APIs, or end user presentation layer. Within each of these parts of the build anatomy, there will be bugs hidden for you to find in new areas, bugs hidden for you to find in areas that used to work but now work differently or not at all, and bug fixes that may repair a broken area, may not repair a broken area the way the developer intended, or repair the intended target sufficiently but in due course cause another area of the system to break.

It is the tester's responsibility to:

- **understand** the anatomy of the system under test and what goals you should have as testers,
- **plan** a (A) strategic course that engineering managers agree will achieve the testing goal, and a (B) tactical course that product and project managers agree is a reasonable duration of time that uses the best and most efficient technology possible to perform thorough, targeted testing,
- **design** good tests through whatever means necessary to achieve your goal
- **execute** those tests in a smart, prioritized fashion with one or many testers, and
- **manage** themselves and others to the priorities and tasks.

If you cannot do any of the above well (and it is likely you cannot, for example my tactical planning is often poor, and I spend too much time in strategic meta thinking as opposed to pragmatic test design, and I am often lazy in getting in synch with my technical managers on my test design because other areas are more interesting to me.)

Know your weaknesses and, if your work environment permits, be as open and transparent with your weaknesses as you can so that your managers can design around those weaknesses and challenge you to overcome them. If you work for a company that punishes those who fail in their work as opposed to openly diagnosing why the failure happened, build healthy working relationships with your peers and quietly ask them for advice

or challenging review of specific areas you know you are weak on. For example, having a trusted developer friend review your test strategy around an API change may be useful. Having a project manager associate review your test plan in order to challenge you on how you are thinking about using your people, testing process, and available testing technology will be useful. Ask yourself "are you working with what you have, or are you thinking of what you would really need to test excellently and doing a diff off of that."

Knowing these things, let's design a BUILD table that captures the sort of changes we have described above that will come together to form a release and support the process we have described. Let's review what a BUILD must contain in itself, and what information it must support:

(You will see and hear the CrUD abbreviation in software development: it stands for CReate, Update, Delete.)

For every anatomic portion of the system (we mentioned data layers, application layers, APIs etc earlier – you have to break the system open with your engineering manager and figure this structure out on your own) there will be two high level notions you as a tester will need to track. Thus, we may synthesize the mind map diagram of a build thusly:

A build contains CrUDs to

> (A) the functional and non-functional aspects of each identified area that (upon accurate testing will) work, work but break something else, or fail, AND
>
> (B) bug fixes that work, work but break something else, or fail

Let's call (A) the PAYLOAD, and let's call (B) the BUGFIXSET.

A payload, bug fix set, and a build will be nearly identical in structure because at a meta level they are all the same. At a practical level they have different owners, different timing, and map to different entities. We will distinguish those differences here and build the tables.

A payload is a collection of stories, which contain acceptance criteria, that are provided for by new or changed code delivery. So, we will need a container to map multiple acceptance criteria to multiple payloads.

A bug fix set is a collection of fixes to bugs that are provided for by code delivery and then tested and ultimately closed or reopened. So, we will need a container to map multiple bug fixes to multiple bug fix sets.

So first we define the payload and its mapping, the noble acceptance criteria:

```
PAYLOAD
==========
PayloadID         (INT) PK NN UQ
PayloadName       (VARCHAR45)
RP                (INT) NN
```

```
            # FK=RP_for_Payload->USER.UserID
DeliveredOn       (DATETIME) NN
ExpectedOn        (DATETIME)
SubTeamID         (INT)
            # FK=SubTeam_owning_Payload->SUBTEAM.SubTeamID
ShortDesc         (VARCHAR45)
LongDesc          (VARCHAR255)
```

In theory the smallest deliverable unit in Agile is the story, but in practice it is common for portions of stories to be delivered to QA. That means QA needs to use acceptance criteria, not stories, as the smallest minimum accepted unit of delivery. So we will map acceptance criteria to payloads, not stories. Writing a query in this schema to find all the stories that are partially or fully delivered will be straightforward.

