IDIOMATIC FILEMAKER

Architectural designs, coding styles, and development strategies

that promote successful large-scale FileMaker®[1] solutions

Initial Release July 28, 2014

Revision 1.2 – August 22, 2014

[1] ScaleFM is an independent entity; this paper has not been authorized, sponsored, or otherwise affiliated with FileMaker, Inc. FileMaker is a trademark of FileMaker, Inc., registered in the U.S. and other countries.

Table of Contents

About the Authors

Author: Chris Irvine, chris@threeprong.com

Co-Author: Perren Smith, perrens@gmail.com

Key contributors: Marc Berning, James Sturm, Jeff Leach, Daniel West

Preface

The authors of this paper worked at Dr. Bott for some period of time leading up to 2014. Although we no longer work for Dr. Bott, our experiences there were crucial to the development of the ideas in this paper. We endeavored to create a working environment and company culture at Dr. Bott that inspired innovation, excellence, and creativity among our developers. The company's mission statement articulated the belief that "our success is found through helping others be successful," and in practice, this meant that we often encouraged our developers to try things outside their traditional job role or even outside the realm of the company's specific needs. This strategy not only benefited them personally but it also encouraged them to bring highlights from that work back to the team and share new ways we could improve our craft.

Another secret to our success at Dr. Bott was the implementation of a collaborative teams that worked in tandem on deadline-oriented projects. In this atmosphere, web developers contributed to FileMaker data processing projects, and FileMaker developers offered their unique skills in web service testing and UI. We also created individual metrics and set monthly or quarterly goals to target the seemingly impossible improvements in areas such as code quality, system performance, and employee productivity; some of our biggest metric shifts came in the final days or even hours of the month. The topics from other software domains that affected our style of FileMaker development were extremely diverse, including query optimization, client api libraries, technical debt, doc block w/tags, abstraction, modularity, high availability, user analytics, unit testing, work queues, and unattended recovery.

Many of these practices started as health discussions around the water cooler, evolved into brainstorming sessions, and finally arrived as uploaded photographs of our white board on an internal wiki (it must be true). To be candid, many of our ideas didn't go as well as we first imagined, and most of the good ideas needed a couple of revisions to get things right.

About the Title

Here is a sampling of quotes that reflect the spirit of idiomatic programming that inspires our title, Idiomatic FileMaker:

> *There should be one—and preferably only one—obvious way to do it.*
>
> *Explicit is better than implicit.*
>
> *Readability counts.*
>
> *–Tim Peters.*[2]
>
> *The non-idiomatic ways aren't wrong, but they usually take longer to type and they always take longer to read, for those who know the idioms.*

[2] Peters, Tim, *The Zen of Python*, https://www.python.org/doc/humor/

–Kate Gregory.[3]

At the very least an idiom is language-specific whereas a design pattern strives, or should strive, to be language-agnostic. Going further, idioms are usually conventions for enhancing readability, or are the superior alternative (on some technical merit) when there is more than one way to do something.

–Luc Danton.[4]

Purpose

What we outline in this paper are items that the authors have decided should be done a certain way because of the way FileMaker works (i.e., its syntax and capabilities). Getting a handle on this topic is tricky because FileMaker is so broad (e.g., database, user interface, procedural programing, functional programing, runtime, and client-server concerns). There are many topics we omitted from this paper because we don't feel like we have sufficiently refined them or because we are still debating the best approach. We instead focus on our most refined and beneficial techniques. The topics in this paper are a collection of development methodologies and strategies the authors felt best complimented the strengths of the FileMaker platform as a unique and comprehensive rapid application development tool.

Worst Case Scenario: You, the reader, is confused by the topics, don't face the same problems we did, and therefore, adopt unnecessary techniques that bloat their solutions.

Best Case Scenario: You, the reader, is energized by these topics (either elated or incensed), recognize similarities between the problems we articulate and the problems you face, you allow our innovations to affect your own development style, and trigger constructive exchanges in the developer community.

Caveat: Our primary environment was a homogenous business office with best of breed Macs, modern FileMaker, very fast LAN, and an extremely fast production server. The mileage you get out of Idiomatic Filemaker certainly may vary if you are focused on mobile development (with slower client CPUs), WAN access, hosting, or *that other operating system*. Indeed, when we start mobile or time-limited projects, there are certainly decisions that need to be made in terms of short-term compromises and long-term performance and maintainability.

Multi-tiered Development

Every FileMaker solution starts with a file in the operating system that contains all of the elements of the database schema—tables, fields, scripts, a relationship graph, and much more. There are various strategies that increase performance and the ease of development by using multiple files. The most common approach is to employ what is known as the "Separation Model." This model uses two files, one to hold the data and another to hold the user interface elements (layouts) and business logic (scripting). Multi-tiered development takes this approach and expands it even further. We would even go so far as to describe the traditional separation model as one specific solution to a specific deployment limitation of the environment, whereas multi-tiered development is a software architecture design pattern. In our following examples we talk about typical applications that utilize multiple files, but in reality you could apply many of these principles with a different specific approach, even a single file solution.

---

[3] Gregory, Kate, http://programmers.stackexchange.com/questions/94563/what-is-idiomatic

[4] Danton, Luc, http://programmers.stackexchange.com/questions/106815/difference-between-idiom-and-design-pattern

Applying a Multi-tiered Paradigm

The multi-tiered approach takes logical groupings of schema components and organizes them in a way that also provides a method for updates to the solution UI and logic without having to replace the data file. This approach also reduces redundant code and isolates functional areas of a system for departments within an organization.

The simplest multi-tiered model contains three files: UI, business logic, and table storage. The main difference between this model and a traditional separation model is that core transactional scripts are contained in the business logic or middle file. The business logic file contains no tables or interface layouts; it only has table occurrences in the relationship graph and blank layouts for the purpose of supporting the scripts it contains. Ultimately, the business logic file is very light weight and is easy to replace with new versions as the development of a solution progresses.

The three-file model scripts that are tied to triggers or that require interactivity with the user are still contained in the interface file. Due to the tight coupling of schema elements in FileMaker, it makes sense to keep interactive scripts next to their coupled companions, the layouts that comprise the interface.

Solution with 5 Files

This three-file model can be taken even further by continuing to partition the functional components of a solution. Most solutions will contain some sort of a common logging script with a corresponding logging table. For ease of development, sometimes it makes sense to entirely isolate all logging schema elements, including both storage tables and scripts, into their own file. This provides a single point of coupling no matter where logging may be used in the system. It also avoids increasing the file size of solution table storage files just for the sake of maintaining a log of activities.

