Avoiding the Pitfalls of Document-centric Data Management

A DMG Federal eBook

William Munson

April 2013
Abstract

The old-school, "document-centric" process of creating and managing project data in documents (e.g., requirements documents, designs, and test plans) is obsolete. Duplication of effort (lack of reuse), difficulty in automating data maintenance, poor data consistency, and other issues constantly affect data quality, drive up cost, and hinder overall program success. Worse still, project teams become enslaved by documentation tasks and meeting documentation standards, often to the detriment of the technical solution under development. Leveraging new approaches using more modern technologies enables project teams to adopt a data-centric approach and get back on the right path. Documents cease to be the masters of the data and once again become the tools of the data user by:

- Analyzing the product data required by each project discipline;
- Identifying opportunities to reuse information;
- Identifying key data element attributes;
- Assigning ownership of data elements;
- Managing the elements in a database; and
- Using Business Intelligence (BI) tools and/or publishing engines to extract the information in human-readable format.

These factors combine to create real-time reports that are always up to date.

Document-centric Data Management

For decades, project data has been managed using predetermined sequences of documents. These document sequences are required and described by standards such as:

- U.S. Coast Guard Commandant Instruction Manual 5000.10A, "Major Systems Acquisition Manual" (2008-present);
- DoD Directive 5000 series (2000-present) and its supporting documents; and,
- Various commercial System/Software Development Life Cycle (SDLC) standards.

Below is a generic example of what I mean by a predetermined sequence of required documents:

1. A Needs Assessment Report is created to identify a missing capability.
2. A Concept of Operations (CONOPS) document is created to describe how the capability should be provided.
3. Based on the CONOPS, an Operational Requirements Document is created.
4. The operational requirements are further refined and formalized in a Performance Specification.
5. A System Design Document is created that will satisfy the performance requirements.
6. A Test Plan is generated from the operational and performance requirements.
7. Test Procedure Documents are created to define the tests required by the test plan.
8. Test Reports are published to collect the results of the executed test procedures.
9. Meanwhile, more documents are being created to form the documentary portion of the solution's user interface: operators' manuals, maintenance guides, and training materials.

As complex as this may sound, in practice the sequence of documents can become labyrinthine, especially for large systems. The basic sequence of creating requirements—specifications—designs—tests—reports—user interface documents can be repeated each time a portion of a complex system is broken out and assigned to a lower level authority such as a subcontractor. In the end, hundreds or even thousands of pages of documents are created in an attempt to manage the data they contain in the course of even a modest project.

The Pitfalls of Document-centric Data Management

Documents are Expensive

The resources required to create these documents have always been burdensome, despite easing slightly in recent years. In the early 1980s, whole publications departments existed solely to create them. Engineers and programmers (which I'll collectively call "problem solvers" in this paper) created the documents, sometimes in long-hand, passed them to typists (or, later, word processing staff using proprietary, dedicated word processing systems) who keyboarded them and passed them back for proofing. Meanwhile, illustrators at drafting tables created the illustrations needed for the documents. Finally, copy preparation staff put together finished camera-ready copy. In the late 1980s, with the adoption of Personal Computers (PCs) and the market stabilization on a few fairly standard PC-based word processing and graphics packages, the word processing staff and most of the illustrators could be dispensed with: now the problem solvers did the typing and illustrating themselves.¹ For the most part and in most enterprises, little has changed since, despite ongoing problems with this traditional approach.

Traditionally Produced Documents Are Difficult to Maintain

The life span of program documents is longer than many appreciate. Major projects can exist for decades. In the course of maintaining the system, process, and/or product resulting from the project, it is always necessary to refer to and update information developed earlier. This is especially true of the requirements the project was intended to satisfy: each time an improvement is made, testing

¹ In those days there was also general grumbling around the industry about "what became of the productivity payoff promised by the PC revolution?" There was little change in the bottom line because more highly paid staff were now preparing the documents for themselves and spending more hours doing it. The productivity improvements materialized in lower document maintenance effort due to the ease of updating a text or graphics file instead of retyping the page or redrawing the figure, and prettier documents (the more typeset look we now see as opposed to the "typewriter style" of old).
must be performed to ensure that the change works and that it did not break existing functionality. Even with de facto market standard tools like the Microsoft® Office suite, file incompatibilities do occur over time, making access to and update of old documents difficult. Most significantly, even with document templates and style guides, no two users of the same publishing tool use the tool the same way. Some use spaces instead of tabs, manually enter paragraph or figure numbers, or even manually create tables of contents. Updating documents created by different authors can be unreliable and time-consuming.

