EasyPDF™

The Acrobat ENCYCLOPEDIA

THREE DECADES OF PRACTICAL EXPERIENCE BUILDING INTERACTIVE PDF FORMS WITH ACROBAT JAVASCRIPT

Techniques • Real-World Solutions • Best Practices

Form Calculations
Powerful field calculations and validation

Field Validation
Ensure accurate data entry with custom validation

User Interactions
Create dynamic forms that respond to user actions

Data Processing
Manipulate, format, and process data efficiently

Advanced Topics
Explore advanced techniques and best practices

The Definitive Resource for Creating Professional, JavaScript-Enhanced Interactive PDF Forms in Adobe Acrobat

TRUSTED • RELIABLE • PROVEN SINCE 2003

EASYPDF™ FORMS

DOCUMENT AUTOMATION MADE EASY

by Marty Karr

© Marty Karr

Preface

Compiled and maintained by
Marty Karr
with the assistance of ChatGPT (OpenAI)

Preview Edition

Version 0.3

This publication is released as a **Preview Edition** and documents Acrobat JavaScript design techniques developed through more than three decades of real-world PDF application development.

Rather than serving as a traditional programming reference, The Acrobat Encyclopedia presents engineering knowledge accumulated while designing, testing, refining, and maintaining the EasyPDF™ family of interactive PDF forms.

The final edition will include a fully interactive Table of Contents and additional navigation features.

Because this Preview Edition will be periodically updated as new chapters are completed, readers are encouraged to revisit the EasyPDF™ Forms website from time to time to download the latest available edition.

Copyright

Copyright © 2026 Marty Karr. All rights reserved.

This Preview Edition may be freely distributed in its original, unmodified PDF format. No portion of this publication may be reproduced, modified, translated, incorporated into another publication, or sold without the prior written permission of the copyright holder.

About This Encyclopedia

Unlike traditional programming references that primarily document language syntax and software features, **The Acrobat Encyclopedia** is built upon engineering experience accumulated while designing and maintaining the EasyPDF™ family of interactive PDF forms.

Rather than attempting to document every Acrobat JavaScript object, property, method, and event, this encyclopedia concentrates on the techniques, design patterns, architectural decisions, performance considerations, Acrobat quirks, and lessons learned that consistently produce reliable, maintainable, and professional interactive PDF applications.

Its purpose is not merely to explain **what** Acrobat JavaScript can do, but to demonstrate **how**, **when**, and **why** proven development techniques can be applied to build dependable PDF solutions that perform effectively in both Adobe Acrobat and the free Adobe Acrobat Reader.

EasyPDF™
Acrobat Encyclopedia
Table of Contents

© Marty Karr

© Marty Karr

© Marty Karr

2.2 updateNames() Design Pattern

Problem

Nearly every EasyPDF™ form maintains a drop-down list containing user-defined records such as customers, contacts, patients, medications, websites, invoices, or passwords. Whenever a record is added, deleted, restored, or renamed, the associated drop-down list must immediately reflect the current contents of the underlying JSON data store.

Early implementations rebuilt each drop-down list using application-specific code. Although functional, this resulted in duplicated programming logic and increased maintenance whenever improvements or corrections were required.

Investigation

As additional EasyPDF™ forms were developed, it became apparent that every application performed essentially the same task:

- Read the JSON data store.
- Build an array of valid record names.
- Exclude internal application properties.
- Sort the names when appropriate.
- Repopulate the drop-down list.
- Restore the current selection whenever possible.
- Update the navigation buttons.

The only differences between applications were the names of the fields and the specific JSON object being processed.

This observation suggested that rebuilding a drop-down list should follow a standardized design pattern regardless of the application.

Solution

Each EasyPDF™ form implements a dedicated updateNames() routine responsible for rebuilding its associated drop-down list from the current JSON data.

Rather than allowing individual application routines to manipulate the drop-down directly, every operation that changes the underlying data simply calls updateNames().

The updateNames() routine becomes the single authority responsible for rebuilding the list, restoring the current selection when appropriate, excluding internal application properties, and updating the associated navigation buttons.

© Marty Karr

Why It Works

By assigning all drop-down maintenance responsibilities to a single function, every record-management operation behaves consistently.

The application no longer needs to remember how to update the user interface after each individual operation. Instead, add(), delete(), restore(), rename(), and similar routines simply update the data and then call updateNames().

This separation of responsibilities reduces duplicated code while making future enhancements considerably easier to implement.

Design Principle

Whenever multiple operations affect the same user interface element, assign responsibility for maintaining that element to a single dedicated routine.

The application should modify its data first and then allow the specialized update routine to refresh the user interface.

Lessons Learned

- A drop-down list should have one authoritative update routine.
- User interface rebuilding should never be duplicated throughout the application.
- Separating data manipulation from user interface updates simplifies maintenance.
- Improvements made to updateNames() automatically benefit every routine that calls it.

Historical Note

The updateNames() design pattern gradually evolved while developing multiple EasyPDF™ forms including the Password Manager, CRM System, Digital Rolodex, Medication Manager, and Smart Invoice. As each application introduced new features, the advantages of centralizing drop-down maintenance became increasingly apparent. The pattern ultimately became one of the fundamental architectural standards adopted throughout the EasyPDF™ family of forms.

EasyPDF™ Engineering Principle

Whenever possible, create one authoritative function responsible for maintaining each user interface component. All application routines should rely on that function rather than attempting to duplicate its responsibilities.

© Marty Karr

3.3 Nested JSON Structures

Problem

Early EasyPDF™ forms stored records using relatively simple JSON structures. While this approach worked well for standalone applications, it became increasingly restrictive as more sophisticated relationships between records were introduced.

For example, the Smart Invoice application required each customer to own multiple invoices while preserving customer-specific information independently from invoice-specific information. Similar requirements later appeared in other EasyPDF™ forms as the applications continued to evolve.

Attempting to maintain these relationships using separate JSON objects or unrelated hidden fields quickly became difficult to manage and increased the likelihood of data inconsistencies.

Investigation

Initially, independent JSON objects appeared attractive because they were easy to understand and straightforward to implement.

However, as additional features were added, every relationship between records required additional programming to keep multiple data stores synchronized.

Simple operations such as deleting a customer, restoring an invoice, assigning unique record identifiers, or updating customer information required multiple objects to remain perfectly synchronized. As application complexity increased, maintaining these relationships became unnecessarily complicated.

It became apparent that the data itself should define the relationships rather than relying upon program logic to reconstruct them.

Solution

EasyPDF™ applications gradually adopted nested JSON structures in which related records became part of the parent object.

For example, each customer object maintains its own collection of invoices. Rather than storing invoices independently and attempting to associate them later, every invoice naturally belongs to its customer within the same JSON hierarchy.

This approach places related information together, allowing the application's structure to mirror the logical relationships between the records it manages.

Why It Works

© Marty Karr

Nested JSON structures significantly reduce application complexity because relationships are preserved automatically by the storage model itself.

Retrieving all invoices for a customer no longer requires searching multiple independent objects.

Deleting or restoring records becomes more reliable because related information remains grouped together.

The resulting code is easier to understand, easier to maintain, and considerably less prone to synchronization errors.

Design Principle

Whenever records possess a natural parent-child relationship, the storage structure should reflect that relationship.

The organization of the data should simplify the application rather than forcing the application to compensate for an unnecessarily complicated storage model.

Lessons Learned

- Store related information together whenever practical.
- Allow the JSON structure to represent real-world relationships.
- Eliminate unnecessary synchronization between multiple data stores.
- Well-designed data structures simplify programming far more than additional application logic.

Historical Note

The transition to nested JSON structures occurred during development of EasyPDF™ Smart Invoice and quickly influenced the design of several additional EasyPDF™ forms. Once the advantages became apparent, nested storage replaced several earlier approaches that relied upon maintaining separate collections of related information. The result was a more maintainable architecture capable of supporting future expansion with considerably less programming effort.

EasyPDF™ Engineering Principle

Design the data structure first.

When the underlying data accurately represents the relationships being modeled, much of the application's complexity disappears naturally.

© Marty Karr

6.5.1 – Why Combo Boxes Became My First Choice

Problem

When I first began developing interactive PDF forms, one problem quickly became apparent. Regardless of the type of form being created, users were repeatedly entering the same information over and over again. Customer names, company names, addresses, cities, physicians, medications, invoice descriptions, and countless other values were typed repeatedly, often several times within the same document or across multiple documents.

Besides being tedious, repetitive typing increased the likelihood of typographical errors, inconsistent spellings, and wasted time correcting mistakes. If a customer name was entered differently from one form to the next, searching and organizing records became increasingly difficult. I wanted a method that would reduce typing while allowing previously entered values to be reused whenever needed.

At the time, Acrobat's editable combo box appeared to be the perfect solution.

Investigation

Unlike a standard text field, an editable combo box offered two important advantages.

First, it allowed users to select previously entered values from a drop-down list instead of retyping them.

Second, because the field remained editable, users could still enter entirely new values whenever necessary.

This combination of flexibility and convenience immediately appealed to me. Each time a new customer, company, physician, product, or other frequently used value was entered, I could simply add it to the combo box so it would be available for future selection.

For many years this approach worked remarkably well.

Users typed less.

Typographical errors were reduced.

Frequently used information gradually accumulated into personalized drop-down lists that became increasingly valuable over time.

At first glance, the problem appeared to be solved.

Solution

To make this work, I developed several document-level JavaScript functions responsible for maintaining each combo box.

Whenever a user entered a value that did not already exist within the list, an On Blur event called the necessary document-level scripts to:

© Marty Karr

- Determine whether the value already existed.
- Prevent duplicate entries.
- Add the new item to the list.
- Optionally sort the list alphabetically.
- Preserve the updated list for future use.

Over time, additional management routines were added to remove obsolete entries, rebuild lists, and perform other maintenance tasks required to keep the combo boxes organized.

For many years, this architecture successfully accomplished exactly what I intended.

Why It Worked

The design addressed several important usability problems simultaneously.

Users spent less time typing.

Frequently used information became available with a single mouse click.

Consistency improved because previously entered values were reused rather than retyped.

Most importantly, the editable combo box allowed new information to be entered whenever necessary without requiring users to leave the field or invoke a separate dialog box.

Considering the Acrobat JavaScript features available at the time, the editable combo box represented a practical and effective solution.

Design Principle

A good design should solve today's problem without unnecessarily limiting tomorrow's solution.

When I first adopted editable combo boxes, they accomplished exactly what I needed. The architecture was appropriate for the tools and knowledge available at the time.

Good engineering decisions should always be evaluated within the context in which they were originally made rather than judged solely by modern standards.

Lessons Learned

One lesson I have learned repeatedly throughout software development is that today's best solution rarely remains the best solution forever.

As applications become more sophisticated, requirements change. What once appeared to be an elegant architecture may gradually reveal limitations that were never apparent during the original design.

Recognizing when an architecture has reached that point is often more valuable than the architecture itself.

© Marty Karr

Historical Note

Looking back nearly twenty years later, I still believe choosing editable combo boxes was the correct decision at the time. They dramatically improved data entry, reduced typographical errors, and provided a level of convenience that standard text fields simply could not offer.

Ironically, the success of the approach eventually exposed its greatest weakness. As the applications continued to evolve, maintaining the combo boxes required an ever-growing collection of management scripts that gradually became more complicated than the original problem they were intended to solve.

That realization marked the beginning of the next stage in the evolution of my Acrobat forms—one that would eventually lead beyond editable combo boxes entirely.

© Marty Karr

6.5.2 – Growing Pains

Problem

For many years, editable combo boxes performed exactly as intended. They reduced typing, minimized typographical errors, and gradually accumulated frequently used values that significantly improved productivity.

As my PDF applications became increasingly sophisticated, however, so did the responsibilities assigned to each combo box.

What had once been little more than a convenient drop-down list gradually evolved into a miniature database requiring constant maintenance. Every new feature introduced additional logic, and every enhancement required another document-level script or modification to an existing one.

At first these changes appeared insignificant. Collectively, however, they transformed a simple solution into an increasingly complicated architecture.

Investigation

Originally, a combo box simply needed to remember previously entered values.