Note that we will provide the capability to track each developer's work per acceptance criteria per payload delivery. This provides for multiple people working on the story over multiple payloads and even multiple builds.

```
ACCEPTANCETOPAYLOADMAP
==========
ACMapID           (INT) PK NN UQ
ACID              (INT)
            # FK=AC_in_Payload->ACCEPTANCECRITERION.ACID
PayloadID         (INT)
            # FK=Payload_Containing_AC->PAYLOAD.PayloadID
RP                (INT) # What developer owned it
            # FK=RP_for_This_AC_in_This_Payload->USER.UserID
BeganWorkOn       (DATETIME) # When did they start it
FinishedWorkOn    (DATETIME) # Delivered to a payload
Comments          (VARCHAR128)
```

Next we define the Payloads brother in arms, the BugFixSet, and its mapping, the lowly bug:

```
BUGFIXSET
==========
BugFixSetID       (INT) PK NN UQ
BugFixSetName     (VARCHAR45)
RP                (INT) NN
            # FK=RP_for_BFSet->USER.UserID
DeliveredOn       (DATETIME) NN
ExpectedOn        (DATETIME)
SubTeamID         (INT)
            # FK=SubTeam_owning_BFSet->SUBTEAM.SubTeamID
ShortDesc         (VARCHAR45)
LongDesc          (VARCHAR255)
```

```
BUGTOBUGFIXSETMAP
==========
BugMapID          (INT) PK NN UQ
BugID             (INT) NN
```

```
            # FK=Bug_In_BFS->BUG.BugID
BugFixSetID       (INT) NN
            # FK=BFS_Containing_Bug->BUGFIXSET.BugFixSetID
RP                (INT)  # Who owns testing this bug
            # FK=RP_of_Bug_In_BugFixSet->USER.UserID
OpenedOn          (DATETIME)
ClosedOn          (DATETIME)
Comments          (VARCHAR128)
```

Good. So, having defined the payload and the bugfixsets, we can now define the build and its two mapping tables to allow multiple payloads and bugfixsets to be assigned to builds:

```
BUILD
==========
BuildID     (INT) PK NN UQ    # The unique ID in the system
BuildName   (VARCHAR45)       # Allow for names such as 0.a.45-101
RP          (INT) NN
                  # FK=RP_for_build->USER.UserID
DeliveredOn (DATETIME)
ExpectedOn  (DATETIME)
SubTeamID   (INT)
ShortDesc   (VARCHAR45)
LongDesc    (VARCHAR255)


BUGFIXTOBUILDMAP
==========
BugFixMapID       (INT) PK NN UQ  # A many to many map
BugFixSetID       (INT)       # for which BugFixSet(s)
BuildID           (INT)       # are in which build(s)
Comments          (VARCHAR128)


PAYLOADTOBUILDMAP
==========
PayloadBuildMapID (INT) PK NN UQ  # A many to many map
PayloadID         (INT)       # for which payload(s)
BuildID           (INT)       # are in which build(s)
Comments          (VARCHAR128)
```

Let's review our punchlist of what our system needs to do:

- User defined descriptions of the anatomy of the application
- Descriptions of the functional and non-functional modes of each area
- Tests which capture useful independent verifications around each area
  - Functional verifications should include positive, negative, and edge case scenarios
  - Non-Functional verifications should include tests that determine a baseline response state, a soak test to ensure memory leaks or other duration-based issues are not present, a scalability test to see if the application response is suitable for 1.5x and 2x expected production load, and occasional stress tests which drives load up linearly in order

to determine the weakest point in the system and determine if its a problem.

- A notion of a test pass, fail, and breaking something else
- A detail notion of a portion of a build passing, failing, or breaking something else
- A high level synthesis notion of how to determine whether a build has passed or failed QA
- A notion of bugs and whether they are triaged, in the waiting room, being remedied, remedy completed and ready to retest, and whether retesting passed or failed.
- What the functional and non-functional changes are in a build
- What defect fixes are in a build
- The tests that address the functional and non-functional aspects in a build

We are missing a few pieces. We don't have a strong description of the anatomy of the system under test. We will need to understand that in order to provide meaningful metrics later in an expedient fashion.

```
ANATOMY
==========
AnatomyID         (INT) PK NN UQ
ReleaseID         (INT)
            # FK=Release_Containing_Anatomy=RELEASE.ReleaseID
AnatomyName       (VARCHAR45)
AnatomyDesc       (VARCHAR255)


ANATOMYDEPENDMAP
==========
AnatomyMapID      (INT) PK NN UQ
ThisAnatomyID     (INT)
            # FK=AnatomyMap_Anatomy_Value->ANATOMY.AnatomyID
ThatAnatomyID     (INT)
            # FK=AnatomyMap_Dependency->ANATOMY.AnatomyID


TESTANATOMYMAP
==========
TestAnatomyMapID  (INT) PK NN UQ
ThisAnatomy       (INT) # Multiple system parts can be
            # FK=AnatomyMapPart->ANATOMY.AnatomyID
CoveredByThatTest (INT) # Covered by many tests and vice versa
            # FK=AnatomyTestCovering->TEST.TestID
```

Good. Now we have enabled metrics to be able to map testing and bug discovery to different areas of the anatomy of the system so managers can run simple reports on how the quality of a build and product are doing in real time.