**Traditionally Produced Documents Do Not Support Traceability**

For many programs, the same information is included in more than one document. Some is simply boilerplate (e.g., the same system overview description often appears in numerous documents) but much is vital and volatile (e.g., requirements that affect products created by multiple subcontractors or design data that impact interfaces with other systems). Once a traditional document is created, there is little visibility into its contents. Knowing which documents are affected by a given change can be difficult. Since project data is interconnected yet developed at different times and often in different locations, the ability to reference data in one document from another document is key (especially when tracing requirements from the concept of operations to a piece-part specification). This can be done using hypertext links within a given word processing package or in a web environment, but these technologies still do not solve the problem.

- Documents are not always accessible to each other, making links useless.
- Since they are based on file locations and file names, links themselves are difficult and tedious to maintain.
- Web pages do not typically translate to hardcopy well, and sometimes hardcopy is needed.
- Even when links are available and working, they are still only pointers: updating reused data still requires following the links and making the same changes over and over again each place the data appears.

**Traditionally Produced Documents Contribute to Duplication of Effort**

If each of six developers has to create a software design document, and each document must contain a system overview paragraph, without additional coordination there is risk that six system overviews will be written. Multiply this by the number of other disciplines (e.g., hardware engineering, system engineering, training, technical documentation, and integrated logistics support) with other deliverables, each of which will likely contain a system overview, and the problem is magnified.

**Traditionally Produced Documents are Burdensome to Problem Solvers**

As we discussed, the PC revolution removed many professional typists and illustrators from the workforce and placed their tasks in the inboxes of the problem solvers. Although I have met some who enjoy writing, many engineers and programmers prefer working with numbers and logic over working with sentences. For others, English is literally a second language and an additional challenge. Either way, if these workers had a passion for creating documents, most of
them would not have majored in engineering in the first place. While many appreciate the necessity of documenting designs or placing comments in their code, many view the imposition and enforcement of the format and content restrictions imposed by traditional document-based development standards to be obnoxious and even unnecessary. They often either give only cursory attention to the documentation portions of their tasks or spend more time and effort than necessary on them. The former approach short-circuits the data management goals of the project, while the latter is simply inefficient. In my experience it takes aggressive management oversight to force problem solvers to comply, and doing so is about as effective as attempting to drive a nail by hitting it with a porcelain vase: the team shatters as problems solvers leave for greener pastures and the documents don’t get created.

**Why the Document-centric Approach Fails**

So the question arises, what are managers trying to achieve by requiring all of this documentation? Historically, the standards that demand the creation of these documents were intended to define a process for developing solutions (e.g., software, hardware, systems, and processes). The document requirements were stated to define the kind of data that must be determined at each point in the process. In theory, the documents themselves were intended to:

1. Communicate developed information to the next downstream effort so that the information could be built upon; and
2. Provide evidence that proper process was followed.

These goals are sensible and fall well within the inherent capabilities of documents. At their core, documents are nothing more than writings that convey, or deliver, information. They deliver information from an upstream author to a downstream user (be it human or machine), so they are suited to purpose 1 above. Since they are written, they can persist well after the information is delivered, so they are also suited to purpose 2.

Unfortunately, in practice, we have demanded far more from documents than they can provide. First, while the processes that demand these documents do not come out and state it, they tend to presume that communication is mostly one-way, from upstream effort to downstream effort. Yes, these processes do have built-in review cycles where readers comment and authors update documents, but they do not readily support the all-too-frequent use case where a change identified while creating a downstream document affects an already-approved document upstream. When this happens, the only support offered by the development process is to require that progress stop while the upstream document is reworked and re-approved. In the real world, there is simply no time for this.

Our difficulty in maintaining traceability in documents is rooted in our second fallacy about them:

1. We rely on the fact that documents persist after the information is delivered for evidence that proper process was followed. Project leadership and quality assurance try to measure program progress by the number of created deliverable documents and data quality by compliance with format, content, and traceability audit results.