Another dimension of the five-file solution is what is known as a helper file. The helper file provides a way to write modular pieces of code that the developer can guarantee will not interfere with the context of other scripts in the interface or business logic file. Given that the context in FileMaker is typically defined as the current window (and its associated layout), traditionally the only way to preserve a user's context when running a script is to create a new window, do some task, return the results, and then close the window. This pattern typically emerges when the script being run requires a context that differs from the user's current context. One example might be a script that creates a new order for the current customer record. If the developer moves the "new order for customer" script to the helper file, the coupling to the current customer context is removed by way of its being housed in a different file.

This context separation can even be used as it relates to transactional scripts in the business logic file. Sometimes scripts are created that are working with multiples contexts for a single task. Of course, simply opening multiple windows allows for this to take place in an easy way, but in recent versions of FileMaker on the Mac there is a performance penalty with each time the new window script step is called. During the lifetime of a connected client, this operation will take a bit longer each time. Eventually, this time adds up to lackluster performance until the FileMaker client is restarted.[5] By carefully crafting scripts in the helper file that perform a single task without requiring the new window script step this problem can be mitigated.

Solution with 3+n Files

As a solution matures, this model scales. For instance, if an organization grows and departments begin to need highly differing interfaces, the developer can begin to make multiple interface files. The same goes for partitioning of the

[5] Performance Tuning in FileMaker v12, http://www.drbott.net/blog/?p=1205

table storage files: if a single table storage file grows to tens of gigabytes in size, it is reasonable to segment it into multiple table storage files to keep any single file from becoming too large to reasonably maintain or backup in a quick manner.

FileMaker also supports external client connectivity via the custom web publishing gateway. If the amount of code required to support external client connectivity becomes large, it makes sense to carve the necessary layouts and scripts out into their own files. This is helpful when it comes to understanding at a glance which components of a solution are served to external entities. One simply has to look at the layouts and fields therein to know what is being used instead of hunting external code to review each and every line. That is, when standard user layouts are used by external entities, sometimes it is hard to know without deep investigation. Lastly, when externally accessed components of a system are siloed into a single file, managing security becomes just that much easier, as security policy is not spread all over the solution.

Additional Advantages

Unfortunately, it is a reality that servers crash—yes, even FileMaker servers. When this happens, sometimes it is necessary to recover the solution files, as they may have been damaged in the process. If the multi-tiered file approach is employed, this event is now easier and quicker to recover from. Instead of waiting for a single file that may contain gigabytes of data in dozens of tables, now recovery can be narrowed down to only a fraction of the solution. This allows users to get back to work quicker and reduces the costs associated with the disaster event.

Loose Coupling

Every seasoned FileMaker developer has encountered challenges related to tightly coupled code and has likely developed solutions for those types of problems. A careful assessment of coupling within FileMaker scripts sheds light on the pros and cons of the options available to the developer and will likely bring you to a conclusion much like our own. Let's introduce a little background on coupling with a real-world example of two extremes.

THE HUMAN ARM	APPLIANCE & OUTLET
Tight coupling	Loose coupling
Balance crucial	Imbalance expected
Modular replacement hard	Plug and play
Unit testing requires a whole body	Test fixtures have minimal requirements
Combining work from multiple developers tricky	Simple agreements allow division of labor
+Harmony +Efficiency	+Upgradable +Specific

FileMaker Scripts and Coupling

In FileMaker, the spectrum of coupling has pretty much everything to do with the use of both Script Parameters and Results or the lack thereof.

ex. Tightly Coupled Script

Window Context

- Layout (TO & Table)
- Found Set & Current Record
- Window Mode (Browse)

Perform Script [“Authorize Customer”]

```
# assume everything is setup for us
Set Field [Customer::IsAuthorized; True]
```

ex. Loosely Coupled Script

Irrelevant Window Context

- Layout (could be any)
- Found Set & Current Record (doesn’t matter)
- Window Mode (any mode is fine)

Perform Script [“Authorize Customer”; Parameter: “customer_id=4377”]

```
# parse script parameters...
# ...
# setup context for customer table
# ...
New Window [Name: "Authorizer"; Style: Document]
Enter Find Mode []
Go to Layout ["Customer" (Customer)]
Set Field [Customer::ID; $customer_id]
Perform Find []
#
Set Field [Customer::IsAuthorized; True]
#
# cleanup & exit...
# ...
```

The tightly coupled script is the stereotypical solution from a new developer who is automating a user's workflow because it is the simple and rapidly developed answer for the workflow at hand. These scripts are the quickest to write and they certainly serve their purpose. They can also be effective, handling errors or unexpected executions.

The loosely coupled script, however, accomplishes the same task but in a way that does not access or use information from the calling context. Developers often start moving in this direction when they start to see operations being reused or accessed from different forms throughout their system.

Parameter Passing Options

A - No Parameter

The first option is to not pass any parameter at all, relying completely on window context, global variables, or data stored in records. This option would be on the tightly coupled end of the spectrum.

```
Perform Script [“Authorize Customer”]
```

B - Single Simple Parameter

The next option is to pass a single **simple typed** parameter, such as text or a number, to a script. This option may move one away from tight coupling. Unfortunately, a single parameter is not usually enough for complex systems unless you fall back on additional information access patterns as above.

```
Perform Script [“Authorize Customer”; Parameter: “4377”]
```

C - Multiple Parameters - Positional

A slightly more sophisticated method is the common approach of encoding **multiple positional** parameters. This is consistent with FileMaker's custom function system or any other C-like programing language. This mechanism requires the designer to author documentation specifying the meaning of each argument position. It is possible to use defaults or variable length arguments. One significant challenge with this approach has to do with readability of the code. Here is a simple and a complex example, both using the positional value list format as the preferred encoding method.

```
Perform Script [“adjustWindow” ; Parameter: List ( 200; 122; 50; 111 ) ]
Perform Script [“customerLookup” ; Parameter: List(1; "o=My Company, c=US"; "sn=Smith*"; ""; ""; 200; 10;
"NEVER") ]
```

D - Multiple Parameters - Let Notation

Another popular option for passing parameters, especially when you start to need more than a few, is to use one of the various name-value-pair techniques. Combining a Name and a Value improves readability over strictly positional parameters. This also opens the door for flexible ordering and even optional parameters.