Our notion of a Bug needs further development, given the ideas we've presented so far. Namely, we need a way of knowing what triage state our bug is in.

Using the medical nursing metaphor, it is often useful to know whether a bug is

- in the waiting room,
- being remedied (in which case it is useful to know its PassValue),
- ready to retest,
- and whether retesting passed or failed.

Example of this pardigm:

PassValue=”New”. It's been “New” for a week. Why? In this case, TriageValue tells us “waiting room”. Oh, okay, nobody has touched it.

In another case, TriageValue might say “being remedied”, which means its still New, and in a developer's queue. In these two examples, the same PassValue leads to two different actions by the manager as a result of the information in Triage.

It is true that TriageState and PassValue could be seen as at least partially redundant. There are worlds in which you could do entirely without one or the other. I decided to leave both tables present in this schema to allow more flexibility for teams that prefer one metaphor over another.

```
TRIAGESTATE
==========
TriageID          (INT) PK NN UQ
LocalTriageID     (INT)       # Allows app layer to order a drop down
TriageName        (VARCHAR45) NN
Comments          (VARCHAR128)
```

Note: As a sneak peak to part 2 in order to give you a practical example of triage using TRIAGESTATE and PASSVALUE, the values I will propose for TRIAGE are:

TriageID	LocalTriageID	TriageName	Comments
0	1	Initial Wait	Waiting room
1	2	Understand	Being seen, understanding the problem
2	3	Design	Remedy is in design
3	4	Fix	Coding in progress to address bug
4	5	Delivery	Coding completed and bug is ready for re-test
5	6	Verify	Bug has been verified solved by QA

Imagine how these values can be combined with Bug PassFail values to give an exact context of how a bug is doing. I'll add the values I plan to add to PASSVALUE in Part 2 so you can contrast:

PassValueID	Name
0	Passed
1	Failed
2	Not Run
3	Queued
4	In Design

Imagine the value and time that will save to project managers, product managers, interested parties outside the subteam, and members of the subteam.

In a world that only tracks a “Bug Status” (remember, we replace the idea of status with a pass value), you would have something like:

Bug 1885, No Run.
Bug 1888, Passed.
Bug 1893, Failed.
Bug 1894, Failed.
Bug 1897, Failed.
Bug 1899, No Run.
Bug 1902, No Run.
Bug 1910, Passed.

That's not really intuitive or useful to anybody – even the QA engineer who set those values! On a big enough project, the QA engineer would have to sit and look up those bugs again to figure out what is going on with them. If it's No Run, how come? If it failed, what are we doing about it? If it passed, when will it be in production? These are natural questions that a good defect management system is going to provide. It will do this by capturing accurate triage-passvalue states as those states occur in real time. The application layer will be responsible for providing the API or interface to accept these triggers.

So, going along with our thinking about this, imagine our proposed system which contains a triage state and a pass value. Those same bugs we used as an imaginary example above might have these triage-passvalue paired states:

Bug 1885, Initial Wait No Run.
Bug 1888, Delivery Passed.
Bug 1893, Failed.
Bug 1894, Failed.
Bug 1897, Failed.
Bug 1899, Fix No Run.
Bug 1902, Delivery No Run.
Bug 1910, Verify Passed.

That information is tremendously more helpful than just seeing the states of bugs. We clearly see that bug 1885 hasn't been touched by development yet. But we see that the other No Run bug, 1902, has been delivered by development to QA and QA has not had time to run it. That tells a much more precise story, yes? Let's look at the two bugs that passed: we see that 1910 has been verified by QA and passed into the build awaiting deployment out to prod. On the other hand, our other passed bug, 1888, has passed Delivery to testing but it hasn't yet been verified by QA as having worked because it hasn't reached Verify Passed. As soon as QA re-tests 1888, the triage-passvalue pair update will change to “Verify Passed”. If the fix to 1888 doesn't pass QA, QA is going to kick it back to the developer and the triage-passvalue pair will change to “Understand Queued”. It is up to the settings in the application layer to adjust the wordsmithing so that the names of the triage-passvalue pairs make sense to your organization.

