Scaling Test Automation using a Metadata-centric Architecture

Donald Maffly

dmaffly@hotmail.com

Abstract

When systems are initially designed, scalability is a consideration that is often overlooked. The first priority is simply to deliver something that meets customer requirements, and everything else is secondary. As a system begins to grow and evolve over time, however, it can become increasingly difficult to meet customer requirements because the system is not able to scale effectively. Lack of scalability creates a bottleneck presenting different obstacles for different functions within a product team (BSAs, Developers, QA, Production Services). For QA it can mean that automated tests become increasingly difficult to maintain, more challenging to trouble shoot, more unruly to navigate, and too time consuming to run.

Refactoring a system to be metadata-centric can effectively address this bottleneck and transform a system into one that is truly scalable. Metadata is data that describes various system components; QA can leverage this information to write generic, metadata-driven automated tests. Consequently, lines of test code are drastically reduced, and ironically, code coverage can actually be dramatically increased. Test code becomes much easier to navigate, to maintain, and to run.

Biography

Donald Maffly works out of Portland Oregon as a QA consultant. He brings over 3 decades of experience as a software developer, test developer and manager. Recent areas of focus have been in business intelligence and data analytics serving both the financial and healthcare industries.

Copyright Donald Maffly 2024

1 Introduction

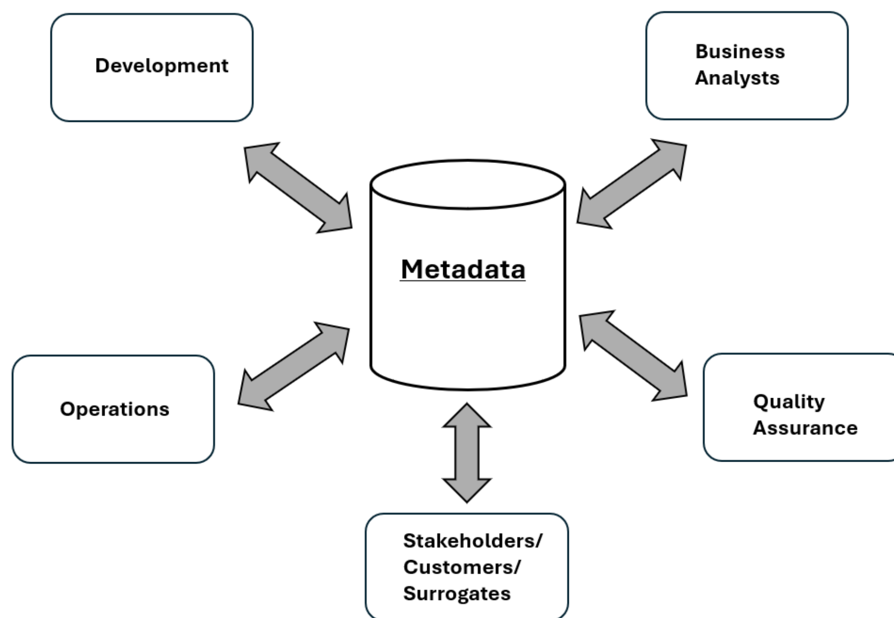
When systems are initially architected, a sometimes narrow set of customer facing requirements can be favored over other internally originating requirements. More often than not, it is likely speed-to-market that trumps all other concerns such as performance, security and scalability to name just a few. As a system grows, however, it can be more difficult to extend and maintain it because initial design was not done with extensibility in mind. This shortcoming rears its head in the form of longer develop times, longer test times, longer operational times and quality may even suffer.

As you dig into code, you might also observe the same old patterns repeat themselves; you might find highly detailed data (e.g. directory paths, file names, etc.) hardcoded into programs. Lines of code can multiply several times over, and with it, code becomes harder to navigate. It may take longer and longer for a system to run through its computations. These are just a few of the telltale signs that a scalability bottleneck is looming.