This “Let Notation” approach leverages our ability to execute snips of text as if they were code by using the native Evaluate() function. This is strongly related to our next option except that the Let Notation method puts the called script at risk of executing arbitrary code with unintended consequences. This approach has some specific merits, such

as performance, but it is contrary to Defensive Programing techniques. When using this method, there are some well-designed add-on functions to help in the construction and, later, the safety checking parameters.[6]

```
Perform Script [“customerLookup” ; Parameter: “$last = \”Smith*\” ;¶$alias_mode = \”NEVER\” ;¶”]
```

E - Multiple Parameters - Name Value Pairs (NVPs)

This technique is similar to the “Let Notation” method except that custom syntax is developed such that the string of parameters is compact and readable by humans. Additionally, the string is never executed. Instead, it is necessary to add on some functions to parse and retrieve specific values. Similar to array accessors or dictionary getters, these functions mean you typically only retrieve the values you need and expect. Unexpected names(keys) are never ingested. Problem values are typically manually checked to be within acceptable norms.

```
Perform Script [“adjustWindow” ; Parameter: List ( “top=200” ;
“width=122” ;
“left=50” ;
“height=111” ) ]
Perform Script [“customerLookup” ; Parameter: List( "connection_id=1" ;
"search_base=o=My Company, c=US" ;
"filter=sn=Smith*" ;
"alias_mode=NEVER" ;
"size_limit=200" ;
"time_limit=10" ) ]
```

Credit for this pattern is in part thanks to the work of others in the community. There are multiple NVP implementations that are very good. A few of our favorites are the following:

- PropertyList (Dr. Bott's own)[7]
- Six-fried-rice[8]

[6] Functions for “Let Notation” style parameters, http://filemakerstandards.org/pages/viewpage.action?pageId=557462

[7] *Name-Value Options for FileMaker Compared*, http://www.drbott.net/blog/?p=1320

[8] *Passing Multiple Parameters*, http://sixfriedrice.com/wp/passing-multiple-parameters-to-scripts-advanced/

F - Nested Parameter Encoding

Going even farther with named parameters allows one to organize hierarchies of information into logical groups. You might use this model to convey the state of some system, a structured report, or the results of multiple chained operations. In the industry, these types of payloads are often expressed as JSON or XML.

```
Perform Script [“displayReport” ; Parameter: List ( "criteria=ficsal_period=2007-Q3\¶
      title=Internal Sales Report\¶¶
sales=invoiced=342.99\¶
      cost=221.11\¶
      count=118\¶¶
credits=invoiced=-45.11\¶
      cost=-33.33\¶
      count=4\¶¶" ) ]
```

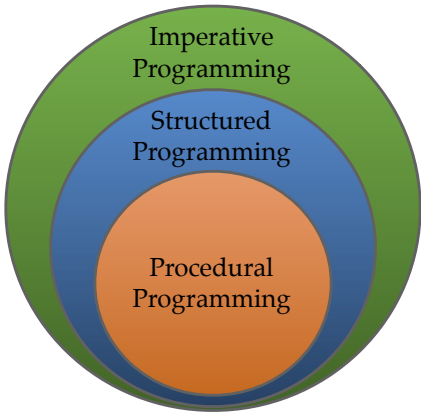
Or you might use a custom function to help with the assembly and some extra white space for readability:

```
Perform Script [“displayReport” ; Parameter: List (
#( "criteria” ;  List (       #( “ficsal_period” ; “2007-Q3” ) ;
                              #( “title” ;  “Internal Sales Report” ) )
) ;
#( “sales” ;  List (                    #( “invoiced” ; 342.99 ) ;
                              #( “cost” ; 221.11 ) ;
                              #( “count” ; 118 ) )
) ;
#( “credits” ; List (         #( “invoiced” ; -45.11 ) ;
                              #( “cost” ; -33.33 ) ;
                              #( “count” ; 4 ) )
); )]
```

Procedural Programming

There is a large body of material on the history and formation of imperative programming, structured programming, and procedural programming. Procedural programming's use of subroutines allows for many advantageous characteristics including the following:

- decomposition (breaking complex problems into simpler ones)
- reducing duplicate code
- reuse of code
- dividing large tasks among multiple developers or project phases
- hiding implementation details from users
- improving traceability



FileMaker provides two procedural programing environments. When developers are faced with a problem that requires reusable code, recursion, or some other procedural characteristic, they often gravitate toward custom functions because of their functional paradigm and multiple parameter support. Unfortunately, it seems that we often abandon this methodology when switching gears to FileMaker's other procedural environment, Scripts. In fact, well-designed FileMaker scripts exemplify all of the above characteristics along with recursion and reentrancy.

FileMaker's call stack, seen in the lower part of the debugger, helps to visualize how each called script has its own private scope for variables, a parameter, and previously executing scripts that unwind later. There are a few aspects of script writing that can unravel some of the procedural programing advantages, such as the following:

- Using global variables to pass arguments – This creates documentation problems for the arguments and data types that serve as inputs to the routine. This is also counterproductive for recursive and reentrant algorithms. The ideal alternative is to pass every argument inside the script parameters using your favorite dictionary/hash technique.
- Using "context" (windows and records) as a way of transferring state to a sub-script – This has similar disadvantages to the use of global variables for passing arguments. Even when the script is called by a button, often all state information can be encapsulated and passed via a parameter.
- Permanently altering context during the execution of a script – FileMaker, upon exiting a script, automatically restores the script state (e.g., local variables). However, the "context" is not restored. Some effort from the script designer is required to put things back to the way they were before exiting.

Scope is the term we use to describe "what can be accessed from here." We most commonly use the term when we talk about a local variable. When we switch to another script, a previously assigned local variable is now "out of scope." Every successful Perform Script step creates a new scope. A global variable may also become out of scope when you switch files.

Context is the term we use to describe all of the state information that a user or program has established in a window. One could almost use the terms *context* and *window* interchangeably. Context includes things like the found set and current record. One way to preserve context is to open a new window to do some work and then close it when you are done. Or one could even maintain 2 or 3 different contexts in different windows.

Naming Scripts

A great way to increase your use of procedural scripts is to adopt some script naming conventions that reflect corresponding behavioral rules. For example, we defined a class of script that we tagged with SF (scripted function) in the name. Those scripts promised to have similar behavior characteristics to a custom function in that they never alter database records, never alter context (or at least restore it). Additionally, we chose that scripted functions be safely called from any context. These SF scripts certainly might use recursion to call themselves, just like a custom function. They could be called (or reused) by any other script at any other time, or from any other file.