Soon additional requirements appeared.

Duplicate entries had to be prevented.

New entries needed to be inserted into the correct location.

Lists required alphabetical sorting.

Obsolete items occasionally needed to be removed.

Some forms required rebuilding the entire list after modifications.

Others required preserving the user's current selection while simultaneously updating the list.

What had begun as a handful of helper functions gradually expanded into a growing collection of document-level scripts, each responsible for managing one small aspect of the combo box.

Individually, none of these scripts appeared particularly complicated.

Collectively, they became increasingly difficult to understand, maintain, and modify.

Solution

Whenever new functionality was required, I simply added another routine.

Additional helper functions were written to:

- **Detect duplicate entries.**
- **Insert new values.**
- **Sort items alphabetically.**

© Marty Karr

- Delete obsolete values.
- Rebuild lists after modifications.
- Preserve selected values whenever possible.

Each individual enhancement successfully addressed the immediate problem.

Unfortunately, every improvement also increased the overall complexity of the architecture.

The solution continued to work, but it was gradually becoming more difficult to maintain.

Why It Worked

From the user's perspective, nothing appeared wrong.

The combo boxes continued functioning exactly as expected.

Drop-down lists grew over time.

Previously entered values remained available.

Data entry continued to improve.

The growing complexity remained hidden entirely within the supporting JavaScript code.

This is often one of the most deceptive stages in software development.

Applications continue working normally while the underlying architecture quietly becomes more fragile.

Because users experience no immediate problems, developers may fail to recognize the long-term maintenance burden gradually accumulating beneath the surface.

Design Principle

Every helper function introduced into a project should reduce complexity rather than merely relocate it.

One of the easiest traps for developers is solving individual problems independently without periodically reevaluating the overall architecture.

A growing collection of helper functions often indicates that the original design is beginning to reach its practical limits.

When maintaining the supporting code becomes more difficult than understanding the original problem, it is usually time to reconsider the architecture itself rather than continue adding additional fixes.

Lessons Learned

One lesson experience has repeatedly taught me is that software rarely becomes difficult to maintain overnight.

Instead, complexity accumulates gradually.

© Marty Karr

Each individual enhancement appears reasonable.

Each additional helper function solves a legitimate problem.

Each modification seems relatively minor.

Years later, however, developers often discover they are maintaining an elaborate framework created to support what originally began as a very simple idea.

Recognizing that gradual accumulation of complexity is one of the most valuable skills software developers can develop.

Historical Note

Looking back today, I can clearly see what I failed to recognize at the time.

The problem was never the combo boxes themselves.

Nor was it any individual document-level script.

The real issue was that I had become so accustomed to extending the original architecture that I never stopped to ask a much simpler question:

"Is there now a better way to accomplish the same objective?"

Ironically, the answer had been quietly waiting inside Acrobat all along.

I simply had not yet discovered it.

That discovery would eventually arrive in the form of popup menus, fundamentally changing how I approached reusable data entry within my PDF applications.

© Marty Karr

6.5.3 – The Straw That Broke the Camel's Back

Problem

By this point, my combo box architecture had matured through years of incremental improvements. New entries could be added automatically, duplicate values were prevented, lists could be sorted alphabetically, obsolete items removed, and previously entered information quickly recalled.

From the user's perspective, everything continued to function normally.

Behind the scenes, however, maintaining those capabilities had become increasingly complicated. Updating a combo box was no longer a simple matter of adding or removing an item. Every modification required careful coordination between the field's current value, its list contents, supporting document-level scripts, and Acrobat's event model.

The architecture still worked.

It simply no longer felt dependable.

Investigation

The turning point occurred while rebuilding combo box lists dynamically.

What initially appeared to be a straightforward operation—

- clearing the existing items,
- rebuilding the list,
- restoring the current selection—

proved to be anything but straightforward.

Repeatedly, I encountered situations where calling `clearItems()` followed by `setItems()` caused the combo box to lose, change, or fail to restore its current value as expected.

Preserving the user's selection suddenly required additional logic.

Execution order became increasingly important.

Assigning the field value too early produced one result.

Assigning it too late produced another.

Sometimes the displayed value and the underlying list no longer agreed.

What should have been a routine maintenance operation had become surprisingly fragile.

The problem was no longer adding data.

The problem was safely rebuilding the control itself.

Solution

Initially, I approached the issue the same way I had solved countless others.

© Marty Karr

I experimented.

I adjusted the execution order.

I temporarily stored the current value.

I restored the selection after rebuilding the list.

Additional helper routines were written.

More conditional logic was introduced.

Eventually, I succeeded in making the process work reliably.

Ironically, solving the immediate problem exposed a much larger one.

I realized I had written an increasing amount of JavaScript—not to improve my applications—but simply to compensate for the behavior of a single Acrobat control.

At that moment, I stopped trying to improve the combo box.

Instead, I began looking for a better architecture.

Why It Worked

Ironically, the solution wasn't another helper function.

It was changing my perspective.

Until then, I had assumed the combo box itself was an essential part of the design.

Once I questioned that assumption, entirely new possibilities became visible.

The real requirement had never been:

"I need a combo box."

The real requirement had always been:

"I need an efficient way for users to select previously entered information while still allowing new information to be added."

Those are very different problems.

Once I separated the user requirement from the Acrobat control implementing it, better alternatives naturally began to emerge.

Design Principle

One of the most valuable lessons I've learned is this:

Never allow a software control to dictate your application's architecture.

Controls are implementation details.

User requirements are design goals.

Whenever increasing amounts of code are devoted to managing the behavior of a single control rather than solving the user's actual problem, it's often a sign that the architecture deserves to be

© Marty Karr

reconsidered.

Sometimes the best solution isn't another workaround.

Sometimes it's replacing the underlying approach entirely.

Lessons Learned

For years, I believed the growing complexity surrounding my combo boxes was simply the unavoidable cost of building sophisticated PDF applications.

I eventually realized the opposite was true.

The complexity wasn't inherent to the application.

It was largely self-inflicted by continuing to build upon an architecture that had already reached its practical limits.

That realization changed the way I approached software development.

Rather than continually asking,

"How can I improve this design?"

I began asking,

"Should I still be using this design at all?"

That single question has influenced countless design decisions ever since.

Historical Note

Looking back today, I no longer remember the exact debugging session that finally convinced me to abandon further investment in combo boxes.

What I do remember is the feeling.
After spending yet another evening wrestling with `clearItems()`, `setItems()`, execution order, value restoration, and supporting helper routines, I finally reached a conclusion that was both obvious and liberating.

The combo boxes were no longer the solution.

They had quietly become part of the problem.

That realization marked the end of one chapter in my Acrobat development experience and the beginning of another.

The search for a simpler, more maintainable alternative had officially begun.

The next stop on that journey would be Acrobat's popup menus.

© Marty Karr

6.5.4 – Discovering Popup Menus

Problem

After years of refining my combo box architecture, I finally reached a point where adding yet another helper function no longer seemed like progress. Every improvement addressed a specific issue, yet the overall design continued growing more complicated.

I had become increasingly convinced that I was solving the wrong problem.

Instead of finding better ways to manage combo boxes, I began asking a much simpler question:

Was there another Acrobat feature capable of accomplishing the same objective more effectively?

That question ultimately led me in an entirely different direction.

Investigation

While exploring Acrobat's JavaScript object model, I encountered a document-level method that had received little attention in the examples and reference material I had previously studied:

```
app.popUpMenuEx()
```

At first glance, it appeared to be little more than a utility for displaying popup menus.

The more I experimented with it, however, the more possibilities began to emerge.

Unlike combo boxes, popup menus were created dynamically.

The menu itself did not need to exist permanently within the document.

Instead, it could be constructed at runtime using JavaScript, displayed when needed, and discarded immediately after the user made a selection.

That single characteristic fundamentally changed the way I began thinking about reusable data entry.

Solution

Rather than asking Acrobat to maintain an editable list inside a form field, I began generating popup menus directly from my own data structures.

Instead of depending upon the behavior of a combo box, I could now:

- Build menu items dynamically.
- Display only the choices relevant to the current task.
- Capture the user's selection.
- Assign the selected value to any field I desired.
- Eliminate much of the maintenance previously required by editable combo boxes.

The popup menu became nothing more than a temporary user interface.

© Marty Karr

The real data remained under my complete control.

For the first time, the user interface and the stored data no longer depended upon one another.

Why It Worked

Popup menus shifted responsibility away from Acrobat's form controls and back into my own application logic.

Rather than allowing the combo box to manage both presentation and storage, each responsibility became independent.

The popup menu handled user interaction.

My JavaScript managed the data.

Because the menu was generated only when needed, I was no longer constrained by many of the limitations associated with maintaining editable combo boxes.

The architecture immediately became more flexible.

Design Principle

One of the most powerful techniques in software design is separating the user interface from the underlying data.

When those responsibilities become independent, each can evolve without forcing unnecessary changes upon the other.

Popup menus taught me that a control displayed to the user does not necessarily have to be responsible for storing the application's data.

That realization would eventually influence nearly every EasyPDF™ form I developed thereafter.

Lessons Learned

Discovering popup menus taught me an important lesson about software development.

Sometimes the solution to a difficult problem isn't another workaround.

Sometimes the solution is discovering a feature you previously overlooked.

Had I continued concentrating exclusively on improving combo boxes, I might never have recognized that Acrobat already provided another mechanism capable of accomplishing much of what I needed.

Exploring unfamiliar portions of an API often produces far greater rewards than endlessly refining familiar ones.

© Marty Karr

Historical Note

Historical Note (Revised)

Although popup menus represented a significant improvement over editable combo boxes, they were not the final destination.

As my applications continued evolving, I eventually discovered that constructing popup menu arrays directly within JavaScript also became increasingly cumbersome as the amount of stored information continued growing.

It would be unfair, however, to discuss popup menus without acknowledging the individual who first introduced me to their potential.

During the early 2000s, when online Acrobat JavaScript resources were still relatively scarce, Thom Parker published an outstanding article explaining the `app.popUpMenuEx()` method through practical, easy-to-understand examples. His clear writing transformed what initially appeared to be an obscure Acrobat feature into a remarkably useful development tool. That article opened my eyes to possibilities I had never previously considered and ultimately influenced the direction of many of my future PDF applications.

For his many years of sharing his knowledge through the Adobe Acrobat community, I extend my sincere appreciation and thanks.

Although popup menus represented an important architectural improvement, they were not the final destination. They changed the way I thought about application design by separating the user interface from the underlying data.

That change in philosophy would eventually lead to what I consider the most significant evolution of all—the use of hidden list boxes as the permanent data source for dynamically generated popup menus.

© Marty Karr

6.5.5 – Hidden List Boxes Change Everything

Problem

Although popup menus represented a significant improvement over editable combo boxes, they did not completely eliminate the underlying maintenance problem.

The popup menus themselves still required data.

Initially, that data was constructed directly within JavaScript using arrays and supporting code. While this approach was certainly more flexible than maintaining editable combo boxes, I eventually encountered a familiar problem.

As the applications continued growing, the JavaScript responsible for constructing and maintaining those arrays also continued growing.

The architecture had improved.

The maintenance burden had merely shifted to another location.

I wanted something simpler.

I wanted a permanent data source that Acrobat itself could manage while allowing my JavaScript to focus solely on application logic.

Investigation

The breakthrough occurred almost by accident.

Rather than storing menu items inside JavaScript arrays or relying upon editable combo boxes, I began experimenting with an Acrobat form control that most developers rarely consider for anything beyond displaying a list of selectable items:

The humble list box.

Instead of presenting it to the user, I simply hid it from view.

That single decision completely changed the architecture.

The hidden list box became nothing more than a permanent repository for reusable values.

Whenever a popup menu was needed, the JavaScript simply read every item contained within the hidden list box, dynamically constructed the popup menu, displayed it to the user, and assigned the selected value to the appropriate text field.

The list box was never intended to be part of the user interface.

It had quietly become part of the application's internal data model.

Solution

Replacing JavaScript arrays with hidden list boxes simplified the entire design.

Each hidden list box became responsible for storing one logical collection of reusable information.

