Please note: This document is in active edit and is being stored on github only as an opportunity for those who wish to review and critique in real time to do so.
Thank you – Eric

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 2: Planning, Designing, and Executing Test Data on the OpenTestCenter Database

1 Introduction

Welcome to Part 2 of the OpenTestCenter documentation. In Part 1, we walked through the conceptual thinking and practical creation of an open source database schema that could house a useful, complete test and defect management system. After having completed Part 1, you should have a working MySQL database installed on your local disk and be ready to either begin using it with your choice of application layer and front end, or with this exercise in Part 2 which deals directly with database-level testing.

If you read Part 1, Part 2 picks up precisely where Part 1 finished. Part 2 will deal with the creation of test data and use the very principles we discussed in Part 1 to create the database... to now test it.

I gave an introduction about myself and my motivation behind creating this project. In brief, there are no open source testing solutions available today that both offer a complete test and defect management solution, while at the same time enabling the test engineer who is likely installing them to understand the conceptual thinking that went into creating the software. My perspective is that it would be far more valuable to both provide a superior open source solution while at the same time opening all design aspects to the test engineer and providing a narrative that brings the engineer along in the complete design of the system, from empty database to full functioning system. So, this series of documents is that narrative.

1.1 Goal

The specific goal of Part 2 is to enable a QA Engineer to think through what good testing looks like when testing a database directly without any application access. This is easy, because if you are reading this in order, you haven't installed or written an application to consume the database we've created in Part 1, and you haven't created any data in the database from Part 1.

At the end of Part 2, you will have an OpenTestCenter database full of a minimal set of optimum-use test data that you created with full knowledge of **why** each row was important – versus adding quasi random rows with no real common sense notion of why you're doing what you're doing.

1.2 Tools

While you are free to use whichever SQL tools you wish, I will recommend that you download and install Sequel Pro in order to work with the MySQL DB that we have created together in Part 1, OpenTestCenterDB.

Once you download and install Sequel Pro, ensure that you have identified your schema in MySQL Workbench with an appropriate name (I called my scheme OpenTestCenterDB) and be sure to forward engineer it. If you have changed the schema name, be sure and uncheck the GENERATE DROP SCHEMA top two options on the forward generation option screen, or Workbench will attempt to drop a schema that does not yet exist and error out.

The advantages to Sequel Pro are that it makes data entry easy using a sort of spreadsheet format, and since this part of our program is all about data entry, we will make good use of a tool that makes such action easy.

2 Write tests

Let's begin writing tests by inserting rows directly into our TEST table. We have set up TEST in such a way as to allow us to enter in just the name of the test. This is an advantages when we want to brainstorm and enter a lot of tests without being forced to enter a ton of other required fields.

2.1 Write tests in an intuitive order that tells a story

Tests that are written as a result of a functional breakdown of a system can be useful. When starting out, it can be advantageous to write tests as a function of how you think about the system. For a test and defect management database, then, it makes sense that you would write tests into such a system in order to see if it will let you write tests. This is as zen as it gets for a tester – a tester writing a test system and testing the system in order to verify it can manage tests.

Insert your first row into the TEST table by bringing up Sequel Pro (or whatever database client you wish to use), set up a database connection to your localhost using your root account, double clicking the TEST table, double click the Content icon up top, and double clicking the first row to add a row.

You'll remember from Part 1, and be able to see in your table structure, that we will enter column data in the order we created the columns in our forward engineering step.

TestID, TestModeID, Name, PriorityID, CreatedOn, LastRun, ReviewedOn, AuthorID, PassValue

These column values are nullable all except the TestID and Name. TestID and Name are the absolute minimum pieces of information we need to create a test, so we created those columns with a Non-Nullable flag, or NN. If you recall, we also told the database that certain columns must contain unique values,

and we identified those columns by specifying a UQ flag.

Insert the following values into the first row by typing "1" into the TestID field and typing the text into the "Name" column and hitting enter. That action results in the following row added:

0, NULL, Test that a row can be added to the TEST table, NULL, 0000-00-00 00:00:00, NULL, NULL, NULL, NULL

As we continue to enter test data I will not be as specific about the names of the columns as you should be able to find that information on your own using the tool.

You'll notice if you attempt to enter a line in TEST, you will need to create two test modes in TESTMODE. Create a Functional and Non-Functional entry.

TestModeID	Name	Description
1	Functional	Functional verification
2	Non-Functional	Usability, performance response

Write the following four tests:

TestID	TestModeID	Name	PriorityID	CreatedOn	LastRun	ReviewedOn	AuthorID	PassValue
1	1	Test that a row can be added to the TEST table	NULL	0000-00-00 00:00:00	NULL	NULL	NULL	NULL
2	1	Add Positive, Negative, and Edge testing Test Modes to the TESTMODE database	NULL	0000-00-00 00:00:00	NULL	NULL	NULL	NULL
3	1	Assign TestModes to the three existing tests in the TEST table	NULL	0000-00-00 00:00:00	NULL	NULL	NULL	NULL
4	1	Create a test set for this testing to keep track of our results	NULL	0000-00-00 00:00:00	NULL	NULL	NULL	NULL

2.2 Create test data for SetID, User, Set

Due to our foreign key relationships, we must enter data for the tables at the top of the dependency tree first. So in order for us to create Sets, we know that we must first enter test data for USER, SETTYPE, and then SET.

UserID	FName	LName	Email	IM	KnownAs	Thumbnail
1	Eric	NULL	NULL	NULL	NULL	NULL

SetTypeID	Name
1	Functional Manual verification
2	Non-Functional Manual verification

SetID	Name	Description	AuthorID	SetTypeID	RP	CreatedOn	LastRun
1	Dynamic test set for OpenTestCenter test data	NULL	1	1	1	0000-00-00 00:00:00	NULL

Test 4, "Create a test set for this testing to keep track of our results", required the above three steps. So let's add those to the STEP table.

StepID	LocalOrderID	Condition	ExpValue	CurrentPassValue	LastPassValue	ParentTest
1	1	Add UserID 1 "Eric" to the USER table.	MySQL accepts the row.	NULL	NULL	4
2	2	Add two Set Types to the SETTYPE table, Functional manual and Non-Functi...	MySQL accepts both rows.	NULL	NULL	4
3	3	Create a set in the SET table called "Dynamic test set for OTC test data"	MySQL accepts the row.	NULL	NULL	4

2.3 Write SQL to track your test data

A good habit to get into is to pop your head up now and then to step back and see where you are at. I find that, at least with my personality, my mind can latch on to a concept or get caught up in the inertia of some work and I can occasionally veer off in a direction that is suboptimal. So, I find it

useful to create tools that enable me to quickly gather some reality data to see how I am doing. Let's create such a tool now.

We've created tests, a test set, and some steps for a test, but they are in different tables and we have no way to see them all together. We can use the SQL language to specify how to join this information together to present it in a way we want.