We'll update BUG accordingly:

BUG
==========
BugID (INT) PK NN UQ

```
ShortDesc         (VARCHAR45)
LongDesc          (VARCHAR255)
OpenedOn          (DATETIME)
ClosedOn          (DATETIME)
Author            (INT)
            # FK=Bug_Author->USER.UserID
OriginatingTestStepID (INT)
            # FK=Test_Step_Which_Found_Bug->STEP.StepID
CurrentPassValue  (INT)
            # FK=Current_Bug_Passvalue->PASSVALUE.PassValueID
RP                (INT) # Person responsible for ensuring bug closed
            # FK=RP_for_Bug->USER.UserID
TriageState       (INT) NN
            # FK=Triage_State_of_Bug->TRIAGESTATE.TriageID
```

A cute idea is the concept of tracking which bugfixes created more problems. This is a metrics thing – it can be useful in contributing to heat maps of where the high value targets likely lay for finding new bugs. It also tends to correspond to higher complexity in the software, even if such complexity is unintentionally designed.

```
BUGBLEED
==========
BugBleedID        (INT) PK NN UQ
ThisBug           (INT) NN
            # FK=BugBleed_Original_Bug->BUG.BugID
CreatedThatBug    (INT) NN
            # FK=BugBleed_New_Bug->BUG.BugID
Comments          (VARCHAR128)
```

Let's create a bleed container for new functionality that breaks other stuff also. “We got a bleeder here!”

```
STORYBLEED
==========
StoryBleedID      (INT) PK NN UQ
ThisACInThisPayload (INT) NN
            # FK=ACBleed_Original_AC->ACCEPTANCETOPAYLOADMAP.ACID
BrokeThisStory    (INT) NN
            # FK=ACBleed_Story_Harmed->ACCEPTANCECRITERION.StoryID
InThisPayload     (INT)
            # FK=ACBleed_Payload_Resp->PAYLOAD.PayloadID
Comments          (VARCHAR128)
```

11.3 People structures

Continue creating the people structures as needed. My form and format here is sloppy because I'm uncertain what I didn't cover above that isn't intuitive from here on out.

```
SUBTEAM
SubTeamID (INT) PK
Name (VARCHAR45)
TeamID (INT) FK to TEAM.TeamID
```

Create the TeamID PK in TEAM so you can map the FK in SUBTEAM to it:

TEAM
UserID
FName (VARCHAR45)
LName (VARCHAR45)
Email (VARCHAR45)
IM (VARCHAR45)
KnownAs (VARCHAR45)
Thumbnail (BLOB)

Some people with very long name shorten or westernize them. Some teams like to use nicknames to keep things fun.

We need a way to map users to Teams and Subteams, so:

TEAMUSERMAP
UserID (INT) PK
TeamID (INT) FK to TEAM.TeamID
SubTeamID (INT) FK to SUBTEAM.SubTeamID

Oops, that doesn't exist yet either, so bring up BUSINESSUNIT and create it:

BUSINESSUNIT
BusinessUnitID (INT) PK
Desc (VARCHAR45)
DeptID (INT) FK to DEPARTMENT.DeptID

Oops, we don't have DEPARTMENT built. OK, build it:

DEPARTMENT
DeptID (INT) PK
Desc (VARCHAR45)

11.4 Product, Story, Acceptance Criteria

Great! So, back up to PRODUCT. A product always belongs to a business unit, so tie it in with:

PRODUCT
ProductID (INT)
BusinessUnitID (INT) FK to BUSINESSUNIT.BusinessUnitID

Everything is starting to tie together now, and we're finding that we need to cascade up and down in our data schema to question our assumptions and see how everything hangs together. This is good. We're now left with defining how the details of the change sets and bug fixes are going to be defined.

I'm going to create a container for an Agile “Story” card that will probably be sync'd with a tool like Mingle. I could create lookup tables for this but I won't, assuming that we'll write an API to simply sync this table, rather than having our application layer also

reproduce a development workflow application (which is not in sync for this effort.)

STORY
StoryID (INT) PK
ShortDesc (VARCHAR45)
LongDesc (VARCHAR255)
AuthorID (INT)
OwnerID (INT)
CreatedOn (DATETIME)
Size (INT)
State (VARCHAR45)

Create a container for the acceptance criteria to map into stories. Allow for local ordering of acceptance criteria within a story for display purposes.

ACCEPTANCECRITERION
AcceptanceCriterionID (INT) PK
StoryID (INT) FK to STORY.StoryID
LocalOrderID (INT) NULL
Title (VARCHAR45)
Desc (VARCHAR4095)

14. Create the database.

Now that we have our initial schema designed, let's start the MySQL service and create the database.

Select Forward Engineer within Workbench.

You're going to find out that you made some mistakes. Here are the top two problems:

- You named tables the same names by mistake. You have to name them uniquely.
- You created foreign keys with the same names. You have to name them uniquely. I found that renaming all my FK's with a simple description of what they do TABLE_which_owns_COLUMN or something works for me.

Note: You can find the database generation source at my github location

https://github.com/eric314/OpenTest