© Marty Karr

Examples included:

- **Customer names**
- **Company names**
- **Professions**
- **Job titles**
- **Business categories**
- **Contractor names**
- **Medications**
- **Status values**
- **Any other frequently reused information**

Whenever a new value was entered, the hidden list box was updated.

Whenever a popup menu was displayed, it simply read its contents directly from the list box.

Whenever an obsolete value needed to be removed, the list box was modified—not the popup menu.

The popup menu no longer maintained data.

It merely displayed whatever data already existed.

That distinction proved remarkably important.

Why It Worked

For the first time, every major responsibility became clearly separated.

The hidden list box stored the data.

The popup menu displayed the available choices.

The visible text field displayed the user's selection.

Each component performed exactly one responsibility.

Because each responsibility was independent, modifying one no longer required rewriting the others.

Adding or deleting values affected only the hidden list box.

The popup menu automatically reflected the updated information because it generated its contents directly from that single source.

There were no duplicate arrays to synchronize.

There were no combo box lists to rebuild.

There was only one authoritative source of information.

© Marty Karr

The architecture had finally become both simpler and more reliable.

Design Principle

One of the most important principles I've learned throughout software development is this:

Every piece of information should have one—and only one—authoritative source.

Whenever the same data exists in multiple locations, those copies eventually become inconsistent.

By allowing the hidden list box to become the single source of truth, every other part of the application simply consumed that information without attempting to maintain its own duplicate copy.

That single design decision dramatically reduced complexity while improving reliability.

Lessons Learned

Hidden list boxes taught me that sometimes the most valuable software components are the ones users never see.

Developers often concentrate on improving visible controls while overlooking the possibility that invisible infrastructure can simplify the entire application.

The hidden list box was never intended to impress users.

Its value came from making the underlying architecture cleaner, easier to maintain, and far more reusable.

Looking back, I no longer think of hidden list boxes as Acrobat form fields.

I think of them as lightweight data repositories that happen to reside inside a PDF document.

That simple shift in perspective changed the way I designed virtually every EasyPDF™ application that followed.

Historical Note

Ironically, this architectural improvement did not originate from a desire to invent something new.

It originated from frustration.

What began as repeated attempts to simplify combo boxes eventually led to popup menus.

Popup menus then revealed another opportunity to simplify the architecture even further.

During one of my discussions with ChatGPT in 2026, we explored replacing JavaScript-managed menu arrays with hidden list boxes serving as permanent data sources for dynamically generated popup menus. At first, the idea seemed almost too simple.

The more I experimented with it, however, the more obvious its advantages became.

The architecture immediately became easier to understand.

The JavaScript became shorter and more reusable.

© Marty Karr

Maintenance became significantly simpler.

Most importantly, the architecture finally reflected the way I wanted my applications to operate.

Looking back today, I consider the introduction of hidden list boxes to be one of the most significant design improvements ever incorporated into my EasyPDF™ forms. It wasn't simply another Acrobat programming technique.

It represented a fundamental change in the way I separated data storage, user interaction, and application logic.

© Marty Karr

6.5.6 Lessons Learned

Problem

Looking back over nearly three decades of Acrobat development, it would be easy to conclude that the story of this chapter is about replacing one user interface control with another. In reality, that was never the lesson.

Editable combo boxes, popup menus, and hidden list boxes were simply milestones along a much larger journey. Each represented the best solution I knew at the time. None of them were failures. Rather, each eventually revealed opportunities for improvement as my understanding of Acrobat JavaScript continued to mature.

The real challenge was never choosing the "perfect" control. It was learning how to recognize when an architecture had reached the limits of its usefulness and having the willingness to redesign it rather than continue adding complexity to preserve an aging solution.

Investigation

One of the most valuable lessons I learned throughout this evolution is that software architecture should always serve the application—not the other way around.

When I first adopted editable combo boxes, they solved a genuine problem by reducing repetitive typing and minimizing data entry errors. As my forms became more sophisticated, however, I found myself writing increasing amounts of JavaScript whose sole purpose was to compensate for the limitations of the control itself. Helper functions multiplied, rebuilding lists became routine, preserving selections required additional code, and maintaining synchronization between stored data and the user interface gradually became more difficult than the original problem I had set out to solve.

Discovering popup menus simplified much of that complexity by separating the user interface from permanent storage. Later, replacing JavaScript arrays with hidden list boxes simplified the architecture even further by giving each logical data set a single authoritative source. The more responsibility I removed from the user interface, the simpler and more maintainable the application became.

Solution

The final architecture evolved into a simple but highly effective pattern:

- Hidden list boxes permanently store and organize application data.
- Popup menus present that data to the user only when needed.
- Visible text fields display the selected value and remain responsible solely for user interaction.

Each component performs one well-defined responsibility.

The user interface no longer manages data.

© Marty Karr

The data no longer depends upon the user interface.

The application simply coordinates the relationship between the two.

This separation of responsibilities has proven to be one of the most important architectural improvements I've adopted across my EasyPDF™ forms.

Why It Works

Good software architecture reduces complexity rather than hiding it.

By separating storage, presentation, and interaction into independent responsibilities, individual components become easier to understand, easier to maintain, and easier to expand. New features can often be added without redesigning the underlying application because each component has a clearly defined purpose.

More importantly, future improvements become evolutionary rather than disruptive. Individual pieces of the architecture can continue to evolve without forcing the entire application to be rewritten.

Design Principle

Choose an architecture that can grow with your understanding—not merely one that solves today's problem.

Every programmer eventually discovers techniques that seem ideal at the time. Experience teaches that the best designs are rarely those requiring the least code initially, but those that continue to remain understandable, adaptable, and maintainable years later.

Good architecture should make future improvements easier rather than more difficult.

Lessons Learned

Several principles emerged repeatedly throughout this journey:

- **The simplest solution today may not remain the simplest solution tomorrow.**
- **Never allow a user interface control to dictate the architecture of your application.**
- **Separate data storage from user interaction whenever practical.**
- **Give every component a single, well-defined responsibility.**
- **Eliminate duplicate sources of information whenever possible.**
- **Redesign when necessary rather than continually extending an architecture that has outlived its usefulness.**
- **Experience is gained not by avoiding mistakes, but by recognizing when a better solution has emerged.**

Most importantly, I learned that successful software development is an ongoing process of

© Marty Karr

refinement. Every generation of a design builds upon the lessons learned from the one before it.

Historical Note

When I first began writing Acrobat JavaScript, I could never have imagined that a simple editable combo box would eventually lead to a complete rethinking of how I designed interactive PDF applications.

Each stage of that journey—from combo boxes, to popup menus, to hidden list boxes—represented a meaningful step forward because each solved a problem the previous architecture could no longer solve as effectively.

Looking back, I have no regrets about the earlier designs. They were the right solutions given my knowledge and experience at the time. Without them, I would never have discovered the improvements that followed.

Perhaps the most rewarding realization is that the journey itself never truly ends. Every solution teaches something new, every design reveals new possibilities, and every challenge becomes an opportunity to build something better.

That philosophy has shaped not only the architecture of my EasyPDF™ forms, but also the pages of this reference.

EasyPDF™ Engineering Principle

Great software architecture is not measured by how long the original design survives, but by how gracefully it evolves as knowledge and experience increase.

© Marty Karr

6.6 Acrobat Reader vs. Acrobat Pro

Problem

Many developers mistakenly assume that because a PDF was created with Adobe Acrobat Pro, every user must also own Acrobat Pro to use it. This misconception often discourages developers from taking advantage of Acrobat JavaScript or designing sophisticated interactive PDF applications.

The reality is quite different.

Adobe Acrobat Pro is the development environment used to design, program, and maintain interactive PDF documents. Adobe Acrobat Reader, on the other hand, is the runtime environment used by the overwhelming majority of end users. Understanding this distinction is essential when designing forms intended for widespread distribution.

Investigation

From the very beginning of the EasyPDF™ project, one objective remained constant:

The customer should never be required to purchase expensive software simply to use an interactive PDF form.

Adobe Acrobat Reader is available at no cost and is already installed on millions of desktop and laptop computers worldwide. More importantly, Reader supports a surprisingly large portion of the Acrobat JavaScript object model.

Many developers underestimate Reader's capabilities because they assume JavaScript is reserved for Acrobat Pro. While certain security-sensitive operations remain exclusive to Acrobat Pro, the vast majority of the functionality required to build intelligent forms—including calculations, validation, formatting, event handling, drop-down lists, hidden fields, document-level JavaScript, and dynamic user interfaces—is fully supported in Adobe Acrobat Reader.

As a result, EasyPDF™ Forms were designed from the outset to execute entirely within Reader while using Acrobat Pro only during development.

Solution

The development model adopted throughout EasyPDF™ is straightforward:

- **Use Adobe Acrobat Pro to design, edit, and program the PDF.**
- **Use Adobe Acrobat Reader as the primary environment for testing and customer use.**
- **Avoid features that require Acrobat Pro unless they are strictly development tools.**
- **Verify every significant change in Reader before considering the form complete.**

This philosophy ensures that every customer experiences the form exactly as intended without purchasing additional software.

© Marty Karr

Why It Works

Separating the development environment from the execution environment produces several important advantages.

The developer benefits from the full design and programming capabilities available in Acrobat Pro while the customer benefits from a free, familiar application requiring no additional investment.

Because the forms execute entirely within Reader, there are no external databases, no cloud subscriptions, no plugins, and no proprietary runtime software to install. The PDF itself becomes the complete application.

This approach also simplifies support because every customer is using essentially the same runtime environment.

Design Principle

Develop for Acrobat Pro.

Deliver for Adobe Acrobat Reader.

Never confuse the development tool with the product your customer actually uses.

Lessons Learned

One of the biggest surprises encountered during the development of EasyPDF™ Forms was discovering just how capable Adobe Acrobat Reader really is.

Many developers dismiss Reader as little more than a document viewer. In reality, Reader supports enough of Acrobat JavaScript to build sophisticated, responsive applications when those applications are designed within Reader's supported capabilities.

The challenge is not overcoming Reader's limitations.

The challenge is understanding where those limitations actually exist and designing accordingly.

EasyPDF™ Engineering Principle

Never require the customer to purchase the developer's tools merely to use the finished product. Build with Acrobat Pro. Deliver with Adobe Acrobat Reader.

© Marty Karr

7.1.1 Selecting the Most Reliable Acrobat Property

Problem

While implementing show/hide navigation arrow buttons for several EasyPDF™ forms, including Smart Invoice, CRM, Digital Rolodex, and Medication Manager, the Acrobat display property produced inconsistent visual results. Although the underlying JavaScript executed correctly, the navigation buttons occasionally failed to update their visible state after records were added, deleted, or restored.

Because the script logic itself was functioning correctly, the inconsistent user interface behavior made the problem particularly difficult to diagnose.

Investigation

Several revisions of the navigation button scripts were tested using the Acrobat display property. Although each revision improved portions of the implementation, the button visibility remained inconsistent under certain conditions.

Further testing indicated that the navigation logic was not at fault. Instead, the problem appeared to be related to Acrobat's handling of the display property for dynamically changing button visibility.

Replacing the display property with the hidden property immediately produced consistent and repeatable behavior across all EasyPDF™ forms.

Solution

The navigation buttons are now shown and hidden using the Acrobat hidden property instead of the display property.

Example

```
var bHide = (fNames.numItems <= 1);

fPrev.hidden = bHide;
fNext.hidden = bHide;
```

The button update routine should be called whenever the contents of the associated drop-down list are rebuilt.

Why It Works

Although both properties control field visibility, practical testing demonstrated that the hidden

© Marty Karr

property provides more reliable results when dynamically updating Acrobat button fields.

Using the hidden property also simplifies the implementation by eliminating unnecessary display state management.

Design Principle

Whenever possible, use the simplest Acrobat property that consistently produces the desired result.

For dynamically updating button visibility, the hidden property proved to be both simpler and more reliable than the display property.

Lessons Learned

• Do not assume Acrobat properties with similar purposes behave identically.

• A simpler solution is often the more reliable solution.

• When a user interface behaves inconsistently, consider the Acrobat property being used before rewriting otherwise correct program logic.