First, decide what basic information we want to see together. Think in terms of what table and column you want to see. Remember, tables can contain columns with the same name, so its helpful to be specific. SQL will force us to be specific when writing the query, so best to get into the habit now.

The base set of information I'd like to see from our system to see what my database contains "at a glance" is:
Test.TestID
Test.Name
TestMode.Name (where TestMode.TestModeID=Test.TestModeID)
Step.LocalOrderID (where Step.ParentTest=Test.TestID)
Step.Condition (where Step.ParentTest=Test.TestID)
User.Name (where User.UserID = Test.AuthorID)

I'd like the test steps to be ordered by the local order ID, which is the purpose of that ID, and I'd like the overall effort to be ordered by the TestID which should give me a by and large sense of what I've written so far.

Go ahead and add this as a test to run in the TEST table in addition to executing the query.

```
SELECT      `TEST`.`TestID`, `TEST`.`Name`, `TESTMODE`.`Name`,
`STEP`.`LocalOrderID`, `STEP`.`Condition`, `USER`.`FName`
FROM  `TESTMODE`, `USER`, `TEST` LEFT JOIN `STEP` ON `STEP`.`ParentTest` =
`TEST`.`TestID`
WHERE       `TESTMODE`.`TestModeID` = `TEST`.`TestModeID` AND
            `USER`.`UserID` = `TEST`.`AuthorID`
```

That should do it, yes? If you execute it as written, zero rows are returned. Why? See if you can figure it out before continuing.

The answer is in the TEST table. If you look at the TEST table, we have not specified a value for AuthorID for the tests we've created. So, let's go ahead and add that test case (and execute it by adding AuthorID=1 to each of our tests.)

Having filled in the AuthorID and executing our query again, we see our expected results as designed.

TestID	Name	Name	LocalOrderID	Condition	FName
1	Test that a row can be added to the TEST table	Positive	NULL	NULL	Eric
2	Add Positive, Negative, and Edge testing Test Modes to the TESTMODE...	Positive	NULL	NULL	Eric
3	Assign TestModes to the three existing tests in the TEST table	Positive	NULL	NULL	Eric
4	Create a test set for this testing to keep track of our results	Positive	1	Add UserID 1 "Eric" to the USER table.	Eric
4	Create a test set for this testing to keep track of our results	Positive	2	Add two Set Types to the SETTYPE table, Functional manual and Non-F...	Eric
4	Create a test set for this testing to keep track of our results	Positive	3	Create a set in the SET table called "Dynamic test set for OTC test data"	Eric
5	Add test steps for test 4 to the STEP table	Positive	NULL	NULL	Eric
6	Write a SQL query to list all tests and steps to check our work as we go	Positive	1	Create the SQL Query specified in the text to display tests, steps, and userID	Eric
7	Add AuthorID to each test which corresponds with the User contained i...	Positive	NULL	NULL	Eric

You'll notice I used a LEFT JOIN in order to display tests with empty conditions in the STEP table. I'm not going to explain LEFT JOINs in this text because if you google it you should be able to get a sense of the difference between an inner join and a left join in minutes. Please do that

now, and play around with the above query to see what happens as you change the filter criteria and the use of inner and left joins.

Before we continue, let's clean up our query to shorten the syntax and adjust the column display.

To shorten the syntax, we will assign identifiers for our tables so we don't need to re-type the table name in the query.

```
SELECT      t.`TestID`, t.`Name`, m.`Name`, p.`LocalOrderID`, p.`Condition`,
u.`FName`
FROM        `TESTMODE` m, `USER` u, `TEST` t LEFT JOIN `STEP` p ON
p.`ParentTest` = t.`TestID`
WHERE       m.`TestModeID` = t.`TestModeID` AND
                  u.`UserID` = t.`AuthorID`
```

That query should return the exact results of the previous query. Now if we wish, we can rename the columns displayed to the user when the query results are returned. This will not rename the columns in the database, it is simply a cosmetic convenience.

Our final adjustment to our query below:

```
SELECT      t.`TestID` `ID`, t.`Name` `Test Title`, m.`Name` `Test Mode`,
p.`LocalOrderID` `Step Number`, p.`Condition` `Step Condition`, u.`FName`
`Test Author`
FROM        `TESTMODE` m, `USER` u, `TEST` t LEFT JOIN `STEP` p ON
p.`ParentTest` = t.`TestID`
WHERE       m.`TestModeID` = t.`TestModeID` AND
                  u.`UserID` = t.`AuthorID`
```

will return:

ID	Test Title	Test Mode	Step Number	Step Condition	Test Author
1	Test that a row can be added to the TEST table	Positive	NULL	NULL	Eric
2	Add Positive, Negative, and Edge testing Test Modes to the TESTMODE...	Positive	NULL	NULL	Eric
3	Assign TestModes to the three existing tests in the TEST table	Positive	NULL	NULL	Eric
4	Create a test set for this testing to keep track of our results	Positive	1	Add UserID 1 "Eric" to the USER table.	Eric
4	Create a test set for this testing to keep track of our results	Positive	2	Add two Set Types to the SETTYPE table, Functional manual and Non-...	Eric
4	Create a test set for this testing to keep track of our results	Positive	3	Create a set in the SET table called "Dynamic test set for OTC test data"	Eric
5	Add test steps for test 4 to the STEP table	Positive	NULL	NULL	Eric
6	Write a SQL query to list all tests and steps to check our work as we go	Positive	1	Create the SQL Query specified in the text to display tests, steps, and...	Eric
7	Add AuthorID to each test which corresponds with the User contained...	Positive	NULL	NULL	Eric

2.4 Write remaining tests for TEST table and its dependencies

Returning our attention back to the TEST table, there are fields in TEST that we have not yet used. We wish to exercise the entire schema, so let's think through reasonable test scenarios for TEST.

We will begin with PriorityID. We need to define our thinking around test prioritization. My framework that I offer to you to take or leave is:

Low: Negative exploratory, negative edge case, or opportunistic tests that have a low probability of turning up bugs
Medium: Positive exploratory and positive edge case that have a reasonable probability of turning up bugs
High: High probability tests that possibly qualify for inclusion in smoke tests
Must-Do: High probability tests of any mode that must absolutely be run in all smoke tests

Implemented in the PRIORITY table, it looks like this:

PriorityID	Name	Value
0	Low	1
1	Medium	2
2	High	3
3	Must-Do	4

The PriorityID is an arbitrary unique value. I just started at 0 for purposes of illustration and because choosing any other starting point didn't lend any value to me (although it might to you.)

The value is an arbitrary weight that future metrics calculations may use. For example, I could imagine an algorithm that computes a heat value based on the priority value times the number of times a test discovered a bug times the priority of that bug. This would provide a matrix color map, if plotted over time, of the "worst neighborhoods" in the code base as discovered by the QA team "so far". Areas of no color in such a heat map may also indicate areas that QA is testing insufficiently or incorrectly – such metrics are useful for engineering holding QA accountable, and vice versa.