Part of this is just natural system evolution. Mistakes may have been made earlier that need to be corrected, or perhaps system requirements changed dramatically over the course of time. The team refactors, and the system evolves. That said, regardless of whether it is during upfront system design, or refactoring code years later, sound software engineering principles apply; sound high level architecture, modularized and divided into domain specific abstractions. Common code can be identified and from there consolidated and condensed into a single abstraction. Instead of having thousands of lines of code reinventing the same old wheel, you now have a single piece of code that performs the same task. Lines of code become drastically reduced, the code is easier to understand and to navigate, and ultimately to develop, test and operate.

This is where *metadata* comes in. One way to understand the concept of metadata is as an abstraction mechanism, and as such it is a cornerstone of sound software engineering. If metadata is given the proper visibility, it can become the hub around which an entire organization can anchor itself when undertaking an ambitious engineering endeavor (see figure 1 below). In this paper, we define metadata, explain how to create it, and provide an example illustrating how metadata can be leveraged to benefit QA and the testing effort.

Figure 1.



\

2 Metadata : What is It?

Metadata is hardly the latest new big thing. The *meta* concept has been around for quite a while in computer science circles. **Meta classes** are used in the Smalltalk object-oriented language to create classes of classes to control the computational model of Smalltalk. **Meta languages** (a.k.a. formal grammars) can be used to generate language parsers, *yacc* is good example; it takes a formal grammar as input and generates a parser from it. **Meta programming** is writing programs that generate other programs; these can be as simple as macros, Java generics, C++ templates, or dynamic SQL, but could also be sophisticated AI programs written in Prolog or LISP that are generating programs or even extending themselves. And then there is **metadata**, the best known example might be in a RDB (relational database) context where a data schema (describing table structure, columns, relationships) is captured itself in the form of data (i.e. tables, columns, etc.).

The Oxford English language dictionary defines 'meta' as 'referring to itself or to the conventions of its genre; self-referential'. So strictly speaking metadata is data that describes data. In the context of this paper, by metadata, we mean data that describes the system under test (SUT) *itself* and furthermore, data that describes *tests* targeting the SUT.

Metadata about the SUT *itself* might contain:

- descriptions of system entities (e.g. files, directories, processes)
- relationships between entities
- higher level abstractions that add deeper hierarchical understanding; e.g. classifying entities into logical groupings
- input descriptions
- output descriptions
- system change history

Metadata about *testing* the SUT might contain:

- descriptions of testing entities (e.g. validation scripts, test results files & directories)
- relationships between testing entities and SUT entities
- higher level abstractions that categorize testing entities into logical groupings

Consumers of metadata range anywhere from development engineers (charged with architecting and building the SUT), QA engineers (charged with testing the SUT), Production Services engineers (charged with running the SUT in a production setting), to business analysts (charged with representing end users of the system and coming up with system requirements).

Metadata can be contained in a relational database, but it doesn't necessarily need to be. It could also be contained in structures within a program, or it could be contained in raw text files. Regardless of how and where metadata is stored, what is critical is that the metadata is easily accessible in some form – preferably via a programming API. Once accessed, metadata can be used to:

- drive test automation of the SUT
- inform the development of the SUT
- drive the running of the system in development/test/production environments
- provide understanding of the inner workings of the SUT
- provide a historical record of changes made to the SUT

3 Metadata: Creation

Early in a system's life cycle, it can sometimes be hard to identify metadata, but generally a good place to start, is using the system architecture itself as a guideline. Which system abstractions are central, and ones that you want to publicize to stakeholders, developers, testers, and operators alike? These might

Copies may not be made or distributed for commercial use

be particular processes, input files, or output files. But sometimes it may not be until a system has grown and evolved that metadata opportunities can become identifiable. Only then, may recurring patterns emerge that are suitable to express as metadata.

Recall that one of the benefits of using metadata is to capture data describing the system in a single place, so that you don't have to sift through thousands of lines of code to understand how the system works. Not only can metadaa aid in understanding and navigating the system, but it make modifying the system much easier. When changes can be confined to metadata, often times system changes can be done quickly, and perhaps even dynamically during the operation of the system, not requiring any development or QA effort best case.