• Once a reliable implementation has been verified, standardize its use throughout the entire application.

Historical Note

This behavior was discovered while implementing show/hide navigation arrow buttons for multiple EasyPDF™ forms. After extensive testing, the hidden property was adopted as the standard implementation for all future EasyPDF™ projects.

7.1.2 Centralizing update?Btns()

Problem

While implementing show/hide navigation buttons throughout the EasyPDF™ family of forms, the initial approach called update?Btns() from numerous locations within the application. Button updates were performed after adding records, deleting records, restoring deleted records, clearing fields, and in several other routines whenever it seemed necessary.

Although this approach worked, it resulted in duplicated code, increased maintenance, and made it more difficult to determine whether every possible update path had been covered.

Investigation

A review of the application flow revealed that navigation buttons do not actually depend on whether a record is added, deleted, restored, or edited.

Instead, the button visibility depends upon only one condition:

The current contents of the associated drop-down list.

Further analysis showed that every operation capable of changing the drop-down list ultimately calls updateNames() to rebuild the list.

This observation suggested that the responsibility for updating the navigation buttons belonged inside updateNames() rather than throughout the individual application routines.

Solution

The update?Btns() function was moved into updateNames().

Whenever updateNames() rebuilds the drop-down list, it immediately updates the navigation buttons before returning control to the calling routine.

As a result, add(), delete(), restore(), and similar functions no longer require individual calls to update?Btns().

This centralizes all button updates into a single location.

Why It Works

The visibility of the navigation buttons is determined solely by the contents of the drop-down list.

Since updateNames() is responsible for rebuilding that list, it naturally becomes the single location

© Marty Karr

responsible for updating the associated navigation buttons.

This approach eliminates redundant function calls while ensuring that button visibility always remains synchronized with the current drop-down contents.

Design Principle

Whenever multiple routines produce the same user interface update, centralize that update into the single function responsible for maintaining the affected user interface component.

This reduces duplicated code, simplifies future maintenance, and minimizes the possibility of inconsistent behavior.

Lessons Learned

• Eliminate duplicate function calls whenever a single centralized location can perform the same task.

• A function should be responsible for maintaining the user interface element that it creates or modifies.

• Simplifying program flow generally improves long-term maintainability.

• Well-defined responsibilities make debugging considerably easier.

Historical Note

This design improvement was discovered while integrating show/hide navigation buttons into multiple EasyPDF™ forms, including the CRM System, Digital Rolodex, Smart Invoice, and Medication Manager.

Once the update routine was centralized, the implementation became both simpler and more reliable, and the technique was adopted as the EasyPDF™ standard for all future projects.

EasyPDF™ Engineering Principle

When a user interface element is maintained by a single function, all related updates should occur within that function rather than being scattered throughout the application.

© Marty Karr

7.1.3 – Separating Trial Limits

Problem

During development of the EasyPDF™ Forms product line, trial-version restrictions were sometimes added directly inside the application functions they affected.

For example, a customer-management application might limit the trial version to three customer records. The quickest way to enforce that restriction would be to place the trial check directly inside the addCustomer() function:

```
function addCustomer() {
if (bTrial && getCustomerCount(oCustomers) >= 3) {
app.alert(
"The trial version allows a maximum of three customers."
);
return;
}

// Continue adding the customer...

}
```

Although this works, the addCustomer() function now performs two unrelated responsibilities.

Its primary purpose is to add a customer record. However, it has also become responsible for determining whether the user's license permits that operation.

As additional trial restrictions are introduced, the same problem can spread throughout the application. Add, edit, delete, save, restore, validation, and navigation functions may gradually become cluttered with licensing conditions.

This makes the application more difficult to read, test, maintain, and convert into a licensed version.

Investigation

The underlying problem was not the trial limit itself. The problem was where the trial-limit logic had been placed.

Trial restrictions are not part of the application's core business logic.

A customer-management function should manage customers. An invoice function should manage invoices. A medication function should manage medications.

© Marty Karr

Whether the user is allowed to perform an operation under a trial license is a separate concern.

Embedding trial restrictions directly inside the application's primary functions creates several maintenance problems:

• The same trial condition may be repeated in multiple locations.

• Trial-limit values may become hard-coded throughout the document.

• Licensing messages may become inconsistent.

• Changing a trial limit may require editing several unrelated functions.

• Removing trial restrictions from the licensed version becomes more difficult.

• Application functions become responsible for both record management and license enforcement.

The investigation therefore led to a broader design question:

Should the application functions themselves enforce trial restrictions, or should they ask a separate licensing function whether an operation is permitted?

The second approach proved cleaner and more maintainable.

Solution

The trial restriction should be moved into a dedicated helper function whose only responsibility is to determine whether the operation is permitted.

For example:

var TRIAL_CUSTOMER_LIMIT = 3;

function canAddCustomer(oCustomers) {
if (!bTrial)
return true;

```
if (getCustomerCount(oCustomers) >= TRIAL_CUSTOMER_LIMIT) {
    app.alert(
        "The trial version allows a maximum of " +
        TRIAL_CUSTOMER_LIMIT +
        " customer records.\n\n" +
        "Please purchase the licensed version to continue."
    );
```


© Marty Karr

```
    return false;
}

return true;

}
```

The addCustomer() function can then remain focused on its intended responsibility:

```
function addCustomer() {
if (!canAddCustomer(oCustomers))
return;

// Continue adding the customer...

}
```

The addCustomer() function no longer needs to know:

• Whether the document is a trial version.

• What the trial customer limit is.

• How customer records are counted.

• What message should be displayed when the limit is reached.

Those responsibilities now belong entirely to canAddCustomer().

The same pattern can be applied to other record types:

```
if (!canAddInvoice(oInvoices))
return;

if (!canAddContact(oContacts))
return;

if (!canAddMedication(oMedications))
return;

if (!canAddPassword(oPasswords))
return;
```

Each helper function should enforce one clearly defined licensing rule.

© Marty Karr

Applications containing more than one stored-record type should also use separate helper functions for each trial restriction.

For example, an EasyPDF™ Medication Manager trial version might allow one patient record while permitting several medication records for that patient.

Those limits should not be combined into one general trial-limit function because they apply to different data collections and represent different licensing rules.

var TRIAL_PATIENT_LIMIT = 1;
var TRIAL_MEDICATION_LIMIT = 5;

function canAddPatient(oPatients) {
if (!bTrial)
return true;

```
if (getPatientCount(oPatients) >= TRIAL_PATIENT_LIMIT) {
    app.alert(
        "The trial version allows a maximum of " +
        TRIAL_PATIENT_LIMIT +
        " patient record."
    );

    return false;
}

return true;

}
```

function canAddMedication(oMedications) {
if (!bTrial)
return true;

```
if (getMedicationCount(oMedications) >= TRIAL_MEDICATION_LIMIT) {
    app.alert(
        "The trial version allows a maximum of " +
        TRIAL_MEDICATION_LIMIT +
        " medication records."
    );

    return false;
}

return true;

}
```


© Marty Karr

The application functions then call only the helper associated with the operation being performed:

```
function addPatient() {
if (!canAddPatient(oPatients))
return;

// Continue adding the patient...

}

function addMedication() {
if (!canAddMedication(oMedications))
return;

// Continue adding the medication...

}
```

This keeps the patient limit independent of the medication limit and prevents one trial restriction from unintentionally affecting another part of the application.

Design Principle

Trial-version restrictions are licensing logic, not application logic.

Every trial restriction should be isolated inside a dedicated, clearly named helper function.

The application's primary functions should ask whether an operation is permitted, but they should not contain the detailed rules used to make that decision.

A function such as addCustomer() should add a customer.

A function such as canAddCustomer() should determine whether the license permits another customer to be added.

Keeping those responsibilities separate produces code that is easier to understand, modify, test, and reuse.

Lessons Learned

The quickest place to add a trial restriction is not necessarily the correct place.

Embedding trial checks directly inside add, edit, delete, save, restore, validation, or navigation functions may work initially, but it gradually mixes licensing concerns with application behavior.

© Marty Karr

The better approach is to:

• Store trial limits in named constants.

• Create one helper function for each licensing rule.

• Keep trial messages inside the licensing layer.

• Return a simple true or false result to the calling function.

• Keep application functions independent of trial-version details.

• Use separate helpers for separate record types and limits.

This design also makes licensed-version development easier.

When the licensed version does not require trial restrictions, the licensing helpers can return true, the trial flag can be disabled, or the trial-control layer can be removed without rewriting the application's core functions.

Historical Note

This principle became increasingly important as the EasyPDF™ Forms product line expanded beyond simple forms into JavaScript-driven PDF applications containing customer records, contacts, invoices, medications, passwords, and other stored data.

Early trial restrictions could be added quickly to individual functions. However, as applications became more sophisticated, that approach risked spreading trial-related conditions throughout the codebase.

Separating trial limits into dedicated helper functions established a cleaner boundary between the application and its licensing controls.

That boundary made it easier to maintain both trial and licensed editions while preserving one stable body of application logic.

The lesson was straightforward but important:

Do not redesign the application around the trial version.

Build the application correctly first, and place the trial restrictions around it as a separate licensing layer.

© Marty Karr

7.1.4 Excluding Counters from Drop-Down Lists

Problem

Several EasyPDF™ forms maintain internal counters within their JSON data store to automatically assign unique record identifiers and maintain application state. Examples include customer IDs, invoice numbers, patient IDs, contact IDs, and similar values that are required internally but should never appear as selectable items within user drop-down lists.

During development it became apparent that rebuilding a drop-down list by simply looping through every property contained within the JSON object unintentionally included these internal counters as selectable entries. Although the application continued to function, the user interface displayed information intended solely for internal program use.

Investigation

Initially, the drop-down rebuilding routines assumed every top-level JSON property represented a valid user record. This assumption proved incorrect as additional application features introduced internal properties such as nextCustomerID, nextInvoiceNo, nextPatientID, nextContactID, and similar counters.

Because these values existed alongside the user records within the JSON structure, they were inadvertently treated as record names whenever the drop-down list was rebuilt.

As additional EasyPDF™ forms adopted the same storage architecture, it became evident that filtering these internal properties individually within each application was neither efficient nor maintainable.

Solution

The updateNames() routines were modified to distinguish between actual user records and internal application properties before populating the drop-down list.

Only valid record objects are added to the list. Internal counters and other application variables are ignored during the rebuilding process.

By centralizing this filtering logic within updateNames(), every EasyPDF™ form benefits from the same reliable behavior without requiring application-specific exceptions elsewhere in the code.

Why It Works

Drop-down lists exist solely for user interaction.

Internal counters exist solely to support application logic.

© Marty Karr

Separating these responsibilities ensures the user interface presents only meaningful record names while preserving all application metadata within the JSON storage structure.

The result is a cleaner interface, more maintainable code, and a reduced likelihood of future programming errors whenever additional internal properties are added to the data store.

Design Principle

Internal application data should remain internal.

Whenever JSON storage contains both user records and application metadata, user interface routines should explicitly filter the data presented to the user rather than assuming every stored property represents a selectable record.

Lessons Learned

• Never assume every JSON property belongs in a user interface control.

• Internal counters should remain invisible to the user.

• Centralize filtering logic within updateNames() rather than scattering exceptions throughout the application.

• Separating application metadata from user-visible data produces a cleaner and more maintainable design.

Historical Note

This design improvement originated during the development of several EasyPDF™ forms after global counters were introduced to automatically generate unique record identifiers. Rather than redesigning the storage architecture, the updateNames() routines were enhanced to recognize and exclude internal counters while rebuilding the associated drop-down lists. The solution proved reliable and was subsequently adopted as the EasyPDF™ standard across all projects.

EasyPDF™ Engineering Principle

User interface controls should display only information intended for the user. Internal application data should remain hidden regardless of how it is stored.

© Marty Karr

7.1.6.1 – Why We Copy Existing PDF Forms

Problem

As Acrobat developers gain experience, they naturally begin reusing previously developed PDF forms as the foundation for new projects. Rather than creating every form field, button, hidden data field, JavaScript object, and supporting script from scratch, it is far more efficient to duplicate an existing PDF and remove or modify features no longer required.