Parameters and Results

Another important convention to adopt is a standard but flexible pattern for script parameters, results, and their documentation. We will begin by describing script results because implementing a simple pattern was greatly beneficial

to us. After observing that every UNIX utility utilizes an "exit status" to make chaining commands easier to perform, we copied their model. Pivoting for FileMaker's error code mentality, we decided that every one of our scripts would exit with some script result. The default non-error result was "error=0¶message=Ok" with the error code being mandatory. Any script that encountered an unexpected error would return the native error number. Developers were encouraged to augment any error with a message that would eventually be read by some end user. Most scripts augmented the result pattern with additional outputs, such as "error=0¶message=donut¶sales_for_rep=7667.42¶region=JPN". If that same script encountered an error, the result might have been "error=401¶message=Rep not found". Under this paradigm, script parameters (inputs) wouldn't necessarily have mandatory elements, but they would follow a similar convention for documentation, typing, and optionals. Here is an example of the documentation pattern we adopted:

```
#@param num $rep_id (req): the ID of the rep
#@param bool $all_regions (opt): when set, total sales in all regions, defaults to rep's assigned region
#@param text $month (opt): fiscal month to update, such as "2014-M3", defaults to current month
#
#@return num $error (req): non-zero indicates a problem
#@return text $message (cond): Human readable message about the general outcome of the script. Required if error.
#@return num $sales_for_rep (req): total sales for that rep for that fiscal month
#@return text $region (opt): if a region is assigned, will be 3 char iso country code
```

Notice in the above example how we have accomplished all of the procedural ideals:

- This script might be part of solving a larger reporting program or it might be broken into submodules that are not our concern.
- Iterative enhancement is possible. Duplicated code can be avoided by simply adding a few more optional parameters that alter the behavior of the script without duplicating it.
- The routine is 100% reusable and can be called from any context.
- This could easily be one developer's contribution to a larger initiative, with it being very simple to defensively validate the documented parameters.
- The implementation details can be easily off-limits to the caller, as everything is clearly defined in the parameter/return documentation.
- Debugging can be done easily with the built-in tools (unlike custom functions). We can even see an easy path to build some unit tests for this script.

Single Exit

As developers on our team adopted the standard script result, a couple of new programing idioms quickly took hold in almost all of our scripts. We found that we usually wanted to accumulate all of our script results and include a single Exit Script step at the end. The challenge was that we needed to be prepared to handle problems that might arise at various stages of the script. Many other languages use a "try" control structure. Here is a similar pattern that we migrated toward:

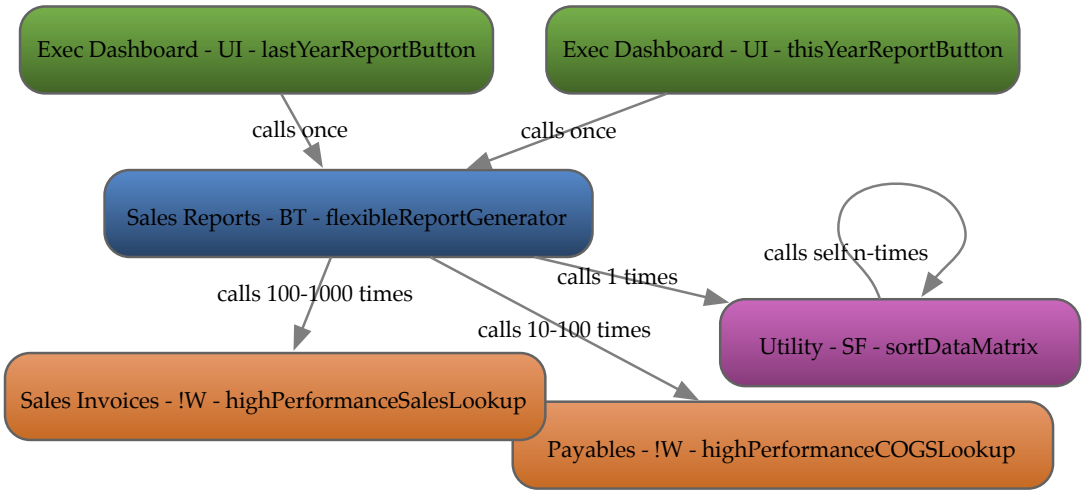
```
# Default is no error
Set [$error ; 0]
Set [$message ; "donut"]

# Think "Try"
Loop
	#...
	#{PARAMETER VALIDATIONS}
	If [$required_param = Bad]
		Set [$error ; -99]
		Set [$message ; "Some bad thing..."]
		Exit Loop If [True]
	End If
	#...
	#...
	#{DATABASE OPERATION-N}
	Set [$error ; Get ( LastError )]
	If [$error]
		Set [$message ; "Operation failed when..."]
		#{CLEANUP/REVERT}
		Exit Loop If [True]
	End If
	#...
	#{FINAL COMPUTATIONS}
	#...
	#End of the block, make sure this loop doesn't run forever
	Exit Loop If [True]
End Loop

Exit Script [List ( "error=" & $error ; "message=" & $message ; "thing=" & $thing)]
```

Additional Script Naming Ideas

Scripts don't all need to conform to the SF behavior rules. Some types of operations, for example, require creating or altering records in your database. To this end, we created another standard script label, BT. Building on the SF concepts, a BT was allowed to alter database records, but the user interface was still off limits (i.e., no dialog boxes or permanent alterations to context). And in the case of our business, we found it important to be very defensive, expecting the unexpected from callers. We also attempted to make operations atomic, such that any error encountered

during a transaction would not leave the database inconsistent with our business design. (Luckily for us, these scripts were typically server safe, meaning that we had a large library of ready-to-use code when FileMaker 13 introduced PSOS.)

Finally, at some point, feedback needs to be provided to an end user in the window or dialog. For scripts that were designed to present user interface, we would apply the UI tag to the script name. (We are proponents of a helper file for performance turning, and thus we adopted a !W tag. This script type was akin to the BT policy, except sub-calls were restricted to avoid concurrent contexts.)



Conclusion

Tell me again, why shouldn't I just write my report routine in a giant monolithic script? Well, if the common advantages above are not enough to win you over, then there are a couple of other key benefits to consider: Maintainable code and Performance (see those specific sections els ware in this paper). Extremely large scripts are hard to maintain, especially when compared to small, contained modules. Extremely large scripts are also very hard to tune for performance. When problems are divided, it is much easier to conquer the sections that are particularly slow.

Least Privilege

We'd like to hear more frequent discussion about this particular security practice in the FileMaker community. Even though many developers are required to include some security controls in their solution, they are often only in reaction to particular problems rather than proactive controls set up during the initial design. If this concept of least privilege isn't ringing a bell, you have probably already seen it in systems like the Apache web server or in your firewall configuration.

```
#Block some bad guys that were giving us grief
Order allow,deny
Allow from all
Deny from 192.168
Deny from bad.guy.net

#or, designed with a least privilege mindset
Order deny,allow
Deny from all
Allow from 10.10
Allow from partner1.com partner2.com
```

We've seen this principle, always encouraged by security specialists, gain popularity in wider circles. For example, users are now often encouraged to operate their computer with reduced privileges to reduce the potential impact of malware. Another example is the growing popularity of sudo over root accounts.[9]

From a practical business perspective, it is extremely difficult to reign in security on a FileMaker solution that was first deployed with controls wide open. If and when you do tighten security, solutions often break or malfunction in unpredictable ways (often so badly that the security controls are just rolled back).