Perhaps the High and Must Do weights should be notably higher, both to provide more useful metrics later on and a clearer differentiation between ID and value. Let's adjust that now.

PriorityID	Name	Value
0	Low	1
1	Medium	2
2	High	5
3	Must-Do	7

Note: Don't be tempted to make rules about what percentage of time you should be spending on writing Low, Medium, High, and Must-Do tests. Different software products with different teams have different needs. The general pattern I see is a strong focus on the high probability, high impact testing. Then as the product matures, more and more time is spent on negative testing to ensure the system reacts to error conditions in a reasonable way.

Also note: If you follow the concept of risk-based testing, which is prioritizing your testing effort according to the probability of failure and the impact of that failure to the business, your test assets will probably follow such a pattern.

Write a test to update the PriorityIDs in the TEST table. Execute that test, entering reasonable priorities. Your TEST table should look very similar to this:

TestID	TestModeID	Name	PriorityID	CreatedOn	LastRun	ReviewedOn	AuthorID	PassValue
1	1	Test that a row can be added to the TEST table	3	0000-00-00 00:00:00	NULL	NULL	1	NULL
2	1	Add Positive, Negative, and Edge testing Test Modes to the TESTMODE database	3	0000-00-00 00:00:00	NULL	NULL	1	NULL
3	1	Assign TestModes to the three existing tests in the TEST table	3	0000-00-00 00:00:00	NULL	NULL	1	NULL
4	1	Create a test set for this testing to keep track of our results	3	0000-00-00 00:00:00	NULL	NULL	1	NULL
5	1	Add test steps for test 4 to the STEP table	2	0000-00-00 00:00:00	NULL	NULL	1	NULL
6	1	Write a SQL query to list all tests and steps to check our work as we go	0	0000-00-00 00:00:00	NULL	NULL	1	NULL
7	1	Add AuthorID to each test which corresponds with the User contained in the...	3	0000-00-00 00:00:00	NULL	NULL	1	NULL
8	1	Create priority IDs in the Priority table	3	0000-00-00 00:00:00	NULL	NULL	1	NULL
9	1	Update PRIORITY weights to generate reasonable metrics	1	0000-00-00 00:00:00	NULL	NULL	1	NULL
10	1	Update current priority of tests in the TEST table	2	0000-00-00 00:00:00	NULL	NULL	1	NULL

Let's continue and fill in the CreatedOn timestamp. MySQL will tell you what

the DATETIME is now by running

```
SELECT now();
```

We can leave the LastRun and ReviewedOn columns empty for now. We can quickly review the schema to see that they are NULLable DATETIME fields, so there isn't much use in exercising them by filling them with values and deleting them. If we were to do this, we'd be essentially testing that MySQL worked, not that our schema was constructed correctly. It is important to keep in mind the target of your tests – I have seen testers get carried away with their testing by testing everything, rather than testing everything reasonably valuable targeted to the system under test. In our case, the system under test is our database schema design: not MySQL. Take a moment and think that through and see if you agree with me. If you disagree, write out why you disagree, and then pretend I am sitting there, read your disagreement, and replied to you with "Consider the perspective that everything in good testing is an expected value equation. How would you go about figuring out how to measure the value of your proposal? Would you pay you for the time and effort needed to do it your way?" If so, go for it.

Let's take a look at PASSVALUE now. In the life and times of a test, in the framework I think of a test in, the basic fundamental buckets a test is going to find itself is as follows. A test will be conceptualized and designed and written, it'll then be lined up to be tested, it will have either passed or failed that test. So, that gives me five fundamental PASSVALUE values (again that you can take or leave as you see fit for your testing framework):

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

Update the PassValue in the tests in TEST, so at this point your TEST table should look something like the following:

TestID	TestModeID	Name	PriorityID	CreatedOn	LastRun	ReviewedOn	AuthorID	PassValue
1	1	Test that a row can be added to the TEST table	3	0000-00-00 00:00:00	NULL	NULL	1	NULL
2	1	Add Positive, Negative, and Edge testing Test Modes to the TESTMODE database	3	0000-00-00 00:00:00	NULL	NULL	1	NULL
3	1	Assign TestModes to the three existing tests in the TEST table	3	0000-00-00 00:00:00	NULL	NULL	1	NULL
4	1	Create a test set for this testing to keep track of our results	3	0000-00-00 00:00:00	NULL	NULL	1	NULL
5	1	Add test steps for test 4 to the STEP table	2	0000-00-00 00:00:00	NULL	NULL	1	NULL
6	1	Write a SQL query to list all tests and steps to check our work as we go	0	0000-00-00 00:00:00	NULL	NULL	1	NULL
7	1	Add AuthorID to each test which corresponds with the User contained in the...	3	0000-00-00 00:00:00	NULL	NULL	1	NULL
8	1	Create priority IDs in the Priority table	3	0000-00-00 00:00:00	NULL	NULL	1	NULL
9	1	Update PRIORITY weights to generate reasonable metrics	1	0000-00-00 00:00:00	NULL	NULL	1	NULL
10	1	Update current priority of tests in the TEST table	2	0000-00-00 00:00:00	NULL	NULL	1	NULL

Update the priorities of the tests in the TEST table, and be sure to enter a row in the TEST table for each action you take. For each mistake or inconvenience you find, enter a row in BUG.

Let's also begin to attend to some of these empty columns. Fill in the CreatedOn timestamp for all the rows in TEST if you haven't been. You've entered PASSVALUE values. Be sure to go through the rows and ensure the PASSVALUEs are accurate. If you are using my system, the value should be 2.

Note: If you are interested in monitoring the growth of your database as you proceed and add test data rows, you can do so by summing the union of the count of our identified tables:

```
select sum(n) FROM (
```

```
select count(*) as n from AcceptanceCriterion union
select count(*) as n from AcceptanceToPayloadMap union
select count(*) as n from Anatomy union
select count(*) as n from AnatomyDependMap union
select count(*) as n from Bug union
select count(*) as n from BugBleed union
select count(*) as n from BugFixSet union
select count(*) as n from BugToBugFixSetMap union
select count(*) as n from Build union
select count(*) as n from BusinessUnit union
select count(*) as n from Department union
select count(*) as n from History union
select count(*) as n from PassValue union
select count(*) as n from Payload union
select count(*) as n from Priority union
select count(*) as n from Product union
select count(*) as n from `Release` union
select count(*) as n from `Set` union
select count(*) as n from SetType union
select count(*) as n from Step union
select count(*) as n from Story union
select count(*) as n from StoryBleed union
select count(*) as n from SubTeam union
select count(*) as n from Suite union
select count(*) as n from Team union
select count(*) as n from TeamUserMap union
select count(*) as n from Test union
select count(*) as n from TestAnatomyMap union
select count(*) as n from TestMode union
select count(*) as n from TestSetMap union
select count(*) as n from TestSuiteMap union
select count(*) as n from TriageState union
select count(*) as n from `User`
) s1;
```