Initially, this approach appears to offer the best of both worlds. Existing layouts have already been refined, common helper functions have already been written and tested, and much of the repetitive work has already been completed. Depending upon the complexity of the original application, reusing an existing PDF can save many hours—or even days—of development time.

For over thirty years, this was my preferred method of developing new EasyPDF™ interactive PDF forms.

Unfortunately, I eventually discovered that copying an existing PDF form also copies something I had never considered.

Investigation

Like most developers, I initially viewed an Acrobat PDF as little more than a collection of pages, form fields, buttons, and JavaScript. My assumption was simple:

Copy the existing PDF.

Remove what is no longer needed.

Add the new features.

From a development standpoint, the process appeared completely logical.

After all, why recreate components that had already proven reliable?

What I failed to recognize (essentially unaware) was that Acrobat applications gradually accumulate years of development history. Temporary fixes, experimental code, hidden storage fields, helper functions, abandoned features, document-level scripts, validation routines, calculation logic, and countless small design decisions quietly become part of the application itself.

The problem presents itself given many of these components are invisible during normal use and remain part of the PDF long after their original purpose has been forgotten.

Every time I copied one of my existing PDF forms, I unknowingly copied that history as well.

Solution

The solution was not to abandon template reuse altogether. However, while reusing existing PDF forms still remains one of the most effective ways to accelerate development and avoid repeatedly

© Marty Karr

solving problems that have already been addressed, experience has taught me there are occasions when inherited form fields, scripts, or hidden relationships become so intertwined that continuing to reuse them creates more work than starting over. In several EasyPDF™ development projects, I ultimately abandoned the copied form fields altogether and recreated them from scratch. Although this ended up being the more time-consuming approach, it often proved to be the faster, cleaner, and more reliable solution. The challenge is recognizing when you've reached that point.

Why It Works

Once developers recognize that a copied PDF contains far more than its visible pages and fields, they naturally begin looking beneath the surface.

Hidden fields are examined.

Document-level scripts are reviewed.

Helper functions are verified.

Unused variables are questioned.

Obsolete code is identified before new development begins rather than after unexpected behavior appears.

Instead of inheriting unknown behavior, the developer regains control of the application from the very beginning.

Design Principle

Never assume a copied PDF form is a clean starting point.

Every copied Acrobat application carries with it the design decisions, experiments, shortcuts, temporary fixes, and hidden dependencies accumulated throughout its development history.

Treat every copied PDF as inherited software—not merely a reusable template.

Lessons Learned

For decades, I believed I was copying completed PDF forms.

In reality, I was copying entire software applications.

That realization explained countless mysterious problems that had previously consumed hours—and sometimes days—of unnecessary debugging.

Once I understood what I was actually inheriting, my entire approach to Acrobat development changed.

© Marty Karr

EasyPDF™ Engineering Principle

Reuse existing PDF forms to save development time—but never assume what you copied is only what you can see. Before adding new features, first understand exactly what you've inherited.

© Marty Karr

7.1.6.2 – Every Copied PDF Has a History

Problem

One of the biggest misconceptions I held during my early years of Acrobat development was believing that copying an existing PDF form simply provided a convenient starting point for a new application. I viewed the copied PDF as little more than a collection of reusable pages, form fields, and JavaScript code that could be modified to suit the needs of the next project.

Only much later did I realize that every copied PDF carries with it the complete history of its own development. Every enhancement, temporary workaround, design decision, abandoned feature, experiment, correction, and forgotten modification becomes part of the application being copied. Whether remembered or not, that history continues to exist beneath the surface.

The danger is not that a PDF has a history. Every software application develops one. The danger lies in forgetting that the history still exists after the application has been copied.

Investigation

Unlike conventional source code, where developers often build new projects from individual modules or libraries, Acrobat applications frequently evolve by copying an existing PDF and modifying it into something new. Initially, this approach appears both logical and efficient. After all, much of the previous work has already been completed and tested.

However, every time a PDF application is expanded, revised, repaired, or enhanced, another layer of development history is added. Over months or even years, these changes accumulate into a collection of decisions that few developers can completely remember.

Eventually, I realized I was no longer copying only form fields and JavaScript. I was unknowingly inheriting years of accumulated engineering decisions—many of which I had completely forgotten making in the first place.

Solution

While reusing existing PDF forms often provide an effective way to accelerate development and avoid repeatedly solving problems that have already been addressed, the real solution had been staring me in the face for years. What I had failed to recognize was that every interactive PDF form carries the complete history of its own development.

Rather than treating it as a clean template, I learned to approach it as inherited software with its own developmental history. Before adding new features, I began asking what already existed beneath the surface. Every hidden field, helper function, document-level script, validation routine, calculation, and object deserved to be reexamined—not because it was necessarily wrong, but because I could no longer assume I remembered why it had originally been created—or whether it

© Marty Karr

still belonged there at all. That simple shift in perspective fundamentally changed the way I approached every new EasyPDF™ application.

Why It Works

Viewing every copied PDF as inherited software changes the developer's mindset from simply reusing previous work to understanding previous work.

Instead of assuming every existing component still serves a useful purpose, each inherited element becomes something to verify, understand, and intentionally preserve—or remove.

This approach significantly reduces the likelihood of unintentionally carrying forgotten behavior into a new application while preserving the valuable engineering experience already invested in the original PDF.

Design Principle

Every software application tells the story of its own evolution.

Some chapters of that story represent thoughtful design improvements, carefully tested features, and reliable solutions to previous problems. Other chapters consist of temporary fixes, abandoned ideas, obsolete code, and modifications whose original purpose may no longer be remembered.

Copying a PDF copies both.

Good engineering begins by recognizing the difference.

Lessons Learned

One of the most important lessons I learned after decades of Acrobat development is that forgotten history never disappears simply because I no longer remember it. During the redevelopment of one EasyPDF™ form, I eventually reached the point where years of accumulated history had become so deeply embedded within the inherited PDF form fields that troubleshooting the application was no longer practical. Regardless of the original cause, I abandoned every inherited PDF form field and recreated them all from scratch. Although this initially appeared to require more work, it consistently proved to be the cleaner, more reliable, and ultimately faster solution. From that point forward, whenever inherited PDF form fields reached that level of uncertainty, I no longer attempted to salvage them—I rebuilt them.

EasyPDF™ Engineering Principle

Every copied PDF inherits the complete history of the application from which it came. Before building upon that foundation, first understand what you have inherited. Forgotten history has a way of becoming tomorrow's unexpected problem.

© Marty Karr

7.1.6.3 – Understanding Application Baggage

Problem

One of the most overlooked realities of Acrobat development is that a PDF application silently accumulates baggage over time. Unlike a traditional software project where obsolete code may be easier to identify, Acrobat documents often retain objects, scripts, fields, and calculations long after they have served their original purpose.

The problem is that most of this baggage remains completely invisible during normal development.

A form may appear perfectly clean when viewed on screen while internally containing years of accumulated remnants left behind during previous revisions. Hidden fields, abandoned buttons, obsolete calculations, unused document-level functions, renamed objects, temporary debugging scripts, and forgotten experiments often remain embedded within the document long after their original purpose has been forgotten.

Because the application continues to function normally, these hidden artifacts are rarely questioned. They simply become part of the application's landscape.

Unfortunately, every hidden object becomes another unknown variable future development must contend with.

Investigation

During the development of numerous EasyPDF™ forms, I gradually came to appreciate that Acrobat rarely removes anything unless the developer deliberately removes it.

As new features were added, existing features modified, and older functionality replaced, the PDF quietly retained many of the objects created during earlier stages of development.

Some remained visible while many did not.

Opening Acrobat's field list often revealed numerous fields that no longer appeared anywhere within the document itself. Hidden buttons, temporary calculation fields, experimental controls, helper objects, and supporting fields frequently remained even though the user would never know they existed.

The same proved true for JavaScript.

Document-level functions written years earlier sometimes remained inside the application even after newer routines had replaced them. Temporary debugging code occasionally survived long after the original debugging session had ended. Calculations once required by an earlier design quietly remained in place even though they were no longer referenced by any visible field.

None of these objects necessarily caused immediate problems.

That is precisely what makes application baggage so deceptive.

© Marty Karr

It quietly accumulates without demanding attention.

Months become years.

Eventually, no one remembers why many of those objects still exist.

Solution

Rather than assuming older applications remain clean simply because they continue functioning correctly, I adopted a different mindset.

Every mature Acrobat application should be viewed as carrying historical baggage until proven otherwise.

Whenever substantial revisions are performed, developers should periodically inspect the document itself rather than focusing exclusively on the visible pages.

Questions worth asking include:

- Does this field still serve a purpose?
- Is this document-level function still being called?
- Is this calculation still required?
- Does this hidden button perform any remaining task?
- Was this object left behind during an earlier design revision?

Removing obsolete objects does more than reduce clutter.

It simplifies future maintenance while reducing opportunities for unexpected interactions between older and newer code.

Clean applications are generally easier to understand, easier to debug, and considerably easier to maintain.

Why It Works

Every object inside an Acrobat document consumes something far more valuable than storage space.

It consumes understanding.

Future debugging becomes increasingly difficult when developers must determine whether an object remains important or merely survived years of previous revisions.

Every unnecessary field increases uncertainty.

Every obsolete function requires investigation.

Every forgotten calculation becomes another possible source of unexpected behavior.

© Marty Karr

The larger the application grows, the greater this maintenance burden becomes.

Removing unnecessary baggage gradually restores clarity to the application's internal structure.

Developers spend less time asking *"Why is this here?"* and more time improving the application itself.

Design Principle

One lesson became increasingly clear throughout the development of EasyPDF™ forms.

Applications do not become difficult to maintain overnight.

They become difficult to maintain one forgotten object at a time.

Application baggage is rarely created through poor programming.

More often, it is the natural result of years of legitimate development.

Features evolve.

Requirements change.

Customers request enhancements.

Earlier design decisions become obsolete.

Temporary workarounds become permanent simply because they continue functioning.

Without periodic housekeeping, these remnants quietly accumulate until understanding the application's internal structure becomes significantly more difficult than writing the next feature.

Maintaining a clean application is therefore not a one-time task.

It is an ongoing engineering responsibility.

Lessons Learned

Application baggage is not always visible.

In many cases, the most significant complexity within an Acrobat application exists entirely behind the scenes.

A form that appears simple to the user may contain years of accumulated development history hidden beneath its polished interface.

Experienced developers eventually learn that maintaining an application involves more than adding new functionality.

It also requires periodically removing functionality that no longer belongs.

Every obsolete object removed today becomes one less mystery to investigate tomorrow.

Good software engineering is measured not only by what remains inside an application, but also by what has been thoughtfully removed.

© Marty Karr

EasyPDF™ Engineering Principle

Every Acrobat application carries its own development history. Hidden fields, obsolete scripts, abandoned calculations, and forgotten objects silently accumulate over time, increasing maintenance complexity long after their original purpose has disappeared. Effective software engineering requires periodically identifying and removing this application baggage so future development remains understandable, maintainable, and reliable.

© Marty Karr

7.1.6.4 – Hidden Dependencies

Problem

One of the most dangerous assumptions a developer can make is believing that every object, function, or script within an Acrobat application serves an obvious and independent purpose.

In reality, many components exist solely to support something else.

A helper function may execute only during document initialization. A hidden field may quietly store information later used by multiple calculations. A document-level routine may appear dormant for months until a specific sequence of events suddenly requires it.

The problem is that these relationships are rarely visible.

A developer sees only the individual object—not the network of dependencies quietly surrounding it.

As applications mature, this makes seemingly harmless cleanup one of the riskiest maintenance tasks. Removing an object simply because its purpose is no longer obvious can unintentionally disable functionality elsewhere in the application.

The most dangerous dependencies are often the ones you never knew existed.

Investigation

One incident during the development of an EasyPDF™ form permanently changed the way I viewed application maintenance.

Months earlier, I had introduced a timeout routine to resolve a particular behavioral problem. The solution worked, the application remained stable, and over time I simply stopped thinking about it.

Later, while reviewing the application's code, I noticed the timeout routine once again.