Our strategy when starting a new file or project area is to make the "default" access the minimum level for which any staff member should be granted access. Depending on the topic of the project, we might set something up like this:

No Guest Access
"Staff" Privilege Set with:
    Scripts - execute only
    Layouts - view only
    Records - view yew, edit no, create yes, delete no
    Menus - minimal

As the solution is being built, the developer needs to decide which users on the system should be granted additional rights. For example, when building a system where accountants and managers are going to be editing data, we might create two new privilege sets for those groups of employees, each with expanded menus and the ability to edit records. (Using a directory with external authentication is really the way to go for any solutions with many files or users.)

There is some additional work upfront for developers when building this type of system because they should test the solution under each of the possible privilege sets. Scripts should sufficiently detect and provide feedback to users attempting to use the system with insufficient access.

Another practical technique to improve security as well as data integrity is to design solutions where your most critical tables are only altered by a limited set of scripts. For example, to improve integrity with our inventory tables, we define three transactions (receiving, shipping, and shrink) that are the *only* permissible ways to edit any of those records. Each of those transactions corresponds to a reviewed and tested script that is responsible for enforcing our business rules. To accomplish this, we use the record "edit limited…" capability, combined with an expression that

[9] *Sudo in a Nutshell*, http://www.sudo.ws/sudo/intro.html

uses Get (ScriptName) or ~ObjectID().[10] Other developers might work around this constraint using the "Run script with full access privileges" strategy, but that is a less viable option when you decide to separate your solution into multiple files.

```
[
        /* edits on inventory table limited to the inventory trifecta */
        Get ( ScriptName ) = "Inventory - BT - receiptOfGoods" // receiving
                or
        Get ( ScriptName ) = "Inventory - BT - shipInventory" // shipping
                or
        Get ( ScriptName ) = "Inventory - BT - adjustInventory" // shrink
]
```

Caveat #1 - Any user who has rights to rename scripts could use this technique to gain unauthorized access. Thus, depending upon how open your system is this might not be your best option. Use ~ObejectID() in this situation.

Caveat #2 - It is easy to get burned if you go so far as to limit default access to things such that some items cannot be be viewed. Good security systems will neither confirm nor deny the existence of an item that a person doesn't have rights to view. It is hard to detect errors in data when you can't tell if they exist. Consider this example: net_sales = SUM (sales) - SUM (credits). If a user's access level doesn't permit them to see some or all of the credits, then this might result in a falsified result that might go undetected.

Caveat #3 - Some developers in the community have promoted abstracted utility scripts to handle raw CRUD operations. While there are some clear benefits to doing this, FileMaker doesn't currently provide a mechanism that limits which scripts can call other scripts. As such, CRUD scripts can make it hard to regulate edits that might violate your business rules. We've often found that coupling business rules into the scripts that perform the actual edits, often across multiple tables, allows us to better address conflicts and rollbacks more appropriately.

Finally, a note on developers using a [Full Access] account. This is actually something we've grown to discourage when developing on mature systems. Early on in a solution's inception, Full Access is needed all the time to design the database architecture. However, once a system has grown in size and is deployed into production, these types of low-level changes should become much less frequent. To avoid accidental munging of data or schema, it is recommended that FileMaker developers set up a second user account to voluntarily limit some types of access. This account, for day-to-day development, would have access to edit scripts, layouts, etc. It wouldn't have access to alter tables or the graph, and it also might not have access to delete records. (This is a practical example of applying the principle of least privilege to yourself.) While working in this limited account, if you encounter something that you need, simply use another instance of FileMaker to briefly log in and make the necessary changes. Subconsciously, this makes low-level changes more deliberate. There are a few downsides to this approach based on the current capabilities of FileMaker, such as custom functions or charting. Adopting this technique may impact your coding preferences, such as reduced usage of custom functions.

---

[10] ObjectID function based on code from Fabrice Nordmann, http://www.1-more-thing.com/FileMaker-Avoid-Hard-coding.html

Performance

This section is coming soon. This topic is a big one and it didn't make it in to the first edition of our paper. We've got a lot of good stuff on this and plan to come back to it.

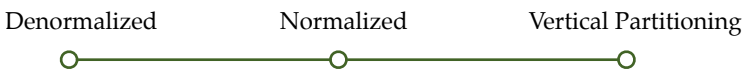
Vertical Partitioning

If you are not already familiar with this or the related terms you will want to do a little background reading. Our normal operation within FileMaker are using "Normalized" related tables with the general idea that every unique piece of information is stored just once with entities (e.g., Customers, Products, Invoices) all related to each other by some sort of ID. A record can 'reference' the truth on some other entity by way of the Foreign ID.

Types of Partitioning

Horizontal partitioning, typically called *sharding*, is the idea of breaking up big tables into more manageable chunks. The simple example here is a phone book that is broken into one volume for A-M and a second volume for N-Z.

Over the last couple years, there has been a rebirth and growing popularity of intentionally *denormalized* databases (such as CouchDB or Mongo) that push towards a different model for data that duplicates information for convenience, simplified database design, or other benefits. The concept of Vertical Partitioning goes in the opposite direction of denormalizing and instead breaks up the columns of a table into one or more tables.



Impacts on Record Access

Most database systems, FileMaker included, treat a record as the smallest granular object. By this we mean that whenever you read a field from a record, the entire stored record is fetched from the server. Additionally, the most flexible security controls are evaluated at the record level. Similarly, FileMaker locks 1 record at a time in a multi-user environment. Furthermore, it is common to see a table definition grow to tens or even hundreds of fields. There may be a few valid reasons to separate what might traditionally seem like a single table into multiple tables (i.e., vertical partitions).

Performance

Many of us have encountered tables that seem to have way to many fields. However, a main concern with this has to do with the actual (or average) "stored" size of a record, typically described as record width. (In FileMaker, unstored calculations and containers seem to be processed independently of a basic record access.) Any operation that requires fetching 1 or more records, such as browsing records, sorting, or go to related records (even across a table occurrence), can be negatively impacted by wide records.