As your table names change, of course, you will need to edit the query accordingly. By the end of this doc, some tables will have been added and dropped, so this query will not work as written. If you wish to skip to the end and have a working query with the current data model as of this writing, it is:

```
SELECT sum(n) FROM (
SELECT count(*) AS n FROM AcceptanceCriterion UNION
SELECT count(*) AS n FROM AcceptanceToPayloadMap UNION
SELECT count(*) AS n FROM Anatomy UNION
SELECT count(*) AS n FROM AnatomyDependMap UNION
SELECT count(*) AS n FROM AnatomyTestMap UNION
SELECT count(*) AS n FROM AssociateMap UNION
SELECT count(*) AS n FROM Bug UNION
SELECT count(*) AS n FROM BugFixSet UNION
SELECT count(*) AS n FROM BugToBugFixSetMap UNION
SELECT count(*) AS n FROM Build UNION
SELECT count(*) AS n FROM BusinessUnit UNION
SELECT count(*) AS n FROM Department UNION
SELECT count(*) AS n FROM History UNION
SELECT count(*) AS n FROM PassValue UNION
SELECT count(*) AS n FROM Payload UNION
SELECT count(*) AS n FROM Priority UNION
SELECT count(*) AS n FROM Product UNION
```

```
SELECT count(*) AS n FROM Relationship UNION
SELECT count(*) AS n FROM RelationshipMap UNION
SELECT count(*) AS n FROM `Release` UNION
SELECT count(*) AS n FROM `Set` UNION
SELECT count(*) AS n FROM SetType UNION
SELECT count(*) AS n FROM Step UNION
SELECT count(*) AS n FROM Story UNION
SELECT count(*) AS n FROM SubTeam UNION
SELECT count(*) AS n FROM Suite UNION
SELECT count(*) AS n FROM SuiteMap UNION
SELECT count(*) AS n FROM Team UNION
SELECT count(*) AS n FROM TeamUserMap UNION
SELECT count(*) AS n FROM Test UNION
SELECT count(*) AS n FROM TestMode UNION
SELECT count(*) AS n FROM TestSetMap UNION
SELECT count(*) AS n FROM TriageState UNION
SELECT count(*) AS n FROM `User`
) s1;
```

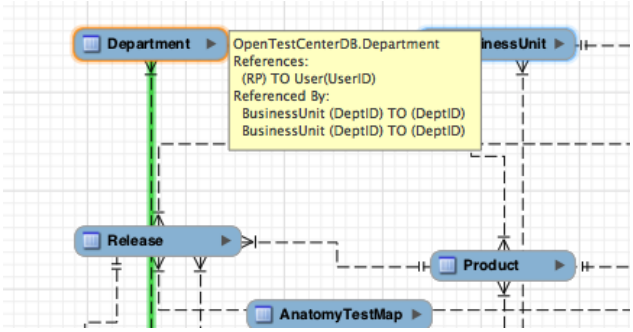
Based on the test rows you've entered, at this point your TEST table contents should look something like the following. Note that all columns now have values except for LastRun and ReviewedOn, which at this point it makes sense to leave null.

TestID	TestModeID	Name	PriorityID	CreatedOn	LastRun	ReviewedOn	AuthorID	PassValue
1	1	Test that a row can be added to the TEST table	3	2012-10-08 09:57:47	NULL	NULL	1	2
2	1	Add positive, negative, and edge testing Test Modes to the TESTMODE table	3	2012-10-08 09:57:47	NULL	NULL	1	2
3	1	Assign test modes to the three existing tests in the TEST table	3	2012-10-08 09:57:47	NULL	NULL	1	2
4	1	Create a test set for this testing to keep track of our results	3	2012-10-08 09:57:47	NULL	NULL	1	2
5	1	Add test steps for test 4 to the STEP table	2	2012-10-08 09:57:47	NULL	NULL	1	2
6	1	Write a SQL query to list all tests and steps to check our work as we go	0	2012-10-08 09:57:47	NULL	NULL	1	2
7	1	Add AuthorID to each test which corresponds with the User contained in the...	3	2012-10-08 09:57:47	NULL	NULL	1	2
8	1	Optimize the SQL query to use table abbreviations and a LEFT JOIN	3	2012-10-08 09:57:47	NULL	NULL	1	2
9	1	Create PriorityIDs in the Priority table with reasonable weights	1	2012-10-08 09:57:47	NULL	NULL	1	2
10	1	Update the priorities of the tests in TEST	2	2012-10-08 09:57:47	NULL	NULL	1	2
11	1	Update the CreatedOn timestamp values of all in TEST	2	2012-10-08 09:57:47	NULL	NULL	1	2
12	1	Brainstorm and enter Pass Values in PASSVALUE	2	2012-10-08 09:57:47	NULL	NULL	1	2
13	1	Update all tests in TEST to contain accurate PassValues of 2 = Not Run	2	2012-10-08 16:31:00	NULL	NULL	1	2

2.5 Write tests to describe the Team, Product, and Release

The next logical step is to create the tests data linking all the way to the product so we can specify what needs to be tested.

The easiest way to do this is by using the ER diagram. The foreign key relationships will light up when you hover over a table, enabling you to trace the relationship between tables up to the top level table. In our case, that top level table is DEPARTMENT.



These are the next steps I followed in writing test data. I think they are somewhat straightforward so I will let the reviewers (that's you, as in right now) tell me where more explanation is wanted:

TestID	TestModelID	Name	PriorityID	CreatedOn	LastRun	ReviewedOn	AuthorID	PassValue
14	1	Create row of data in DEPARTMENT	2	2012-10-08 16:31:00	NULL	NULL	1	2
15	1	Create department owner in user table and update DEPARTMENT	2	2012-10-08 09:57:47	NULL	NULL	1	2
16	1	Create a testing user Suzy in the USER table	2	2012-10-08 09:57:47	NULL	NULL	1	2
17	1	Create a testing center of excellence business unit in BUSINESSUNIT belonging to R&D	2	2012-10-08 09:57:47	NULL	NULL	1	2
18	1	Create a product in the PRODUCT table called OpenTestCenter that belongs to the business unit just created	2	2012-10-08 09:57:47	NULL	NULL	1	2
19	1	Create a team called Test Tools Delivery Team in the TEAM table belonging to the business unit.	2	2012-10-08 16:49:00	NULL	NULL	1	2
20	1	Update the TEAMUSERMAP to map who is on the team just created.	2	2012-10-08 16:49:00	NULL	NULL	1	2
21	1	Create a subteam called OpenTestCenter team in the SUBTEAM table belonging to the team in test 19.	2	2012-10-08 16:49:00	NULL	NULL	1	2
22	1	Map your user onto the OTC subteam in TEAMUSERMAP	2	2012-10-08 17:00:00	NULL	NULL	1	2
23	1	Create the alpha and beta releases of OTC in RELEASE	2	2012-10-08 17:00:00	NULL	NULL	1	2
24	1	Define two suites of tests in SUITE, an OpenTestCenter smoketest and a full functional suite	2	2012-10-08 17:00:00	NULL	NULL	1	2
25	1	Of the tests available in TEST, map them to test suites by adding rows to TESTSUITEMAP	2	2012-10-08 17:00:00	NULL	NULL	1	2