At first glance, it appeared unnecessary.

The application had been functioning correctly for quite some time, and nothing about the surrounding code suggested the timeout continued performing an important task. Believing it had simply become obsolete, I removed it as part of what I thought was routine code cleanup.

Initially, everything appeared normal.

Only after additional testing did the application begin exhibiting the very behavior the timeout routine had originally been written to prevent.

The connection wasn't immediately obvious.

The timeout routine looked unrelated to the problem now appearing elsewhere in the application, making it easy to assume the cause must lie somewhere else. Only after restoring the original code did the application immediately return to its expected operation.

© Marty Karr

That experience taught me an unforgettable lesson.

The timeout routine had not merely solved an earlier problem.

It had quietly become part of the application's internal behavior.

Although the code appeared inactive, removing it exposed a dependency that had remained hidden until the moment it disappeared.

Solution

From that point forward, I stopped evaluating objects individually and began evaluating relationships.

Before removing a field, function, calculation, or script, I learned to ask several important questions.

- Why was this originally added?
- Does another routine depend upon it?
- Does it execute only under specific conditions?
- Is it preserving application state behind the scenes?
- Am I removing something simply because I no longer remember its purpose?

Only after answering those questions could I confidently decide whether an object truly belonged in the application.

Sometimes the investigation confirmed that the object had become obsolete.

Just as often, however, it revealed relationships that would never have been discovered through visual inspection alone.

Why It Works

Software rarely fails because an individual object is defective.

More often, it fails because a relationship between two objects has been unintentionally broken.

Hidden dependencies are particularly deceptive because they frequently disguise themselves as unnecessary code.

A helper function that rarely executes.

A variable that seldom changes.

A timeout routine whose original purpose has long since been forgotten.

Individually, each appears insignificant.

Collectively, they may represent critical pieces of the application's internal behavior.

Once those relationships are understood, debugging changes dramatically.

© Marty Karr

Instead of asking, "Can I remove this?"

Experienced developers begin asking,

"Who still depends upon it?"

That single change in perspective prevents countless hours of unnecessary troubleshooting.

Design Principle

As EasyPDF™ applications continued to evolve, one engineering principle became increasingly clear.

Applications should never be viewed as collections of independent objects.

They are interconnected systems.

Every feature introduces relationships.

Every improvement potentially creates new dependencies.

Some remain obvious.

Others quietly disappear into the background until years later when a well-intentioned cleanup effort unknowingly removes something still performing useful work.

Good engineering therefore requires understanding relationships every bit as much as understanding code.

Deleting an object is easy.

Understanding everything connected to it is considerably more difficult.

Lessons Learned

Hidden dependencies rarely announce themselves.

Instead, they patiently wait until someone removes or modifies an object believed to be unnecessary.

Over the years, I learned that code whose purpose is no longer obvious deserves far more respect than code whose purpose is immediately understood.

The timeout routine taught me that appearances can be deceiving.

The fact that I no longer remembered why the code existed did not mean the application had stopped depending upon it.

From that point forward, I adopted a simple rule.

Never remove code simply because it appears inactive.

First understand why it was written.

Then determine whether anything still depends upon it.

© Marty Karr

Only then should its future be decided.

That single habit prevented far more problems than any amount of debugging ever could.

EasyPDF™ Engineering Principle

Every mature Acrobat application is an interconnected system. Fields, scripts, calculations, variables, and events frequently depend upon one another in ways that are not immediately visible. Code whose purpose is no longer obvious should never be assumed obsolete until its relationships have been fully understood. Experienced developers investigate before deleting because hidden dependencies often continue performing hidden work.

© Marty Karr

7.1.6.5 – The Biggest Revelation of All

The Problem

For more than thirty years I believed I understood Adobe Acrobat Pro exceptionally well. I knew its form fields, JavaScript, calculations, document actions, page events, and virtually every tool I relied upon to create sophisticated interactive PDF applications. Whenever something behaved unexpectedly, I assumed the cause almost certainly originated in my own code.

Looking back, the greatest mistake I have made throughout my Acrobat development career to date is believing that every unexplained behavior must have been caused by something I created.

It wasn't until I began collaborating with ChatGPT that I slowly came to appreciate something I had never fully considered throughout more than three decades of Acrobat development. Acrobat itself possesses an extensive internal framework whose behavior can influence a PDF application in ways that are often invisible to the developer. While this underlying framework generally functions exactly as intended, its accumulated history, legacy design decisions, compatibility mechanisms, and undocumented interactions can occasionally produce behavior that appears completely unrelated to the code being written.

In my opinion, this realization represents the single greatest revelation of my entire Acrobat development career.

The Investigation

This revelation did not occur overnight. It emerged gradually as seemingly unrelated problems began revealing a common pattern.

Earlier chapters described how mature PDF applications accumulate significant internal baggage over time. Hidden fields, document-level scripts, page actions, calculations, validation routines, bookmarks, links, metadata, and countless other components quietly become part of the document's internal structure. Many remain invisible during everyday development, yet they continue influencing how the application behaves.

One example involved a timeout routine that initially appeared to be unnecessary. Removing it seemed perfectly logical because the surrounding code no longer appeared to depend upon it. Yet removing the routine unexpectedly altered the application's behavior. The problem wasn't faulty JavaScript. The problem was an invisible dependency that had quietly evolved within the application over time.

Another example appeared while working with Acrobat's Edit Links feature. I initially assumed links behaved much like form fields and therefore should be copied and manipulated in similar ways. They were not. Eventually I discovered that links could indeed be copied, but they followed

© Marty Karr

an entirely different set of rules. The limitation wasn't my understanding of JavaScript. It was my incorrect assumption about how Acrobat itself implemented the feature.

Individually, each incident seemed unrelated. Collectively, they exposed a much larger truth. I wasn't simply debugging my own code anymore. I was also learning how Acrobat's internal framework influenced the behavior of my applications.

The Solution

The solution was not another JavaScript technique.

It was a fundamental change in how I approached debugging.

Instead of immediately asking:

"What did I do wrong?"

I learned to first ask:

"Is Acrobat itself contributing to what I'm seeing?"

That single question completely changed my investigative process.

Sometimes the answer remained exactly what I expected. The problem genuinely originated in my own code.

Other times, however, the behavior resulted from Acrobat's own implementation, accumulated document history, legacy compatibility features, or internal mechanisms that simply were not obvious during normal development.

Once I stopped assuming every unexplained behavior belonged to me, solving difficult problems became significantly easier.

Why It Works

Every mature software platform eventually develops its own personality.

Years of development introduce new features while preserving compatibility with older ones. Internal frameworks evolve. Legacy behaviors remain. New capabilities are often layered upon existing architecture rather than replacing it entirely.

Adobe Acrobat is no exception.

Its remarkable backward compatibility is one of its greatest strengths, allowing PDF documents created many years ago to continue functioning today. Yet that same longevity also means Acrobat contains decades of engineering decisions working together beneath the surface.

Most developers rarely notice this internal framework because, under normal circumstances, everything simply works.

When something doesn't, however, that hidden framework suddenly becomes part of the debugging process whether we recognize it or not.

© Marty Karr

Understanding this distinction transforms debugging from an exercise in self-blame into a disciplined investigation of both the application and the platform executing it.

Lessons Learned

Perhaps the most valuable lesson I have learned is that experience alone does not guarantee complete understanding of a software platform.

After more than thirty years developing interactive PDF applications, I genuinely believed I understood Acrobat's behavior remarkably well. Yet some of the most valuable lessons I have learned arrived only after questioning assumptions I had carried throughout my entire career.

Many of the most difficult problems I encountered were never caused by poor programming.

Some originated from inherited document history.

Others resulted from hidden dependencies.

Still others reflected Acrobat's own implementation of features that behaved differently than expected.

Recognizing this distinction fundamentally changed how I troubleshoot every application I create today.

EasyPDF™ Engineering Principle

> **Before rewriting working code, determine whether the behavior originates from your application or from the platform executing it. Not every bug belongs to the developer. Some belong to the environment in which the application lives.**

Final Thoughts

One of the greatest dangers facing experienced developers is believing that years of experience eliminate the possibility of surprise.

In reality, the opposite is often true.

The longer we work with a mature platform, the easier it becomes to assume we already understand its behavior. Those assumptions eventually become invisible, quietly shaping every debugging decision we make.

The greatest mistake I have made throughout my Acrobat development career to date was believing every unexplained behavior originated in my own work.

Today, I know better.

> **The biggest revelation of my Acrobat development career was realizing that not every unexplained behavior originated in my own code. Sometimes the most important problem to solve is the one you didn't create**

© Marty Karr

7.1.6.6 – Knowing What to Keep

The Problem

One of the earliest lessons ChatGPT and I learned together involved one of my validation routines while redeveloping EasyPDF™ Smart Invoice. After reviewing my JavaScript, ChatGPT correctly observed that one of my validation scripts had gradually become much longer than necessary. Like many mature routines, it had evolved over years of development as new requirements, corrections, and safeguards were added whenever unexpected situations arose.

From a software engineering standpoint, shortening the routine appeared to be the obvious solution.

On paper, the recommendation made perfect sense.

Reality proved to be far more complicated.

Although the rewritten validation routine appeared cleaner and considerably shorter, it also introduced a surprising number of unexpected problems. Each rewrite corrected one issue only to reveal another. What initially appeared to be unnecessary complexity gradually revealed itself to be something entirely different.

The original routine wasn't simply long.

It contained years of accumulated experience.

The Investigation

As we continued refining the rewritten validation routine, our confidence gradually gave way to frustration. Despite numerous revisions, subtle problems continued appearing that had never existed in the original implementation.

Eventually I suggested something neither of us had initially expected.

"Let's restore my original validation script."

Almost immediately the application returned to the stable behavior it had exhibited for years.

That moment became an important turning point.

The issue had never been whether the original routine was perfect.

The issue was that we were attempting to improve something we did not yet fully understand.

The longer routine wasn't protecting itself.

It was protecting the application.

Every seemingly unnecessary statement represented a decision that had survived years of real-world testing. Some addressed bugs I had long forgotten. Others handled situations that only occurred under very specific circumstances. Still others quietly compensated for Acrobat behaviors discovered through experience rather than documentation.

© Marty Karr

Once we recognized that reality, our approach changed completely.

Instead of asking how to shorten the routine, we began asking why every section existed.

Only after answering that question were we able to simplify portions of the code without sacrificing the reliability the original routine had earned over time.

Ironically, both of us learned the same lesson.

Experience deserves the benefit of the doubt.

The Solution

The experience fundamentally changed how I evaluate mature code.

Today, whenever I encounter an older routine that appears unnecessarily long or overly defensive, I no longer assume it should immediately be rewritten.

Instead, I begin asking questions.

What problem was this code originally written to solve?

Has it quietly prevented problems for years?

Do I fully understand every decision represented within the routine?

If the answer to those questions is "not yet," I leave the code alone until I understand it completely.

Understanding now comes before improvement.

Not after it.

Paradoxically, this approach often produces cleaner software because improvements are based upon knowledge rather than assumptions.

Why It Works

Reliable software rarely becomes reliable by accident.

Every mature application represents hundreds—sometimes thousands—of engineering decisions accumulated over many years. Some remain obvious. Others quietly disappear beneath the surface as new features, corrections, and refinements are added.

Developers naturally admire elegant code.

There is nothing wrong with elegance.

However, elegance should never come at the expense of reliability.

Code that has successfully survived years of production use contains something far more valuable than attractive formatting.

It contains proven experience.

© Marty Karr

Once that experience is discarded without understanding why it existed, rebuilding it often becomes surprisingly difficult.

Knowing what to improve is important.

Knowing what to preserve is equally important.

Lessons Learned

Looking back, I no longer view our repeated validation script rewrites as wasted effort.

Quite the opposite.

They taught both of us something neither of us fully appreciated at the beginning.

ChatGPT learned that mature code often contains practical knowledge invisible to someone seeing it for the first time.

I learned that proven code is not beyond improvement—but improvement should only occur after its existing behavior is completely understood.

Eventually we accomplished exactly what we originally set out to do.

The validation routine became shorter, cleaner, and easier to maintain.