Security

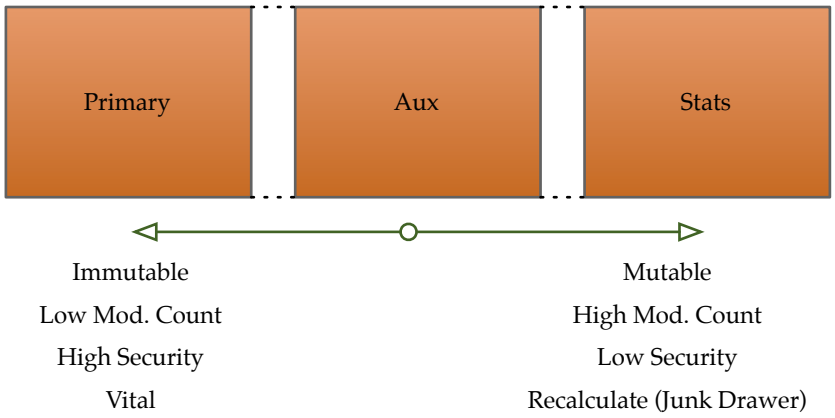
Multi-user and business solutions often require multiple tiers of access or edit controls on a single entity. A simple example might be a sales invoice that goes through a few phases of existence. In an early phase, broad edit access to an invoice is necessary to create it and populate all the necessary information. Once issued, a sales invoice then needs to have all its vital data locked. In a second phase, standard operations likely need to be performed on an in-

voice, such as accepting payments that adjust the invoice's unpaid balance. In a final phase, an invoice essentially needs to be immutable, but users might still want to tag, classify, or group records in order to build sales reports. Report requirements certainly might be changed months after an invoice should have been "locked."

Collision Avoidance

Building a robust database solution certainly requires detecting and appropriately handling situations when a record can't be modified because another user is already editing it. Because an entire record is locked when a user is editing a single field, there are situations where you can choose to partition entities such that different users can be editing different parts of an entity without interfering with each other. For example, you might choose to partition a product catalog table such that pricing information is separate from marketing descriptions, allowing two different employees to perform work on the same set of products without encountering collisions.

It is never advisable to require a specific relational database design because different models function better or worse depending on a specific use case. However, we did often find that we tended to repeat a specific pattern for some of our larger or complex entities. It involved breaking a design of a table into 3 typical vertical partitions. The first partition was core, the least mutable, most secure. The second partition was more mutable and less secure. The third partition was the least secure and sometimes even volatile.



In the sales invoice example above, we would have called first (core) partition *Invoice*. It was only editable for about 12 hours after its creation timestamp and only editable by a small set of authorized employees. The second partition would have been called *Invoice_Aux*. It would have been editable by a larger group of employees and remain editable for a longer period of time, maybe even indefinitely. The third partition would have been called *Invoice_Stats*. It would have likely been editable by just about any staff member and would often contain data, such as statistics, that could have been completely recomputed in the case of data loss. Any time a developer needed to add a field to the Invoice entity, they would make a design decision, as to which of the three partitions would be most appropriate for this field. This generalized 3-partition approach positively addresses both performance and security. However, it seems that customized partitioning decisions are still necessary to promote collision avoidance.

Caveat: Calculation fields that are based on adjacent fields from that same table can be stored. If you start moving calculations or dependencies to other tables, those calculations can no longer be stored or indexed. If not considered carefully, this could result in worse performance then a non-partitioned table.

Maintainable Code

Maintainability encompasses many topic areas, more than we can cover here. Everyone seems to agree that it is obvious when you encounter rotten, hard to maintain code. There also seems to be some consensus on two common criteria related to measuring maintainability, the time it takes to make a change and the risk that the change will break something. In FileMaker, since the system is a full stack (data storage, relational design, coding, and user interface), implementations selected by developers when designing a system can affect any one of these areas.

This is a great time to address a misconception about creating and adopting coding standards. If you haven't used them before, you might feel that they will box you in, slow you down, or limit your creativity in problem solving. From practical experience, we suggest that the opposite is typically true. Using a FileMaker example, when you adopt a standard "record looping" idiom, you actually write code faster because you are repeating something that is familiar. It actually gives you more time to think over what is unique about this specific problem and program. Another example might be how you name and organize your table occurrences. If you adopt a standard, then ninety percent of the time, you don't really need to think about how something should be named and placed, because you just follow your convention. Ten percent of the time, you need to do something special, so you are able to focus your concentration on designing and describing the specific aspects that make a case unique. There is a key concept to take away here: When you adopt standards, the idea is to always use them until you encounter a special case that doesn't work within the standard—deviate from the standard just long enough to accurately solve your problem. Later, when you or other developers encounter code that deviates from the standard, it will stand out, causing you to ask "why was this done differently?"

Just as a reminder, loose coupling is a huge contributing factor to maintainable code. Scripts that are loosely coupled help to reduce rigidity and fragility. Review our section on loose coupling if you haven't already read it.

Develop Coding Standards - A Style Guide for your Scripts

In a typical medium to large sized FileMaker solution, over half of the system objects will be script steps. Developing a good coding style and standards for scripting is one of the important areas for an overall maintainable system. Coding quality is judged on a different standard by every developer. Some want more comments, others want fewer comments. Some want to see 10 functions(scripts) with 20 lines each, while others might want to see 1 function with 200 lines. There are a few core areas that you should consider and then adopt your own standard. From a coding style perspective, filemakerstandards.org[11] is a great place to read up on some style standards. Not only does it give you a great place to start, but the site openly displays some of the debate involved in reaching their standard, which will help you reassess some of your own ideas. One acceptable choice would be to accept their recommendations as-is. If that doesn't work for you, then draft yourself a cheat sheet of standards you want to use.

Make a Script Template

The next step after deciding on some guidelines for coding style, is to set yourself up with a script template that gets you started with the best foot forward on every new script. Duplicate that template every time you need to start a

[11] FileMaker Coding Standards, http://filemakerstandards.org/

new script. We'll share one of our templates here, with a little commentary about each part. Your own coding values should guide the development of your own template.

In our example template, you will see that we borrowed from style guidelines that are popular with other programing languages. Specifically, we adapted a DocBlock commenting standards from Javadoc and PHPDoc. We developed a short list of tags that we expected our developers to use when marking up specific categories of comments, and those tags helped us establish a standard documentation header. A consistent header block in every script significantly reduces the comprehension time when seeing a script for the first time. Something as simple as asking a developer to summarize the purpose of their script in 3 lines or less has a profound impact on code quality. It also improves the quality of feedback you receive on a code review.