---

2 The second tenet of the Manifesto for Agile Software Development relates to this: “We have come to value working software over comprehensive documentation.”
2. We simultaneously fail to appreciate that inherent in this persistence is the characteristic of being time-bound. Like all recordings, documents by their very nature can only reflect data at a given point in time.

For example, an approved Operational Requirements Document (ORD) reflects the requirements at the time it was approved. Typically, though, operational requirements change as tradeoffs are made in the course of solution development. When this happens, the approved ORD is only capable of being an obsolete document. Its only remaining use is to prove that we completed the work that produced it. Instead, since we choose to ignore the fact that documents are time-bound, we update it (or, more often, realize we should update it but don't get around to it). While this might preserve traceability, what else does it accomplish? It may keep the Quality Assurance folks (who are auditing our work by checking our documents) off our backs, but that's about it. The ORD has already done everything it could do: it communicated the approved requirements at a given point in time to its downstream efforts. We insist on updating it because instead of seeing it as a vessel for communicating information at a given point in time, we try to use it as the definitive repository of approved operational requirements. This is the root cause of the shortcomings of traditional document-based information management: it views documents as repositories, and they cannot support this use. Once again, we find ourselves facing a nail, porcelain vase in hand, shards at our feet.

Data-centric Data Management

Clearly, documents are the wrong tool for data management. Despite arguments that they are more user-friendly (i.e., they provide a familiar user interface) and can be created and updated using common tools and skills, these characteristics cannot overcome the simple fact that expecting documents to act as information repositories is asking too much of them.

So what is the right tool to manage data? Information is best stored and maintained in a database. When snapshots of current data (documents) are needed, the information is extracted to create them. Let's take a look at how this can be done.

Overview of the Data-centric Data Management Approach

Defining Data Element Types

Project data can be divided into different data element types. Data element types are classes of data that share common functions and characteristics. Figure 1 shows some basic examples.

![Figure 1: Sample Data Element Types](image)
The System Overview is a textual and/or graphical description of a system. Typically, these are developed early in the program as the CONOPS is developed.

Applicable Documents are documents that pertain to the program such as specifications, contracts, or other externally-maintained documents. Applicable Documents are identified throughout the program lifespan.

Requirements are statements of conditions or capabilities necessary for the system to meet its objectives. In the broadest sense, Requirements define the objectives as well. These are created throughout the development phases of the program, and may be refined and updated throughout the program lifespan.

Test Procedures are sets of actions that must be taken to demonstrate that the system’s requirements have been met. They are (or should be) developed concurrently with the Requirements.

Maintenance Procedures are descriptions of actions that must be taken to keep the system available and operating properly.

An individual instance of a data element type (e.g., an individual requirement or procedure) is called a data element. Data elements can vary in size: they can be anything from a single number or word, figure, and drawing, up to and including complex tables, paragraphs, or even complete documents, depending upon the granularity of data management that is needed for the project. Data elements are stored in the project data repository: a database.

### Identifying Data: Attributes

If all project data is stored in a database, a means must be provided to identify and retrieve the data on demand. This is done by defining attributes (a type of metadata [data about data]) that are common to each data element within a data element type. This is simpler than it might sound. Figure 2, below, identifies possible attributes for each of our sample data element types. Table 1: Sample Attribute Definitions

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Element ID</td>
<td>An identifier unique to the data element.</td>
</tr>
<tr>
<td>Element Owner</td>
<td>Identifies the individual or group with permission to authorize changes to the data element contents. In some cases, the owner may be the author, in other cases, it may be some other authority.</td>
</tr>
<tr>
<td>Project ID</td>
<td>Identifies the project(s) on which that the data element is used.</td>
</tr>
<tr>
<td>Overview Description, Bibliographic Info, Requirement Text, and Procedure Text</td>
<td>Contains the actual text of the item.</td>
</tr>
<tr>
<td>Applicability</td>
<td>Identifies specific documents for which the document is applicable.</td>
</tr>
<tr>
<td>Parent Requirement ID</td>
<td>Traces a requirement to a higher-level requirement or a test procedure to one or more requirements.</td>
</tr>
<tr>
<td>Requirement Author and Procedure Author</td>
<td>Identify the individual who created the requirement or procedure text</td>
</tr>
<tr>
<td>Requirement Type and Test Type</td>
<td>Identifies special classes to which the element may belong (e.g., System Requirement vs. Performance Specification or Performance Test versus Regression Test).</td>
</tr>
</tbody>
</table>
Avoiding the Pitfalls of Document-centric Data Management