2.6 Conceptualize and author the Anatomy

I'll talk about the Anatomy of OpenTestCenter because it's in my mind and I don't expect it to be in yours (yet.) Let's do these:

TestID	TestModelID	Name	PriorityID	CreatedOn	LastRun	ReviewedOn	AuthorID	PassValue
25	1	Of the tests available in TEST, map them to test suites by adding rows to TESTSUITEMAP	2	2012-10-08 17:00:00	NULL	NULL	1	2
26	1	Create the anatomy of the OpenTestCenter app in ANATOMY for both releases	2	2012-10-08 17:00:00	NULL	NULL	1	2
27	1	Establish anatomy dependencies by entering them in ANATOMYDEPENDMAP	2	2012-10-08 17:36:00	NULL	NULL	1	2

My first take on the Anatomy of the application structure is as follows:

AnatomyID	ReleaseID	AnatomyName	AnatomyDesc
1	1	Database	Database system: MySQL
2	1	Data model	Data ER diagram representing the model
3	1	Data config	MySQL configuration incl indexes and performance settings
4	1	Application	Java app layer providing QA-centric data input and algorithmic enforcing
5	1	Caching	Caching and metadata precalculations for metrics and system information
6	1	Logging	Application logging layer
7	1	Presentation	User interface
8	2	Database	Database system: MySQL
9	2	Data model	Data ER diagram representing the model
10	2	Data config	MySQL configuration incl indexes and performance settings
11	2	Application	Java app layer providing QA-centric data input and algorithmic enforcing
12	2	Caching	Caching and metadata precalculations for metrics and system information
13	2	Logging	Application logging layer
14	2	Presentation	User interface

There are probably a few plus-and-minuses that will be introduced in beta (or maybe a lot) but as of this writing that is my visualization of the end game of OpenTestCenter. As this is really just test data at this point, it's not terribly important to get the content right exactly right- but the content should *have integrity within itself* to someone looking at it.

Note: What I mean by have integrity within itself is that if you were to enter, "Front End", "App layer", and "Back end", that would be in integrity within itself. It would clearly be a much abbreviated version of an n-tier system. Using test data of "aaaaa", "bbbb", and "c1c1c1" would be useless test data. It lacks integrity and usefulness. I've met a number of QA engineers that choose the latter. Then later it's nothing but facepalms for everyone. Don't be that guy. Or gal.

OK, back to the Anatomy of the system. Our system allows for us to identify dependencies within the Anatomy, which are captured in the ANATOMYDEPENDMAP. This will be useful to us as we look at the system from a product perspective to see what sort of testing and test results look like from a dependency point of view. This is probably not useful for small projects and far more

useful for large, multi-team projects.

AnatomyID	ReleaseID	AnatomyName	AnatomyDesc
1	1	Database	Database system: MySQL
2	1	Data model	Data ER diagram representing the model
3	1	Data config	MySQL configuration incl indexes and performance settings
4	1	Application	Java app layer providing QA-centric data input and algorithmic enforcing
5	1	Caching	Caching and metadata precalculations for metrics and system information
6	1	Logging	Application logging layer
7	1	Presentation	User interface
8	2	Database	Database system: MySQL
9	2	Data model	Data ER diagram representing the model
10	2	Data config	MySQL configuration incl indexes and performance settings
11	2	Application	Java app layer providing QA-centric data input and algorithmic enforcing
12	2	Caching	Caching and metadata precalculations for metrics and system information
13	2	Logging	Application logging layer
14	2	Presentation	User interface

When we get to writing our application, we might code a routine to return a dependency tree to graphically indicate what system parts are dependent on what other system parts, and most importantly, what those dependencies *mean* as a function of time to deliver.

Note: Don't get caught up in providing metrics just to provide metrics. For example, your manager might ask you for pass/fail metrics. OK, those are easy enough to provide, so you might just provide them. But then, most managers will come back to you after you've delivered them and ask you when you think testing will be done. And then after you've discovered defects, most managers will come back and ask when development will need to have those fixed in order to deliver on time. And you'll need to think about the risk of each of the defects not being delivered, and that probably gets tricky, and it's difficult to explain in the moment because you were just in the middle of something. A good test and defect management system will not only compile metrics for you, it will enable you to set useful weights and metadata in order to provide *predictive utility*. The more useful data QA can enter about a system under test to such a predictive system (ours will be such a system), the more managers can access useful synthesized information instead of seeing data that they needs further explanation which requires interrupting you.

Now we want to map our current tests to the parts of the Anatomy they cover. A smart application will provide both an API and a fast & easy user interface to do this. We have only our SQL client, so we visualize the most likely useful mapping and we enter it line by line.

As you enter this data, imagine the dependencies in your mind. I may have missed some, or may have entered redundant mappings. In the real world this would be challenged, but for purposes of test data it's fine for now.

TestAnatomyMapID	ThisAnatomy	CoveredByThatTest
1	1	1
2	2	2
3	2	3
4	2	4
5	2	5
6	2	6
7	2	7
8	2	8
9	2	9
10	2	10
11	2	11
12	2	12
13	2	13
14	2	14
15	2	15
16	2	16
17	2	17
18	2	18
19	2	19
20	2	20
21	2	21
22	2	22
23	2	23
24	2	24
25	2	25
26	2	26
27	2	27
28	2	28
29	2	29
30	2	30
31	2	31
32	2	32
33	2	33
34	2	34
35	2	35
36	2	36
37	2	37
38	2	38
39	2	39

2.7 Author tests for the contents of a Build: Payload and Bugfixset

You might recall from Part 1 that I conceptually defined a software build as containing two buckets of stuff which I called a Payload and a Bugfixset. A payload is stuff from engineering (developers, DBAs, front end artists, whatever) that needs to be tested. A Buxfixset is stuff that someone (maybe QA) found broken and is now thought to be fixed with this bucket of band-aids or root cause tweaks.

By defining a build with these two components, we can then say that a build consists of one or more payloads, and one or more bugfixsets, from one or more teams. On big projects, this can be very useful to separate and track. Because in the real world, payloads change all the time (even up to and after the time the build is due) and bugfixsets change all the time.

So let's do these:

TestID	TestModelID	Name	PriorityID
28	1	Write a few rows in PAYLOAD that describe the functional updates to OTC you have created in the back end.	2
29	1	Create a row in BUGFIXSET that will contain the bugs we want to include in alpha	2
30	1	Create rows in TRIAGESTATE that indicate the triage levels of a bug	2
31	1	Create some bugs in the BUG table	2
32	1	Create rows in BUGTOBUGFIXSET to map bugs just fixed to their bug fix set	2
33	2	Try to create a row in BUGTOBUGFIXSET that maps a bug ID that doesn't exist. The database should r...	2

I won't talk about 28 and 29 because I think they are straightforward. I'll touch on Triagestate for a moment. You'll recall in Part 1 I gave you a sneak peak into my philosophy of using two sets of information to describe a bug – its TriageState and its PassValue. You may agree or disagree with this approach – I gave some real world examples in Part 1 if you'd like to flip back to that and read it through if you only skimmed it before.