The difference was that we earned the right to simplify it.

We did not simply assume simplification was automatically an improvement.

That distinction has influenced every application I have developed since.

EasyPDF™ Engineering Principle

> Never rewrite working code simply because it appears longer than necessary. First understand why it has continued working successfully. Only then have you earned the right to improve it.

Final Thoughts

One of the greatest misconceptions in software development is believing that shorter code is automatically better code.

Sometimes it is.

Sometimes it isn't.

Experience has taught me that every mature application contains routines that quietly represent years of accumulated engineering knowledge. Their value cannot always be measured by their size, elegance, or appearance.

Today, I no longer judge existing code by how old it is or how long it has become.

© Marty Karr

I judge it by one simple question.

Has this code earned my trust?

If the answer is yes, I first seek to understand it before attempting to improve it.

Knowing how to write better code is an important skill.

Knowing what to keep is wisdom.

© Marty Karr

7.1.6.7 – A New Development Philosophy

The Problem

When I first began developing interactive PDF applications, I viewed software development primarily as a technical exercise. Success was measured by writing JavaScript that worked, designing intuitive forms, and solving one programming problem after another. Every challenge appeared to have a technical solution if I simply worked long enough to find it.

Over time, that perspective began to change.

As my applications became larger and more sophisticated, I discovered that many of the most difficult problems were no longer programming problems at all. They were problems of understanding. Every copied PDF carried a history. Mature applications accumulated hidden dependencies. Seemingly unnecessary code often protected against problems I had long forgotten. Occasionally, Acrobat itself contributed behavior that could easily be mistaken for programming mistakes.

The greatest challenge was no longer learning JavaScript.

The greatest challenge became learning how to think differently about software.

The Investigation

The lessons presented throughout this section did not emerge from a single project or a single discovery. They accumulated gradually over more than three decades of developing Acrobat applications.

I learned that applications evolve much like living systems. Every modification leaves behind traces of the decisions that preceded it. Features are added, bugs are corrected, workarounds become permanent, and assumptions quietly become part of the application's behavior. Eventually, distinguishing intentional design from accumulated history becomes increasingly difficult.

Working with ChatGPT reinforced many of these lessons.

On several occasions, we discovered that simplifying mature code before fully understanding it often introduced new problems. Together we also recognized that unexpected application behavior did not always originate in my JavaScript. Sometimes the application itself—or Acrobat's own internal behavior—played a role.

Each lesson challenged assumptions I had held for years.

Collectively, they reshaped the way I approach every development decision.

© Marty Karr

The Solution

Today, my development philosophy begins with understanding rather than modification.

When I encounter unexpected behavior, I no longer assume that my code is automatically at fault.

When I inherit mature code, I no longer assume that shorter automatically means better.

When I discover a hidden dependency, I no longer rush to eliminate it before understanding why it exists.

Instead, I begin by asking questions.

What is this application trying to tell me?

Why does this behavior exist?

What decisions came before mine?

Only after answering those questions do I begin making changes.

Ironically, this slower approach often produces faster progress because it avoids solving the wrong problem.

Understanding has become the foundation upon which every improvement is built.

Why It Works

Programming languages continue to evolve.

Software platforms continue to evolve.

Even Acrobat itself continues to evolve.

The one constant throughout that evolution is sound engineering judgment.

Technical knowledge teaches developers how to write software.

Engineering judgment teaches developers when to change software—and when not to.

That distinction cannot be learned from documentation alone.

It develops through investigation, mistakes, patience, and experience.

The longer I develop Acrobat applications, the more convinced I become that engineering judgment ultimately contributes more to software reliability than technical ability alone.

Lessons Learned

Looking back, I realize that the most valuable lessons of my Acrobat career rarely came from successfully writing new code.

They came from investigating failures.

They came from questioning assumptions.

© Marty Karr

They came from discovering that what initially appeared obvious often proved incomplete.

Most importantly, they came from accepting that software development is not simply about finding answers.

It is about learning to ask better questions.

That single realization has influenced every EasyPDF™ application I have developed.

It has also become the foundation upon which I continue building future applications.

EasyPDF™ Engineering Principle

> **Technical skill enables you to build software. Engineering judgment enables you to build software that endures. Develop your understanding before developing your solution.**

Final Thoughts

If there is one lesson I hope readers carry forward from this section, it is this:

Software development is not a race to write the most code, create the cleverest algorithms, or produce the shortest routines.

It is the continual pursuit of understanding.

Every application has a history.

Every mature program reflects countless engineering decisions.

Every unexpected behavior deserves investigation before correction.

The longer I develop Acrobat applications, the less interested I become in proving how much I know.

Instead, I have become far more interested in discovering what I have yet to understand.

That simple shift in perspective has changed the way I approach every project, every problem, and every line of code.

It is more than a programming technique.

It has become my development philosophy.

© Marty Karr

7.2.1 Event Sequence and Execution Order

Problem

One of the most common sources of frustration encountered by Acrobat JavaScript developers is determining why a script that appears to be correct does not always execute when expected. A field may contain the proper value while a calculated total remains unchanged, a validation routine may seem to execute too early or too late, or an interface update may not occur until after the user has completed another action.

These situations often lead developers to conclude that their JavaScript contains a programming error when, in reality, the script itself is functioning exactly as written. The problem lies not in the code, but in understanding when Acrobat chooses to execute that code.

Unlike traditional programming environments where statements generally execute in the order they are encountered, Acrobat is an event-driven application. Every user action initiates one or more events, and Acrobat processes those events according to its own predefined execution sequence. Understanding both the sequence of events and the order in which Acrobat executes them is essential to designing predictable, reliable interactive PDF applications.

Investigation

As increasingly sophisticated EasyPDF™ Forms were developed, it became apparent that Acrobat performs far more work than simply accepting a field value and immediately executing every related script.

Instead, Acrobat divides document processing into a series of individual stages. Depending upon the document design and field type, those stages may include:

- Keystroke processing
- Validation
- Formatting
- Value commitment
- Calculations
- User interface refresh

Each stage serves a specific purpose and must complete before Acrobat advances to the next. This processing model ensures that every event receives valid information from the previous stage before continuing.

For example, calculations cannot reliably execute until Acrobat has determined that user input is valid and committed the final field value. Likewise, formatting cannot display the final

© Marty Karr

appearance of a value until Acrobat knows that value has been accepted. User interface updates naturally occur after the document has reached a consistent state.

Recognizing this sequence explains why calculations, validation routines, or visual updates sometimes appear to execute later than expected even though the JavaScript itself is operating correctly.

Solution

Rather than assuming every operation should occur immediately after a field changes, design every script around the Acrobat event specifically intended to perform that task.

Use each event for the responsibility it was designed to perform.

For example:

- Use Keystroke events while the user is actively entering data.
- Use Validate events to determine whether the completed input should be accepted.
- Use Format events to control how accepted data is displayed.
- Use Calculate events to update dependent values only after the necessary information is available.
- Use Mouse Up events to perform operations initiated by push buttons after the user intentionally activates the control.

Avoid relocating code simply because another event appears to execute sooner. Instead, determine where the operation logically belongs within Acrobat's processing model and allow Acrobat to complete each stage in its normal execution order.

Applications designed around Acrobat's event architecture require fewer programming workarounds, are easier to debug, and produce far more predictable behavior.

Why It Works

Acrobat's event architecture was intentionally designed to separate document processing into individual, well-defined stages instead of attempting to perform every operation simultaneously.

By completing one event before beginning the next, Acrobat ensures that each stage operates on reliable information. Validation occurs before calculations depend upon user input. Formatting reflects committed values rather than temporary keystrokes. Interface updates occur only after the document has reached a consistent state.

© Marty Karr

This disciplined execution model protects document integrity while eliminating many timing-related problems that developers mistakenly attribute to JavaScript itself.

Once developers begin thinking in terms of Acrobat's processing sequence rather than individual scripts, debugging becomes significantly easier. Many seemingly complex problems disappear simply because the code has been moved to the event responsible for performing that operation.

Design Principle

Never design Acrobat applications around the assumption that code executes immediately after a field changes.

Instead, design every routine with the understanding that Acrobat—not the JavaScript—controls the sequence and timing of event execution.

Assign each programming task to the event specifically designed to perform that responsibility, then allow Acrobat to complete its normal processing sequence without interference.

Working with Acrobat's event model rather than attempting to bypass it produces applications that are simpler, more reliable, and considerably easier to maintain.

Lessons Learned

- Many apparent JavaScript failures are actually misunderstandings of Acrobat's event timing.
- Correct JavaScript executed during the wrong event frequently produces incorrect or unpredictable results.
- Acrobat—not the script—determines when each event executes.
- Selecting the proper event often eliminates unnecessary programming workarounds.
- Reliable interactive PDF applications are built by working with Acrobat's execution sequence rather than attempting to force operations to occur prematurely.
- Understanding when code executes is often just as important as understanding what the code does.

EasyPDF™ Engineering Principle

One of the most valuable lessons learned during the development of EasyPDF™ Forms is that Acrobat's event model should never be treated as an obstacle to overcome. It is the framework that governs every interactive document.

© Marty Karr

Before rewriting JavaScript that appears to be malfunctioning, first determine whether the routine is executing during the proper Acrobat event. More often than not, the solution is not additional code, but placing existing code in the event Acrobat intended for that purpose.

Understanding Acrobat's event sequence and execution order transforms debugging from a process of trial and error into one of logical analysis. Once developers learn to work with Acrobat's event architecture instead of against it, their applications become more predictable, more maintainable, and significantly easier to develop.

© Marty Karr

7.2.2.1 Validate Event

Introduction

When I first began developing Acrobat JavaScript forms, I tended to place code wherever it appeared to work. As my projects became more sophisticated, that approach gradually produced unnecessary duplication, inconsistent behavior, and increasingly difficult maintenance. One of the most valuable lessons I eventually learned was that the Validate event is often the best location for confirming user input and coordinating many of the tasks that follow.

Contrary to what its name might suggest, the Validate event is not limited to simply determining whether a value is acceptable. Properly used, it can become a central decision point that helps maintain data integrity while simplifying the overall design of a form.

Problem

Early versions of my forms often contained similar validation logic scattered throughout multiple events. One field might perform validation during a Keystroke event, another during a Format event, and still another after the field lost focus. While each solution appeared to work independently, maintaining the overall project gradually became more difficult.

As additional features were added, changing a simple validation rule frequently required modifying several separate scripts. This increased the likelihood of introducing inconsistencies or overlooking a location where the same logic had been duplicated.

The result was not only additional work but also code that became progressively harder to understand months or even years later.

Cause

The underlying problem was not Acrobat's event model itself. Instead, it was my tendency to distribute related logic across numerous events simply because Acrobat made those events available.

While each event serves a specific purpose, placing similar validation routines throughout multiple locations eventually creates unnecessary complexity. As projects grow, duplicated logic becomes one of the most common sources of maintenance problems.

I eventually realized that the issue was not validating too often—it was validating in too many different places.

Solution

Over time I adopted a much simpler design philosophy.

Whenever practical, I centralized field validation within the Validate event. Rather than allowing similar checks to appear throughout multiple event scripts, I treated the Validate event as the primary location for determining whether entered information should be accepted.

© Marty Karr

Once validation successfully completed, other document-level functions could safely assume the data was valid before performing additional operations such as saving records, refreshing user interface elements, or updating related information elsewhere in the document.

This approach significantly reduced duplicated code while making validation logic much easier to locate, understand, and maintain.

Another benefit was consistency. When every field followed the same general design philosophy, debugging became considerably easier because I always knew where validation decisions were being made.

Benefits

Centralizing validation within the Validate event produced several long-term advantages throughout my EasyPDF™ Forms.

- Reduced duplicated validation code.
- Simplified maintenance by keeping related logic together.
- Made debugging considerably easier.
- Produced more consistent behavior across different forms.
- Allowed document-level scripts to assume validated data before performing additional processing.
- Improved long-term readability of the code.

Perhaps the greatest benefit was not measured by the number of lines of code removed, but by how much easier the forms became to understand when revisiting them months or even years later.