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Procedure Title</td>
<td>The title or name of the procedure.</td>
</tr>
<tr>
<td>Approval State</td>
<td>Identifies whether the content of the data element has been approved.</td>
</tr>
</tbody>
</table>

For each data type, a corresponding record type (i.e., table) is configured in the database with the attributes as columns and the data elements as rows.

**Provide Human-readable Document/Deliverable Support**

BI tools and publishing engines enable a database configured in this way to provide support for document generation. When a document is needed, queries are run against the database to extract the needed data. For example:

- An interface requirements document for Project XYZ can be generated by querying the database as follows:
  - Retrieve System Overview Description from System Overview table where Project ID = XYZ and Approval State = Approved.
  - Retrieve Bibliographic Info from List of Applicable Documents table where Project ID = XYZ, Applicability = Interface Requirements Document, and Approval State = Approved.
  - Retrieve Element ID and Requirement Text from Requirements table where Project ID = XYZ, Requirement Type = Interface, and Approval State = Approved.

The results of these queries will provide a system overview, list of applicable documents, and list of approved interface requirements -- the basic building blocks of an interface requirement document.

- A user maintenance manual for the same project can be generated by the following queries:

Retrieve System Overview Description from System Overview table where Project ID = XYZ and Approval State = Approved.

, on the following page, explains the use of each attribute.

![Figure 2: Sample Attributes](image-url)
Table 1: Sample Attribute Definitions

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Element ID</td>
<td>An identifier unique to the data element.</td>
</tr>
<tr>
<td>Element Owner</td>
<td>Identifies the individual or group with permission to authorize changes to the data element contents. In some cases, the owner may be the author, in other cases, it may be some other authority.</td>
</tr>
<tr>
<td>Project ID</td>
<td>Identifies the project(s) on which the data element is used.</td>
</tr>
<tr>
<td>Overview Description, Bibliographic Info, Requirement Text, and Procedure Text</td>
<td>Contains the actual text of the item.</td>
</tr>
<tr>
<td>Applicability</td>
<td>Identifies specific documents for which the document is applicable.</td>
</tr>
<tr>
<td>Parent Requirement ID</td>
<td>Traces a requirement to a higher-level requirement or a test procedure to one or more requirements.</td>
</tr>
<tr>
<td>Requirement Author and Procedure Author</td>
<td>Identify the individual who created the requirement or procedure text</td>
</tr>
<tr>
<td>Requirement Type and Test Type</td>
<td>Identifies special classes to which the element may belong (e.g., System Requirement vs. Performance Specification or Performance Test versus Regression Test).</td>
</tr>
<tr>
<td>Procedure Title</td>
<td>The title or name of the procedure.</td>
</tr>
<tr>
<td>Approval State</td>
<td>Identifies whether the content of the data element has been approved.</td>
</tr>
</tbody>
</table>

For each data type, a corresponding record type (i.e., table) is configured in the database with the attributes as columns and the data elements as rows.

Provide Human-readable Document/Deliverable Support

BI tools and publishing engines enable a database configured in this way to provide support for document generation. When a document is needed, queries are run against the database to extract the needed data. For example:

- An interface requirements document for Project XYZ can be generated by querying the database as follows:
  - Retrieve System Overview Description from System Overview table where Project ID = XYZ and Approval State = Approved.
  - Retrieve Bibliographic Info from List of Applicable Documents table where Project ID = XYZ, Applicability = Interface Requirements Document, and Approval State = Approved.
  - Retrieve Element ID and Requirement Text from Requirements table where Project ID = XYZ, Requirement Type = Interface, and Approval State = Approved.

The results of these queries will provide a system overview, list of applicable documents, and list of approved interface requirements -- the basic building blocks of an interface requirement document.
A user maintenance manual for the same project can be generated by the following queries:

- Retrieve System Overview Description from System Overview table where Project ID = XYZ and Approval State = Approved.
- Retrieve Bibliographic Info from List of Applicable Documents table where Project ID = XYZ, Applicability = User Maintenance Manual, and Approval State = Approved.
- Retrieve Procedure Name and Procedure Text from Maintenance Procedures table where Project ID = XYZ, Maintenance Level = User, and Approval State = Approved.