Other topics addressed in this paper are also demonstrated in the template such as nested parameters, error checking, result accumulation, and single exit.

```
##
# Short Description (3 lines max)
#
# Long Description (optional, manually wrap your lines, multiple paragraphs ok)
#
#@history
# 01/01/2014 - Joe Deveoper - Initial Version, IT Tracker #90210
#
#@assumptions
# To use this script you should first (context, layout, windows, found sets)
#
#@link https://wiki.yourcompany.dom/***
#
#@param num $customer_number (req): the number of the customer for order to be created
#@param bool $verbose (opt): when 1, provides additional user feedback, defaults to 0
#@param propertyList $transaction (req):
#     @param enum $type (req): One of "VISA" or "Cash"
#     @param num $transaction_id (cond): If type is visa, must be set
#
#@return num $error (req): non-zero indicates a problem
#@return text $message (cond): Human readable message about the general outcome of the script. Required if error.
#
#@todo Finish writing this script
#@todo Figure out a way to make a unit test for this!
#@todo Get a code review from another team member
#
#
#
# Parse out script parameters here.
Set Variable [$script_params; Value:Get ( ScriptParameter )]
Set Variable [$customer_number; Value:GetAsNumber ( #Get ( $script_params; "customer_number" ) )]
Set Variable [$verbose; Value:GetAsBoolean ( #Get ( $script_params; "verbose" ) )]
#
# Single-iteration loop
Loop
  #
  # Verify current window mode
  Exit Loop If [Get ( WindowMode )          //  Browse mode will return 0/false]
  #
  # < your code goes here >
  # Example sub-script call
  //  Perform Script [<unknown>]
  Set Variable [$script_result; Value:If (    Get ( LastError ) ;     List ( # ( "error" ; Get ( LastError ) ) ; # ( "message" ; "Perform Script error" ) ; ) ; Get ( Script
  Set Variable [$script_error; Value:GetAsNumber ( #Get ( $script_result ; "error" ) )]
  If [$script_error]
    #handle it
  End If
  #
  #
  Exit Loop If [True          //  Always exit the single-iteration control loop to prevent infinite spin! ]
End Loop
#
# Cleanup steps: close worker windows, gather script results, etc...
Set Variable [$result; Value:List (      # ( "error" ; If ( IsEmpty ( $error ) ; 0 ; $error ) ) ; # ( "message" ; $message ) ; )]
#
# Log what we were attempting to accomplish  using a standardized logger script - if appropriate
Set Variable [$log_params; Value:List       (      # ( "action" ; "Action Name" ) ; # ( "amount" ; Null ) ; # ( "backorder_qty" ; Null ) ; # ( "code" ; Null ) ; # ( "da
//  Perform Script [<unknown>]
#
# Inform the user of any errors encountered/generated. (For UI and TR scripts only, otherwise delete to avoid getting graded down.)
If [$error]
  Show Custom Dialog ["Error"; $message]
End If
#
#
# That's it - exit script!
Exit Script [Result: $result        //  We always return the result variable ]
#
```

An abridged script template from one of our front-end files. Additional templates will be available in sample solution files.

Standardize Tables, Fields, and TOs

Another area, which is probably of similar importance, is to standardize your tables, fields, and table occurrence names. For discussion on tables and fields, we would again defer to filemakerstandards.org. Again, you might consider creating a template table that has the primary key and auto-enter fields that you want to standardize for every table. As far as table graph organization, there are many schools of thought. We can't speak to all of them, but our heavy use of Anchor-Buoy seemed to have many positive attributes in our multi-power-user, multi-developer environment. There are some other good options out there, but we will use our naming standard as an example of how standardization can streamline and help clarify the graph.

We defined a three-letter acronym for every core table name. For example, Customers mapped to CUS, Invoices to INV, and Inventory to IVY. If we partitioned an entity, we'd add a standard fourth letter, such as Customers_Aux as CUSX. We made a rule for ourselves that every layout would be attached at an anchor (or its partition). The same was true for the context of any calculated field. This strategy adds a little time and size to your relationship graph in exchange for better accuracy and development efficiency when scripting and building layouts. We used a self-documenting strategy for the TO names. (A double underscore connects any acronym to its definition and helps with menu ordering.) TOs in a pop-up menu would look like this:

```
INV__Invoies (Current Table)
—
INV_ACS__AccountSummary
INV_AUT__Authorizations#Approved
INV_BCP__PrebuiltBarcodes#InvoiceNumber
INV_CUS__Customers
INV_IAD__InvoiceAddresses#BillTo
INV_IAD__InvoiceAddresses#ShipTo
INV_IAD__InvoiceAddresses#SoldTo
INV_INI__InvoiceItems
INV_INI_COI__CustomsOrderItems
INV_INI_COI_CTY__Countries
```

When pulling a field onto a layout, the TO name literally explains the relationship path and distance from the anchor. The acronyms are defined as you go. Whenever tables are joined in an unnatural way or with a filter, the hash tag explains what's different. Our large solution had multiple front-end files, business scripting files, data storage files, and a web access file. Graph nodes would be added, as-needed, to any of these files at a cost that seemed trivial in most situations. Because of the self-imposed consistency and self-documenting nature, this method was very quick to locate or determine whether the needed TO was currently available from any anchor in any file. When crafting your own graph standards, you should read some of the material that inspired us, which is listed at the bottom of this section.

Wouldn't it be great if...

...it wasn't necessary to maintain a separate legend to keep track of table acronyms?

...if TO names on the pop-up menu were organized sensibly like your pretty graph?

...you knew when you were picking the right table occurrence?

DONE! Acronyms are defined as you descend the graph.

DONE! The menu matches a depth-first alphabetical search of your graph (almost).

DONE! The pop-up menu shows how your entity is related.

Standardize Layout Object Names

One other area to consider for a naming standard is at the opposite end of the FileMaker platform with layout objects. As more of our layouts utilize tabs, triggers, and, recently, pop-overs, it becomes necessary to access many of the layout objects by name. Rather than a flood of objects that are hard to find, we adopted a rather straightforward, nested naming style that uses the element type+label, with a dot to separate each tier:

```
tab_cus_rma
tab_cus_rma.portal_issues
tab_cus_rma.portal_issues.field_id  (before putting an button action on a field)
tab_cus_rma.portal_issues.field_id#button (after putting a button action on a field)
```

Tip: FileMaker adds a second object and groups the two when you add a button action to some elements. To support the "go to object" step, improve clarity, and reduce hair loss, we recommend naming the object before and after giving it a button action.

See also Architectural principles DRY[12] and SOLID.[13]

See also Anchor Buoy and Data Structures post by Six Fried Rice[14] and the excellent 2008 paper titled "*Approaches to Graph Modeling*"[15]

Live Development

The future is here today. Many development platforms would love to be able to support live development. We get a chance to use it today. And while it isn't a perfected science in FileMaker, it is certainly viable. There are certainly

---

[12] Don't Repeat Yourself, Hunt, Andrews & Thomas, David, *The Pragmatic Programmer*

[13] SOLID (Single responsibility, Open-closed, Liskov substitution, Interface segregation and Dependency inversion), Martin, Robert, *Principles of OOD*

[14] http://sixfriedrice.com/wp/six-fried-rice-methodology-part-2-anchor-buoy-and-data-structures/

[15] Cologon, Ray, "Approaches to Graph Modeling" http://www.nightwingenterprises.com/Resources/approaches_to_graph_modeling_en.pdf

many risks involved with live development, but understanding some of the benefits and limitations can help you to decide if this is something you want to do.

A standard disclaimer goes here: Yes, some of these recommendations, along with just about everything else, put you at risk for losing data. However, if done correctly, development on a live system can provide many merits and maybe even a competitive advantage over other system platforms. Live development also means that you can do A/B testing against a live platform and data. It also allows users or automated systems to detect and report issues with new code earlier.