TriageID	LocalTriageID	TriageName	Comments
0	1	Waiting	Waiting room
1	2	Understanding	Being seen, understanding the problem
2	3	Designing	Remedy is in design
3	4	Fixing	Coding in progress to address bug
4	5	Delivery	Coding completed and bug is ready for re-test
5	6	Verification	Bug has been verified solved by QA
6	7	Shipment	Bug has been deployed to staging and prod.

I'll share the bugs that I entered into my BUG table as I went along. I wanted to try to use the OpenTestCenter database to track the real bugs I was finding in OpenTestCenter, so these examples are actual examples of things I was finding. As I found these bugs, I would stop and make design changes, synchronize the model with the database, add the changes to git, and push to github at the end of a session.

BugI	ShortDesc	LongDesc
1	DEPARTMENT table had an extra Desc field	NULL
2	DEPARTMENT description fields are too short	Can't enter in the data I want to
3	BUSINESSUNIT description fields are too short	Same as bug 2
4	TEAM RP wasn't marked as NN	RP is required for the TEAM to indicate the team leader
5	TEAMUSERMAP didn't have correct PK	The PK for TEAMUSERMAP should be a map ID, not a TeamID
6	TESTANATOMYMAP didn't have correct PK	Similar to 5, the PK should be a unique map id
7	PAYLOAD description fields way too short	Description fields need to be set way past 45 chars
8	SUITE description fields too short	Along the lines of 7
9	PAYLOAD doesn't contain a BuildID FK	Build should not contain a PayloadID, Payload should contain a BuildID to enable many Payloads to one Build
10	BUGFIXSET doesn't contain a BuildID FK	Same as 9 but for BFS
11	BUILD should not contain a payloadID or BFSID	Related to 9 and 10, Build should have payloadID and bugfixsetIDs removed
12	BUILD ShortDesc column is too short	The 45 should really be a 128
13	Related bugs aren't clear	There's no way in the DB to provide a mapping to which bugs are related to one another or depend on other bugs being fixed
14	Initial pass at relationship schema failed	The initial design of relationships did not show common sense or connected thinking
15	AC title isn't long enough	Make the title field 128 long

The BigFixSet I created was:

BugFixSetID	BugFixSetName	RP	DeliveredOn	ExpectedOn	SubTeamID	ShortDesc	LongDesc	AssociatedBuild
1	OTC alpha bugs	1	2012-10-08 17:51:00	2012-10-08 19:00:00	1	Initial group of alpha bug fixes found along	NULL	NULL

...and the BugToBugFixSetMap rows I wrote were:

BugMapID	BugID	BugFixSetID	RP	OpenedOn	ClosedOn	Comments
1	1	1	1	2012-10-08 19:12:00	2012-10-08 19:12:00	Creating map for initial testing
2	2	1	1	2012-10-08 19:12:00	2012-10-08 19:12:00	NULL
3	3	1	1	2012-10-08 19:12:00	NULL	NULL
4	4	1	1	2012-10-08 19:12:00	NULL	NULL
5	5	1	1	2012-10-08 19:12:00	NULL	NULL
6	6	1	1	2012-10-08 19:12:00	NULL	NULL
7	7	1	1	2012-10-08 19:12:00	NULL	NULL
8	8	1	1	NULL	NULL	NULL
9	9	1	1	NULL	NULL	NULL
10	10	1	1	NULL	NULL	NULL
11	11	1	1	NULL	NULL	NULL
12	12	1	1	NULL	NULL	NULL
13	13	1	1	NULL	NULL	NULL
14	14	1	1	NULL	NULL	NULL

I wanted to be sure to enter some rows with full data, and some rows with less data to see how our application handles that later. It had nothing to do with being lazy or wanting to get the table full of test data so I could go eat dinner. Absolutely nothing to do with any steak dinner that may or may not have been awaiting me.

2.8 Relationships and Associations

One of the more powerful features of this system is including the concept of relationships and associations between entities. Good QA engineers are like scientists – they remember the intricacies of a systems capabilities and weaknesses, they compartmentalize what works well and what doesn't, and by grouping bugs, stories, and tests together in their head they are able to synthesize a "feel" for where to look for bugs and how testing should be adjusted based on what is coming in with the build.

The reason that QA engineers have to keep all this in their head is because there aren't any testing tools that come remotely close to enabling the tester to easily and sensibly enter this information. The goal of our system is to not only solve this problem, but to solve it beautifully and transparently. Surely many smart engineers have tried to tackle this problem before and failed, making the problem only for the most arrogant to now attempt.

Let us begin.

The approach I have taken is to quantify first entities, and second the kind of dependencies among those entities. In entities, we have stories with acceptance criteria (substitute "requirements" if you are running a waterfall shop), builds which contain code to satisfy those criteria, tests to test those criteria, bugs that emerge from failure to meet expectation, more builds with more code to satisfy the new criteria, and testing which results in more bugs, possibly more stories, and possibly changes in scope.

Let us consider the idea of a relationship (the bachelor(ette)s among us shiver...) No no... a relationship as in one thing sharing a common criteria with another thing. (Whew.)

Let us define relationship as a group of stuff that shares something in common. The stuff, for us, is our entities. For example:

Imagine an online reservation system that is designed to take and enforce your dinner reservation at a restaurant. The stories or requirements that specify the product behavior, the code that is written to solve those

requirements, the bugs that emerge from code faults, and the builds that lead up to that system in the end are all related. Because that relationship is so obvious and far-reaching, we call that relationship a Product, and the notion of a product is a nearly universal one and an easy one to begin with.

But, let's take this idea of the dinner reservation system as a product and break it down into a possible picture of reality of delivering it. There are smaller relationships inside of the "Product" relationship, yes?

There is probably a piece of functionality that inputs a customer name into a record or set of records and perhaps even looks up the customer's history to see what time or seat they prefer, maybe even matched to the restaurant. There are probably several stories for this functionality (you can imagine several business and technical requirements that would be needed to describe this), a number of builds, and at least 20-30 bugs. Wouldn't it be nice if we had a way to relate them somehow – since they are, in reality, related?

So, we provide for the concept of Relationships. RELATIONSHIP describes the relationships we have, and RELATIONSHIPMAP describes the members of a relationship.

RELATIONSHIP

RelationshipID	Name	Comments
1	Schema tweaks	Entities that are related to schema adjustments (such as field lengths) for al...
2	Schema redesigns	Entities that address functional confusions or complete misses in the schem...