To this end, we want to capture metadata information that expresses small differences in different commonly occurring classification of objects in the system that you want to give broader visibility to. These object classifications might be:

- directories where input/output files live
- the structure of filenames
- how the file is used
- the relationship between objects in the system
- when processes are run
- the order that processes are run

Armed with metadata situated in a single place, developers are able write reusable code, testers to write generic test cases, and system operators are able to drive the running of the system.

4 Metadata: an Example

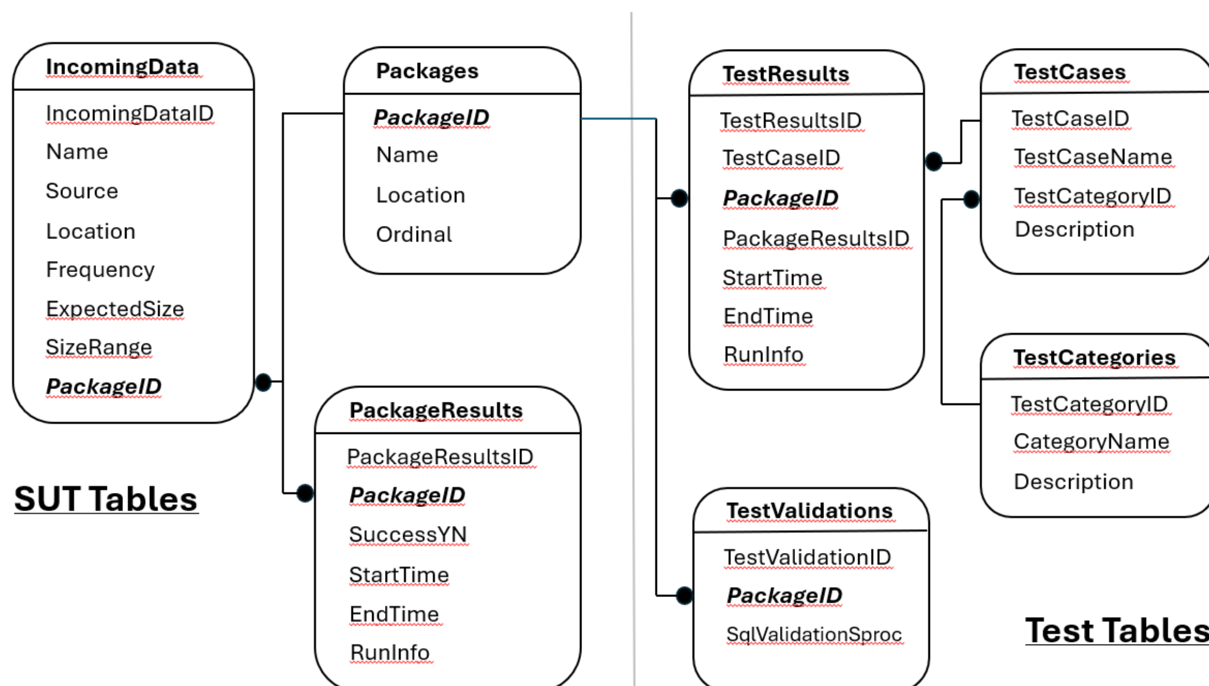
A large regional bank in the Pacific Northwest has a data warehouse system built on the Microsoft stack (SqlServer, SSIS, Azure Dev/Ops). It ingests data from many cooperating financial institutions, and processes it via ETL (Extract, Transform, Load) processes where data goes through several refinement phases. Finally, data lands in a data warehouse, where it can then be served via data visualization tools like Power BI to bank executives to help them make informed business decisions.

This data ingestion process is complex and must be done within a tight time frame bounded on one side by the time the last incoming file arrives and on the other side by the time consumers in the bank need to access the processed data in the data warehouse. Other complexities include

- ordering dependencies exist between ETL processes
- frequency that incoming files/data are delivered vary
- incoming data is delivered in a variety of different formats (flat file, Excel, other databases)
- incoming data format of incoming data changes occasionally
- incoming data from a particular source can straddle multiple files

The entity relationship (ER) diagram (see figure 2) depicts metadata tables which are designed with these requirements in mind. These tables capture essential SUT information that varies between data ingestion processes and as such can be fed in as parameters to generically written code, be it implementation code, test code, or operational code. Furthermore, it enables the system to be easily configurable. Thus, changes do not necessarily need to require a major effort on the part of either development or QA.