Schema Interactions

One of the most important issues to understand with live development is how interactions with system schema, especially while using full access, can tie up different areas of the solution. It is helpful to point out that there is a bit of gray area with the definition of what "schema" means in a FileMaker solution. At one end of the spectrum, you might describe your tables and relationship graph as the schema. At the other end of the spectrum, might be everything but your data (what you get when you save a clone). In the middle are things like privilege sets, custom functions, scripts, and layouts.

Interactions with table definitions, something which is very low level, can trigger long lasting locks that may prevent users from creating or editing records for long periods of time. Interactions with things like table privilege sets can seem to create a checkpoint that stalls user calls until all previous calls have finished executing. For example, a user that is performing a long running find, keeps the security changes from being saved and, likewise, any new find requests from other users can't start until the security changes are saved.

In our experience, almost every potential problem can be detected and safely handled by well-written scripts that check for errors when trying to create or edit records. At the same time, if some of your scripts do not have any error checking, then you are likely to get unpredictable behavior when they try to create or edit records.

Auto-Enter Serial Numbers

A lot of fear surrounds editing field definitions on a table in a live system, and for good reason. The largest source of schema-locking issues are fields that use the Auto-Enter Serial number. It makes sense that this is the case. When you have retrieved the “next value” for one of these fields, it isn't safe for anyone else to create a new record because when you save your database changes, the serial number could be stale, causing duplicate values or primary keys. This problem is made larger because retrieving information for any "Options" window of any field in that table, seems to acquire this lock. In our experience, this type of lock has probably been the greatest cause of grief, because of how long it may be in place. To be clear, here are the specific situations to avoid:

Conditions:

- A table includes at least one field with the Auto-Enter serial number on creation.
- You bring up the "Options" window for ANY (existing or new) field in that table.

Result:

- A schema lock is acquired against that table.
- Any attempt to create a new record on that table will cause Error 303 "Database schema is in use by another user"

Release:

- The lock remains until you dismiss the previous Manage Database window.

Tip: If you just need to read a formula of a calculation but can't safely click the options button, print that field instead. Printing table definition information doesn't lock anything. This is also useful if you need to see inside script steps when another user is editing that script.

Table Re-org

A less serious but still significant low-level schema lock can occur when modifying the database in other ways. This is best described as any activity that causes the storage of table records to be reorganized. This could include actions such as manually adding an index to a field, creating or updating a stored calculation, or maybe even adding or removing a simple field. During the period when this reorganizing takes place, other users cannot modify any record in that table.

Conditions:

- Someone has manually initiated changes to a table definition by clicking OK on the Manage Database window.

Result:

- A schema lock is acquired against that table.
- Any attempt to lock a record (such as a set field) will cause Error 302 "Table is in use by another user"

Release:

- The lock remains while the table is being reorganized, which could range from a imperceptible fraction of a second to many minutes, depending on the complexity of the change and the number of records in the table.

Tip: If an index is automatically added to a field, it does not seem to trigger this lock error, but adding an index manually through field options does.

Best Practices

Let's run down a few of the less obvious best practice recommendations if/when you plan to develop on a live system.

Tables

Avoid using auto-enter serial numbers. Instead, when you need a primary key, consider using a random UUID, which doesn't trigger any sort of schema locking. If you do need a human readable document number, consider pre-allocating numbers or developing your own sequence generator that minimizes the impacts that can be caused by auto-enter serial numbers. If revamping an old table, don't stop after the primary key, as other fields utilizing this feature can also trigger table locks.

Scripts

Robust error handling in your scripts is especially important. Focus your initial efforts on any script step that could open (lock for editing) or create a record, including steps that automatically create a record via a relationship.

Adopt standard development conventions and procedures. Including a change history in the comment header of your scripts can be extremely helpful, especially in live environments where bugs need to be located and fixed quickly. If possible, ask a peer to review your script changes any time you are making significant enhancements or altering some critical component. Other team members are likely to notice edge cases or design flaws that you just can't see as the initial designer of a solution. Consider using a separate ticket/bug database, integrating those ticket numbers right into the change history.

Editing scripts that are being actively executed by other users is pretty risky and might even expose you to flaws or bugs in the FileMaker platform. This is particularly a concern for any long running scripts. A better route is to fork the script, adding a suffix to the script name like V2 or Dev. Make a minor change to the calling script(s) that routes execution to the new version for a small set of developers and testers. Finally, deploy the new script by updating references to the older script to point at the new one. (Doing this on a large system really necessitates using a DDR analyzer like InspectorPro or BaseElements.)

Relationship Graph

Be aware that schema updates seem to be propagated to other users in a best-effort fashion. Adding new TOs to your graph is one of the relatively safe operations, but client computers on rare occasion seem to lag behind. If you need to be absolutely sure that all users have received some critical schema update, then you may need them disconnect and reconnect to the affected files.

Use a "Maintenance Mode" strategy

Consider regulating critical portions of your system, so that edits can only be performed by a small number of scripts. For example, define a business policy that requires all edits in inventory to go through one of three well-defined scripts. Not only does that provide a great way to audit and verify the accuracy of the system, but it also means that you can suspend operations in a specific area to allow for maintenance. Users could receive a message about inventory operations being unavailable for the next 30 minutes. While inconvenient, this is far less impactful then taking down an entire system. You could even take this further with a system that is zoned off with set of maintenance flags for each section.

```
Exit Script [ "error=-999¶message="Inventory system under maintenance until 7/19/2014 10PM PST. Call help desk with questions at x5555." ]
```

Be Aware of Referenced Objects

DDR analyzers like InspectorPro or BaseElements can be extremely helpful when determining which other objects call the object you are working on. When cleaning out old objects, those tools can alert you to references that you might not be aware of. But, this doesn't necessarily cover everything. Using functions like Evaluate(), ExecuteSQL(), or steps like Go To Layout By Name[] can create systems with uncounted references. You should use one of the well-documented techniques that help to intentionally trigger a reference count in these situations. Also be aware, if you have any external dependancies such as PHP, CWP, or ODBC. In those situations you need to create manual documentation of items that are externally referenced. If you are using PHP or CWP, consider setting up an extra file just for these types of access. Requiring all external connections to pass through a dedicated file means that you will have TOs, layouts, and scripts that give you clues as to the items that could be referenced externally.

Additional Files Might Be Good

If your solution is large, there may be advantages to breaking things up into more files. This could include multiple front-end files or multiple table storage files, grouped on topic. See the earlier topic in this paper for recommended scenarios. Some types of schema locks (security changes) and backups (database pause) affect one file at a time. Disruptions and even disaster recovery might be contained to a single smaller file.