RELATIONSHIPMAP

RelationshipMapID	RelationshipID	ThisTest	ThisBug	ThisStory	ThisRelationship
1	1	NULL	1	NULL	NULL
2	1	NULL	2	NULL	NULL
3	1	NULL	3	NULL	NULL
4	1	NULL	4	NULL	NULL
5	1	NULL	5	NULL	NULL
6	1	NULL	6	NULL	NULL
7	1	NULL	7	NULL	NULL
8	1	NULL	8	NULL	NULL
9	1	NULL	9	NULL	NULL
10	1	NULL	10	NULL	NULL
11	1	NULL	11	NULL	NULL
12	1	NULL	12	NULL	NULL
13	2	NULL	NULL	1	NULL
14	2	NULL	NULL	2	NULL
15	2	NULL	NULL	3	NULL
16	2	NULL	NULL	4	NULL
17	2	NULL	NULL	5	NULL
18	2	NULL	13	NULL	NULL

Note that we have allowed for relationships to be related to other relationships. Be one with the zen.

For one to one cause and effect relationships, we provide the concept of an association. Think of a relationship as a group of things happening to share a similar quality. Anything in one relationship can be a part of as many other relationships as it wants. Associations are a special kind of link – they are causal links, meaning they have a direction. We might think of them as a vector, as a vector is a scalar value plus direction. In our case, our value is simply identity (=1), and the direction is implicit in the table in that one thing causes another. A cursory examination of the schema reveals how we establish this relationship:

ASSOCIATEMAP

AssociateID	ThisBug	ThisStory	ThisRelationship	ThisTest	ThatBug	ThatStory	ThatRelationship	ThatTest	Comments
1	13	NULL	NULL	NULL	NULL	2	NULL	NULL	NULL
2	NULL	3	NULL	NULL	14	NULL	NULL	NULL	NULL
3	13	NULL	NULL	NULL	NULL	3	NULL	NULL	NULL
4	NULL	3	NULL	NULL	NULL	NULL	NULL	34	NULL
5	NULL	3	NULL	NULL	NULL	NULL	NULL	35	NULL
6	NULL	3	NULL	NULL	NULL	NULL	NULL	36	NULL
7	NULL	3	NULL	NULL	NULL	NULL	NULL	37	NULL
8	NULL	6	NULL	NULL	NULL	NULL	NULL	1	NULL
9	NULL	6	NULL	NULL	NULL	NULL	NULL	2	NULL
10	NULL	6	NULL	NULL	NULL	NULL	NULL	3	NULL
11	NULL	6	NULL	NULL	NULL	NULL	NULL	4	NULL
12	NULL	6	NULL	NULL	NULL	NULL	NULL	5	NULL
13	NULL	6	NULL	NULL	NULL	NULL	NULL	6	NULL

This mapping will be extremely helpful for our application layer (or a stored procedure in the DB) to calculate a number of metrics and critical path calculations and notifications. If our system knows about how things are related, and what caused other things, it can apply algorithmic knowledge to offer predictability metrics to managers and engineers alike. You can't predict if you're missing data, so gather as much useful data up front as possible without creating a huge inconvenience for the tester and developer.

Given that, you should be able to enter and execute these test rows:

TestID	TestModeID	Name	PriorityID
34	1	Enter in relationships between bugs, bugs and stories	2
35	1	Enter in RELATIONTYPE values of one to one, one to many, many to many, many to one	2
36	1	Revisit Relation schema and create ASSOCIATEMAP, RELATIONSHIP, RELATIONSHIPMAP tables	2
37	1	Create relationships among the common bugs and stories	2

2.9 Fill out Departmental and system Anatomy test data

Go through and enter in some sample data for Department, BusinessUnit, Team, and SubTeam. This will be useful for when we build out our application layer so that we have something a bit more significant and interesting to return versus entering one row.

TestID	TestModeID	Name	PriorityID
38	1	Go through and enter more sample data for Department	2
39	1	Enter more sample data for BusinessUnit	2
40	1	Enter more sample data for Team	2
41	1	Enter more sample data for SubTeam	2

I'll share with you what I entered; you can enter anything you want that makes better sense to you.

DEPARTMENT

DeptID	Desc	RP
1	R&D	2
2	Billing	1
3	Customer Service	1
4	Project Management Office	1
5	Sales	1
6	Facilities	1
7	Vendor Management	1
8	Human Resources	1
9	Events	1
10	Information Technology	1

BUSINESSUNIT

BusinessUnitID	Desc	DeptID	RP
1	Testing Center of Excellence	1	3
2	Emerging Technology	1	1
3	Legacy Support	1	1
4	Data Warehousing	1	1
5	Accounts Receivable	2	1
6	Accounts Payable	2	1
7	Invoice Management	2	1
8	Help Desk	3	1
9	Repair Management	3	1
10	Service Desk	3	1
11	PMO Hiring	4	1
12	PMO Training and Excellence	4	1
13	Sales hiring	5	1
14	Sales management and metrics	5	1
15	Building management	6	1
16	Issue management	6	1
17	HVAC	6	1
18	Furniture	6	1
19	New event management	9	1
20	Safety and Code Assurance	9	1
21	Expense Management	9	1
22	Hiring	8	1
23	Termination Management	8	1
24	Compensation	8	1
25	Benefits	8	1
26	Invoicing	7	1
27	Communication	7	1
28	New Technology	10	1
29	Support Desk	10	1
30	Management	10	1
31	Testing	10	1

TEAM

I obviously focused on a PMO org as a base unit of measure here.

TeamID	Desc	BusinessUnitID	RP
1000	Testing Tools delivery team	1	1
1001	Development	2	1
1002	Development	3	1
1003	Development	4	1
2000	Project Management	1	1
2001	Project Management	2	1
2002	Project Management	3	1
2003	Project Management	4	1
2004	Project Management	5	1
2005	Project Management	6	1
2006	Project Management	7	1
2007	Project Management	8	1
2008	Project Management	9	1
2009	Project Management	10	1
2010	Project Management	11	1
2011	Project Management	12	1
2012	Project Management	13	1
2013	Project Management	14	1
2014	Project Management	15	1
2015	Project Management	16	1
2016	Project Management	17	1
2017	Project Management	18	1
2018	Project Management	19	1
2019	Project Management	20	1
2020	Project Management	21	1
2021	Project Management	22	1
2022	Project Management	23	1
2023	Project Management	24	1
2024	Project Management	25	1
2025	Project Management	26	1
2026	Project Management	27	1
2027	Project Management	28	1
2028	Project Management	29	1
2029	Project Management	30	1
2030	Project Management	31	1