Figure 2.



In the ER diagram, observe that SUT tables are organized on the left side, whereas test metadata tables on the right; this separation is intentional, so that test tables can be deployed separately, as they are not something that belong on a production system. Furthermore, dependencies between SUT and Test tables are uni-directional, i.e. Test tables depend on SUT tables, SUT tables do NOT depend on Test tables.

Observe that the *Packages* table is the essential target of the testing effort as well as the overall running of the system, and as such it is centrally located in the ER diagram; foreign key relationships (*PackageID* highlighted in bold italic font) to the *Packages* table exist in four other tables—both SUT tables and test tables.

SUT metadata tables contain metadata that is specific to the SUT; the *Packages* table (i.e. representing SSIS packages) describes peculiarities with the packages such as their name, location and ordering dependencies. The *IncomingData* table describes the different types of data sources that are associated with each package. And the *PackageResults* table are where package results are logged.

Similarly, test metadata tables contain data which is used to inform the test effort and ultimately to drive the running of tests. Table *TestValidations* maps SQL validation scripts (as stored procedures in the database) to be run against particular packages; *TestResults* contains pass/fail results. Higher level testing abstractions are contained in tables *TestCategories* and *TestCases*, which are used to drive the running of the tests.

For the sake of this paper, we have abridged the metadata diagram, so that we could more easily convey the benefits that metadata brings. In actual practice, however, metadata could be more complex. For example, the banks datawarehouse system outputs data in different formats, and we have excluded that here. Also, as far as test metadata is concerned, there are likely to be packages or classes of packages to which a certain test case can be run on—in which case a new package group abstraction (and table) could be added, which would group packages together, and a package grouping foreign key would be added to the *TestCases* table. This also has been omitted from our ER diagram for the sake of simplicity.

Together both SUT and test metadata above create the foundation for which generic test code can be written (in pseudo code) below. The example below demonstrates the coding of a happy path test case to be run against all packages which have run frequency = “Daily”.   This test is written to run all the “daily” packages, but could be reworked to take a *PackageID*, or even a package grouping id, and run a smaller subset of packages:

```
Procedure TestRunPackagesDailyHappyPath(<params>)
{
   Get all SsisPkgs with Frequency='Daily'
   Order these packages per Ordinal value;
   For each pkg in this ordered list of packages
   {
      From IncomingData get input filename and drop directory, and drop it
      Run the package;
      Check for run status in PackageResults;
      Run any validation scripts associated with the pkg;
      Log PASS/FAIL test status in TestResults table
   }
}
```

Here is a failure test case coded to verify that a package behaves as expected (i.e. fails) when an expected incoming file is *not* found:

```
Procedure TestRunPackagesMissingFile(<params>)
{
   Get all SsisPackages with Frequency='Daily';
   Order package in SsisPackages table per Ordinal column;
   For each pkg in SsisPackages
   {
      From IncomingData get drop directory, and remove any files there
      Run the pkg;
      Expect run status error;
      Log PASS/FAIL test status in TestResults table
   }

}
```

Other failure tests that could be written could also be metadata driven. For example, incoming files could be dropped that don’t conform to *ExpectedSize* and *SizeRange* parameters; using the *Ordinal* column, packages could be run in the wrong order. When SUT failures are not encountered during run, this would constitute a test failure.

The big takeaway here is that the tester is relieved from coding separate test cases for each SSIS package, thus reducing code bloat by possibly as much as an order of magnitude. Test code becomes much easier to maintain – particularly when SUT changes are restricted to just the *Packages* and/or *IncomingData* tables. Also, other gains are made by having system metadata and testing metadata co-located. The most obvious of these is during test failure anaylyis (which requires human intervetion), using the *TestResults* table’s direct references to *PackageResults*, we quickly get one step closer to determining root cause..   Furthermore, co-location of system and testing metadata fits hand in glove with agile development practices, where test driven development and pairs programming are the norm. A synergy between development and testing is enabled and productivity gains are to be had.