The results of these queries will provide a system overview, list of applicable documents, and a list of approved user-level maintenance procedures -- the basic content of a user maintenance manual.

Of course, this sample attribute set and these queries would not be detailed enough to generate a ready-to-deliver interface requirements document or depot maintenance manual, but a more robust attribute set and a publishing engine configured with appropriate report types could do so. Even with this simple set, though, the basic set of current, approved data can easily be extracted. As any technical writer can attest, getting the correct data is the most difficult part of creating a document.

**Friendly Data Entry Interface**

Working with data in this way may seem tedious or confusing, but with the proper tool it is actually easier for non-authors than writing a document because it can be form-driven. Database interface/workflow tools can be configured to allow users to enter data via forms tailored to each record type. Figure 3 on the following page depicts a possible submission form for a test procedure record. (A data viewing form would likely look similar.)

**Figure 3: Test Procedure Record Form Mock-up**

<table>
<thead>
<tr>
<th>Test Procedure Record ID 001347912</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project ID</td>
</tr>
<tr>
<td>Parent Reqt</td>
</tr>
<tr>
<td>Test Type</td>
</tr>
<tr>
<td>Procedure Name</td>
</tr>
<tr>
<td>Procedure Text</td>
</tr>
</tbody>
</table>
A test procedure author would summon this form from the tool (perhaps by selecting a pushbutton on the parent requirement's form). The Record ID is automatically assigned by the tool. The author would work with the controls on the form as follows:

- Select the project ID from the Project ID pulldown (in the use case where the form was opened via a pushbutton on the parent requirement record, this would automatically be populated).
- Select the parent requirement ID from the Parent Reqt pulldown (in the use case where the form was opened via a pushbutton on the parent requirement record, this too would automatically be populated).
- Select the test type from the choices on the Test Type pulldown.
- The Owner field could default to the logged in user, but modified via the Owner ID pulldown if appropriate.
- The Author field could default to the logged-in user, but modified via the Author pulldown if needed.
- The name for the procedure would be entered into the Procedure Name field.
- The text for the procedure would be entered into the Procedure Text field (configured to scroll if needed).

**Why Data-centric Data Management Succeeds**

**Ensuring Data Reuse**
Keeping the data in the database means that only one version of the data can exist at a given point in time. This not only encourages data reuse and elimination of duplication of effort, it demands it.

**Enhancing Change Control Support and Traceability**
With the data-centric approach, there is now a single point of control for every occurrence of every data element. For example, if the system overview or a procedure changes, the changes will automatically be reflected in every future iteration of every document generated from the database.

The data interrelationships in a system like this may at first seem to complicate change control, but since the data is managed in a database, change control is actually enhanced. Since each record carries an attribute that identifies its parent, queries can be written to identify all records affected by a change to any given record. By requesting that the owners of the affected records be returned in the query, a list of stakeholders to any given record change can immediately be created. This parent-child tracking also provides top-to-bottom and bottom-to-top traceability.
Increasing Data Consistency and Quality
By recording each data element once and using it everywhere it is needed, a data-centric approach significantly increases data consistency. The same data is used everywhere. It also increases the likelihood of data inaccuracies being identified and corrected early by widening the audience for each data element.

Lower Data Development Burden
The form-driven data entry approach eases problem solvers' documentation workload. They no longer have to worry about meeting documentation standards and can instead focus on what they do best: creating solutions. Depending upon the granularity and detail built into the database and reporting capability, the data-centric approach can automatically produce human-readable documents or even formal deliverables.

Conclusions
The purpose of documents is to communicate data as it exists at a single point in time. They are unequalled in their ability to accomplish this task, however, when we try to bend them to become data repositories, they fail. Advances in information technology have allowed us to develop tools that enable the project data formerly retained in documents to be managed in databases instead. Database and data reporting tools enable the data to be extracted from the database to create up-to-date documents at will. With this data-centric approach, we finally have a tool suited to our task, and can stop "pounding nails with vases".