SUBTEAM

SubTeamID	TeamID	Desc
1	1000	OpenTestCenter team
2	2000	Performance Improvement and Feedback
3	2001	Performance Improvement and Feedback
4	2002	Performance Improvement and Feedback
5	2003	Performance Improvement and Feedback
6	2004	Performance Improvement and Feedback
7	2005	Performance Improvement and Feedback
8	2006	Performance Improvement and Feedback
9	2007	Performance Improvement and Feedback
10	2008	Performance Improvement and Feedback
11	2009	Performance Improvement and Feedback
12	2010	Performance Improvement and Feedback
13	2011	Performance Improvement and Feedback
14	2012	Performance Improvement and Feedback
15	2013	Performance Improvement and Feedback
16	2014	Performance Improvement and Feedback
17	2015	Performance Improvement and Feedback
18	2016	Performance Improvement and Feedback
19	2017	Performance Improvement and Feedback
20	2018	Performance Improvement and Feedback
21	2019	Performance Improvement and Feedback
22	2020	Performance Improvement and Feedback
23	2021	Performance Improvement and Feedback
24	2022	Performance Improvement and Feedback
25	2023	Performance Improvement and Feedback
26	2024	Performance Improvement and Feedback
27	2025	Performance Improvement and Feedback
28	2026	Performance Improvement and Feedback
29	2027	Performance Improvement and Feedback
30	2028	Performance Improvement and Feedback
31	2029	Performance Improvement and Feedback
32	2030	Performance Improvement and Feedback

2.10 Map Anatomy through to Tests

Let's do these now:

TestID	TestModeID	Name	PriorityID
42	1	Enter in mapping relationships between Anatomy and tests	2
43	1	Ensure Anatomy and AnatomyDependencies test data are filled in	2
44	1	Enter in a number of rows for AnatomyTestMap	2

In test 42 initially and extended in test 44, we map tests to what Anatomy they cover:

ANATOMYTESTMAP

TestAnatomyMapID	ThisAnatomy	CoveredByThatTest
1	1	1
2	2	2
3	2	3
4	2	4
5	2	5
6	2	6
7	2	7
8	2	8
9	2	9
10	2	10
11	2	11
12	2	12
13	2	13
14	2	14
15	2	15
16	2	16
17	2	17
18	2	18
19	2	19
20	2	20
21	2	21
22	2	22
23	2	23
24	2	24
25	2	25
26	2	26
27	2	27
28	2	28
29	2	29
30	2	30
31	2	31
32	2	32
33	2	33
34	2	34
35	2	35
36	2	36
37	2	37
38	2	38
39	2	39

For test 43, remember we should have our Anatomy defined:

ANATOMY

AnatomyID	ReleaseID	AnatomyName	AnatomyDesc
1	1	Database	Database system: MySQL
2	1	Data model	Data ER diagram representing the model
3	1	Data config	MySQL configuration incl indexes and performance settings
4	1	Application	Java app layer providing QA-centric data input and algorithmic enforcing
5	1	Caching	Caching and metadata precalculations for metrics and system information
6	1	Logging	Application logging layer
7	1	Presentation	User interface
8	2	Database	Database system: MySQL
9	2	Data model	Data ER diagram representing the model
10	2	Data config	MySQL configuration incl indexes and performance settings
11	2	Application	Java app layer providing QA-centric data input and algorithmic enforcing
12	2	Caching	Caching and metadata precalculations for metrics and system information
13	2	Logging	Application logging layer
14	2	Presentation	User interface

Enter the data of how our system anatomy is related to one another:

ANATOMYDEPENDMAP

AnatomyMapID	ThisAnatomyID	ThatAnatomyID
1	3	1
2	3	2
3	4	2
4	5	2
5	4	6
6	7	4
7	10	8
8	10	9
9	11	9
10	12	9
11	11	13
12	14	11

2.11 Map Build, Bugs, and TestSets

Finally, ensure we have test data for a sample build, mappings for a BugFixSet, and a mapping for a TestSetMap:

TestID	TestModeID	Name	PriorityID
45	1	Ensure there is a row in BUILD to represent a past or future build	2
46	1	Enter remaining bugs into BUGTOBUGFIXSETMAP table to presume all existing bugs will be in our curr...	2
47	1	Go through and map the test sets in TESTSETMAP	2

BUILD should look something like:

BuildID	BuildName	RP	DeliveredOn	ExpectedOn	SubTeamID	ShortDesc
100	Initial alpha	1	2012-10-08 19:12:00	2012-10-08 19:12:00	1	NULL

BUGTOBUGFIXSETMAP should look something like:

BugMapID	BugID	BugFixSetID	RP	OpenedOn	ClosedOn	Comments
1	1	1	1	2012-10-08 19:12:00	2012-10-08 19:12:00	Creating map for initial testing
2	2	1	1	2012-10-08 19:12:00	2012-10-08 19:12:00	NULL
3	3	1	1	2012-10-08 19:12:00	NULL	NULL
4	4	1	1	2012-10-08 19:12:00	NULL	NULL
5	5	1	1	2012-10-08 19:12:00	NULL	NULL
6	6	1	1	2012-10-08 19:12:00	NULL	NULL
7	7	1	1	2012-10-08 19:12:00	NULL	NULL
8	8	1	1	NULL	NULL	NULL
9	9	1	1	NULL	NULL	NULL
10	10	1	1	NULL	NULL	NULL
11	11	1	1	NULL	NULL	NULL
12	12	1	1	NULL	NULL	NULL
13	13	1	1	NULL	NULL	NULL
14	14	1	1	NULL	NULL	NULL

And finally TESTSETMAP should capture each test, the set it's placed in, the priority in the set that you've identified, and the execution order. Because we've been loose with our database rules for this set (to allow maximum flexibility as defined by the tester in the application), we can simply enter something like this:

TSMapID	SetID	TestID	TestPriorityInSet	ExecutionOrder
1	1	1	1	0
2	1	2	1	0
3	1	3	1	0
4	1	4	1	0
5	1	5	1	0
6	1	6	1	0
7	1	7	1	0
8	1	8	1	0
9	1	9	1	0
10	1	10	1	0
11	1	11	1	0
12	1	12	1	0
13	1	13	1	0
14	1	14	1	0
15	1	15	1	0
16	1	16	1	0
17	1	17	1	0
18	1	18	1	0
19	1	19	1	0
20	1	20	1	0
21	1	21	1	0
22	1	22	1	0
23	1	23	1	0
24	1	24	1	0
25	1	25	1	0
26	1	26	1	0
27	1	27	1	0
28	1	28	1	0
29	1	29	1	0
30	1	30	1	0
31	1	31	1	0
32	1	32	1	0
33	1	33	1	0
34	1	34	1	0
35	1	35	1	0
36	1	36	1	0
37	1	37	1	0
38	1	38	1	0
39	1	39	1	0
40	1	40	1	0
41	1	41	1	0
42	1	42	1	0
43	1	43	1	0
44	1	44	1	0
45	1	45	1	0
46	1	46	1	0
47	1	47	1	0

2.12 Fini!

You now have a database by and large full of a reasonable amount of realistic test data that should load into any application we build. Our test data will ensure that we can rely on reasonably complete results from the data layer as we experiment with algorithms and presentation with the application.

Have fun with it! Thank you for supporting OpenTestCenter.