Copies may not be made or distributed for commercial use

5 Metadata: other benefits to the testing effort

5.1 Gauging Test Coverage

Test coverage metrics provide a critical SDLC function, and act a measuring stick for the completeness of a test effort, and ulitmately for the quality of the SUT itself. Calculating test coverage metrics can be challenging particularly for large, complex systems, and particularly when there do not exist direct links between test cases, test code, and system source code. Traceability from test case to SUT function becomes muddied.

Manually scripted test cases can provide traceability by *name* to a functionality or a given scenario. Also, links between test cases and agile 'stories/epics' provided by collaborative development environments such as Atlassian and Azure DevOps are genuinely very useful. But naming references and links are often too distant from the actual system itself to be entirely credible. Furthermore test results can be output to an entirely different media and need to be parsed and then painstakingly mapped back to SUT functionality to really understand how test results relate to the SUT as a whole.

By contrast, metadata can be used to make more direct, and hence more credible, links between an actual process function in the SUT and actual tests, since the same metadata properties are used to drive both, and they can be documented in a test results table with direct references to metadata entities. If such a test results table lives in a database, then a simple SQL JOIN query between test results and SUT metadata make it is very easy to demonstrate what parts of the SUT were tested as well as which were not.

5.2 Generating Test Plans from Change Sets

With large systems it can be simply too time consuming and ultimately too costly to run entire regression test suites (along with the time to analyze test failures) to be practical. If code changes to a system are narrow in scope for a given production release, then it is reasonable to assume that so too could a corresponding test plan be made narrower, and thus less time consuming and costly to run. For example, revisiting the bank data warehouse system example, suppose a change were made to a single financial institution ingestion module, if metadata existed that mapped system dependences to test cases were in place, a suite of test cases (from the TestCase table) could be computed. Such dependencies could be at a module level, or even down to the smaller granulatity of implementation files (e.g. C#, SQL, etc).

5.3 Generating Test Code using Generative AI

Generative AI tools like ChatGPT have demonstrated that given problem description, it is capable of generating small programs in a wide variety of programming languages. There is no reason to doubt that we are close to the day when generative AI could generate programmed test cases, if that day is not already upon us.

The key to making that work is providing generative AI quality information as input that it needs to code a test case. Metadata fits that bill very nicely; it provides structured data that is easily digestible. Metadata coupled with human language guidance should very easily provide generative AI the direction it needs to generate relevant and useful test case code.

6 Concluding Remarks

At our regional bank, we observed the following before vs. after improvements after QA embraced the metadata concept:

- lines of test automation code were reduced by at least an order of magnitude and hence much less time was spent maintaining the code

Copies may not be made or distributed for commercial use

- test pipelines experienced many fewer false fails, and with that time was saved not going down rabbit holes tracking down root causes

- automation code was much more robust and could easily sustain changes to the underlying SUT without breaking—also a big time saver

While this paper has focused mostly on QA's use of metadata to supercharge the testing effort, it goes without saying that QA does not occur in a vacuum. QA is intimately connected with the rest of an organization; requirements/acceptance criteria authored by BAs inform QA's test case writing, operations and QA share the similar concern of simply running the system, and developers and QA might even share the same cubicle in paired programming agile development situations.

If the language of system metadata becomes the *lingua franca* of these interacting communities, then there becomes a lot less room for misunderstanding, and a lot more opportunity for collaborative and highly productive work. And finally, bringing generative AI tools into this mix has the potential to super-charge this collaboration and the engineering effort as a whole.

References

Goldberg, Adele. 1983. *Smalltalk 80: The Interactive Programming Environment*. Boston: Addison-Wesley Publishing Company (now Pearson).

Levine, John R, Tony Mason, and Doug Brown. 1992. *Lex & yacc*. Sebastopol, CA: O'Reilly and Associates.

Stroustrup, Bjarne. 2013. *The C++ Programming Language*, Sebastopol, CA: Addison-Wesley Professional (O'Reilly).

Abrahams, David. 2004. *The C++ Template Metaprogramming*, Sebastopol, CA: Addison-Wesley Professional (O'Reilly)