Lessons Learned

One of the most important lessons I learned while developing EasyPDF™ Forms was that successful Acrobat programming is rarely about writing more code. It is usually about placing code in the most appropriate location.

The Validate event gradually became one of the cornerstones of my development philosophy because it provided a logical point at which user input could be confirmed before other processing occurred.

By centralizing validation instead of scattering similar routines throughout numerous events, my forms became easier to maintain, easier to debug, and far more consistent in their overall behavior.

Looking back, I consider this change in design philosophy to be one of the more significant improvements I made while developing commercial Acrobat JavaScript applications.

© Marty Karr

7.2.2.2 Format Event

Introduction

One of the first advantages I recognized when using the Format event was not how it helped me as a programmer, but how much time it saved the user. Rather than expecting users to enter dates, phone numbers, Social Security numbers, ZIP Codes, currency, and other information in a specific format, Acrobat could automatically display those values in a clean, professional format immediately after entry.

To me, this seemed like an obvious way to improve the user experience. Surprisingly, however, I found that many professionally produced PDF forms—including numerous government forms—failed to take advantage of this capability. Users were often expected to manually type punctuation, separators, and formatting that Acrobat could have handled automatically.

That realization reinforced one of the guiding principles behind EasyPDF™ Forms: whenever Acrobat can eliminate repetitive work for the user, it probably should.

Problem

As I worked with numerous commercial and government PDF forms, I was often surprised to find users still expected to manually enter punctuation and formatting for dates, phone numbers, ZIP Codes, and other common fields. Acrobat had long been capable of performing much of this work automatically. To me, requiring users to type information the software could easily format seemed like a missed opportunity to improve the overall user experience.

While I appreciated the ability to automatically format user input, I also recognized that the Format event had a very specific responsibility. Its purpose was to improve how information was presented to the user—not to perform validation or business logic. Keeping those responsibilities separate resulted in cleaner, more maintainable code as my applications grew in size and complexity.

As additional features were added, I found it increasingly important to let each Acrobat event perform its intended task. The Validate event confirmed user input, the Calculate event handled calculations, and the Format event focused solely on presentation. This separation of responsibilities simplified both debugging and long-term maintenance.

Cause

The confusion stemmed from misunderstanding the purpose of the Format event.

Unlike the Validate event, whose responsibility is determining whether entered information should be accepted, the Format event is primarily concerned with presentation rather than decision making.

Trying to use the Format event for validation or business logic often resulted in code that was more difficult to follow and maintain.

© Marty Karr

Solution

I eventually adopted a much simpler philosophy.

Whenever possible, I allowed the Validate event to determine whether data was acceptable and the Calculate event to perform calculations when required. Once those operations were complete, the Format event simply displayed the resulting value in an appropriate format.

Whether presenting dates, currency values, percentages, or other formatted information, I treated the Format event as a display layer rather than another location for business logic.

Separating these responsibilities made each event easier to understand while producing forms that behaved more consistently.

Benefits

Using the Format event primarily for presentation offered several advantages.

- Improved readability without altering stored values.
- Clear separation between validation, calculation, and presentation.
- Reduced unnecessary event interaction.
- Simplified debugging by limiting each event to a specific responsibility.
- Produced a more consistent user experience throughout my forms.

Although the Format event was not one of the most heavily used events in my applications, assigning it a well-defined role helped simplify the overall architecture of my projects.

Lessons Learned

One lesson became increasingly clear as my forms evolved.

Not every event needs to perform complex processing.

Sometimes the most effective design is allowing each event to perform one responsibility exceptionally well. By treating the Format event primarily as the presentation layer, I avoided unnecessary complexity while making the behavior of my forms easier to understand and maintain.

Looking back, I found that keeping formatting separate from validation and calculations resulted in cleaner code, fewer unintended side effects, and a much more organized application.

© Marty Karr

7.2.2.3 – Calculate Event

Introduction

The Calculate event is responsible for one of Acrobat's most powerful capabilities—automatically performing mathematical operations as users complete a PDF form. Rather than expecting users to manually calculate totals, taxes, balances, percentages, or other derived values, Acrobat can instantly perform these operations whenever the underlying data changes.

Throughout the development of EasyPDF™ Forms, I viewed the Calculate event as more than a way to perform arithmetic. It became another opportunity to eliminate repetitive work for the user. Every calculation Acrobat performed automatically was one less opportunity for a mathematical error and one less reason for users to reach for a calculator. By automating repetitive calculations, forms became faster to complete, more accurate, and considerably more professional.

Problem

Many PDF forms continue to require users to manually perform calculations before entering the results into the document. Whether totaling invoices, calculating sales tax, determining balances, or computing percentages, every manual calculation introduces another opportunity for error.

As calculations become more complex, a single mistake often affects several related values. An incorrect line-item total produces an incorrect subtotal, which then affects taxes and ultimately the grand total. Users should not be expected to verify every calculation when Acrobat can perform the work instantly and consistently.

Cause

Although Acrobat provides a dedicated Calculate event, many PDF developers either overlook its capabilities or use it only for simple arithmetic. Others distribute calculation logic throughout multiple field events, making applications increasingly difficult to understand and maintain as they grow in size.

Early in my own development, I also relied heavily on Acrobat's built-in Custom Calculation Scripts because they appeared to be the natural place for field calculations. While this approach worked well for smaller projects, I eventually discovered that larger business applications benefited from a more centralized design.

Solution

The Calculate event became the foundation for automating every repetitive mathematical operation within my forms. Line-item totals, subtotals, taxes, discounts, balances, percentages, and grand totals were all calculated automatically whenever the underlying data changed.

As my calculation-intensive applications evolved—particularly forms such as Smart Invoice and Estimate Worksheet—I gradually transitioned away from relying exclusively on Acrobat's individual Custom Calculation Scripts. Instead, I centralized much of the calculation logic within

© Marty Karr

document-level JavaScript functions that could be called whenever calculations needed to be updated.

This architectural change produced several important advantages. Calculation formulas existed in one location rather than being scattered among numerous calculated fields. Updating a formula required modifying a single function instead of searching throughout the document for multiple Custom Calculation Scripts. As applications became larger and more sophisticated, centralized calculation functions also simplified debugging, encouraged code reuse, and made future enhancements considerably easier to implement.

This approach also reinforced an important design principle that became increasingly evident throughout my development work. Validate events verified user input. Format events controlled presentation. Calculate events—or the document-level functions supporting them—performed the mathematics. Giving each component a clearly defined responsibility resulted in cleaner, more maintainable applications.

Benefits

Using the Calculate event and centralized calculation functions consistently throughout my applications provided numerous advantages:

- Eliminated manual arithmetic and calculator use.
- Reduced mathematical errors.
- Automatically updated dependent values whenever information changed.
- Improved user confidence by displaying accurate totals immediately.
- Centralized complex formulas for easier maintenance.
- Encouraged reusable document-level JavaScript functions.
- Simplified debugging by reducing duplicated calculation logic.
- Produced cleaner, more modular applications.

Perhaps the greatest benefit was that users rarely noticed the calculations themselves. They simply experienced forms that responded immediately, displayed accurate totals, and required less effort to complete.

Lessons Learned

One of the most important lessons I learned was that users should provide information—not perform mathematics. Whenever Acrobat can accurately perform repetitive calculations, it should.

I also learned that while Acrobat's built-in Custom Calculation Scripts work well for smaller forms, they are not always the best long-term solution for larger applications. As my projects became more sophisticated, centralized document-level calculation functions proved easier to maintain, easier to debug, and far more adaptable to future enhancements.

Today, I still appreciate the convenience of Acrobat's Calculate event. However, I now view it as part of a larger application architecture rather than an isolated field event.

© Marty Karr

7.2.2.4 – Blur Event

Introduction

The Blur event occurs whenever a field loses the input focus. Although it is often overlooked in favor of Validate, Format, or Calculate events, I found the Blur event to be one of Acrobat's most practical tools for improving the responsiveness of interactive PDF applications.

Unlike the Validate event, which determines whether user input is acceptable, the Blur event allows an application to react after the user finishes editing a field. This makes it an ideal location for updating related information, refreshing dependent fields, calling document-level JavaScript functions, and performing operations that should occur only after the user has completed their input.

As my applications became more sophisticated, the Blur event evolved into an important part of the overall application workflow, helping keep forms synchronized without interrupting the user's data entry.

Problem

One of the challenges of developing interactive PDF forms is determining when certain operations should occur. Updating information too early can interfere with user input, while waiting too long may leave displayed information out of date.

Many developers attempt to perform every operation within the Validate or Calculate events. While this may appear to work initially, it often results in unnecessary processing, duplicated logic, or event sequencing problems as applications become more complex.

Cause

The problem is not the Blur event itself but misunderstanding its intended purpose.

Each Acrobat event serves a specific role. Validate confirms user input. Format controls presentation. Calculate performs mathematical operations. The Blur event simply indicates that the user has finished working with a particular field and moved elsewhere.

Recognizing this distinction made it much easier to determine where various operations belonged.

Solution

During the development of my EasyPDF™ Forms, I originally performed many save and calculation operations immediately as data changed. As my applications evolved , I gradually changed that approach by allowing users to complete their data entry before triggering many of those operations from the Blur event. Once the field lost focus, document-level JavaScript functions could safely save records, refresh dependent calculations, update indexes, or synchronize related fields without interfering with Acrobat's normal event sequence. This seemingly small architectural change significantly improved application reliability while keeping individual field events simple and predictable.

© Marty Karr

Because the user had already completed editing the field, the Blur event also provided a natural point for refreshing dependent fields, updating indexes, synchronizing drop-down lists, enabling or disabling controls, or performing other application-specific tasks that were not strictly validation or calculation.

Benefits

Using the Blur event consistently throughout my applications provided several important advantages:

- Allowed processing to occur after the user completed editing.
- Reduced unnecessary event activity while users were still entering data.
- Simplified field event scripts by delegating work to document-level functions.
- Improved application responsiveness.
- Helped synchronize related fields throughout the document.
- Encouraged cleaner separation of responsibilities between Acrobat events.

As my applications evolved, the Blur event became less about individual fields and more about coordinating the overall behavior of the application.

Lessons Learned

One of the most valuable lessons I learned was that not every operation belongs in the Validate or Calculate events. Many application tasks simply need to occur after the user finishes entering information.

The Blur event provided the ideal opportunity to trigger those operations without interrupting the user's workflow. Combined with document-level JavaScript functions, it became an effective mechanism for coordinating complex application behavior while keeping individual field events simple and easy to maintain.

Rather than viewing the Blur event as an afterthought, I eventually came to appreciate it as an important part of building responsive, well-organized Acrobat applications.

© Marty Karr

EasyPDF™ Acrobat Encyclopedia

A.1 Future Publication Notes

This reference was not written from theory.

Every technique, design pattern, and solution documented in this reference was developed, tested, and refined while creating the EasyPDF™ family of interactive PDF forms.

Many of the topics discussed here are not found in standard Acrobat JavaScript references because they were discovered through years of practical development, experimentation, debugging, and refinement while solving real-world problems.

The purpose of this reference is to document those lessons—not only to preserve them for future EasyPDF™ development, but also to help others avoid rediscovering the same solutions through trial and error.

© Marty Karr

EasyPDF™ Acrobat Encyclopedia

A.2 Using the HTML5 `download` Attribute for PDF Downloads

Subject: Missing HTML download Attribute Prevented Matomo Download Tracking

Problem:

Following a Windows 11 / Microsoft Edge update, PDF download links that previously behaved as expected began opening PDFs directly in the browser instead of downloading them. As a result, Matomo Analytics failed to record PDF downloads as download events.

Cause:
The HTML anchor (<a>) tags linking to PDF files did not include the HTML5 download attribute. Browser behavior apparently changed, causing Edge to display PDFs inline rather than treating them as downloadable files.

Solution:

Add download attribute to all PDF download links.

Before:

```
<a href="licensed forms/easypdf-medication-manager-licensed-copy.pdf"
  title="Download FREE Lifetime License (Limited-Time Offer)" >
  EasyPDF™ Medication Manager
</a>
```

After:

```
<a href="licensed forms/easypdf-medication-manager-licensed-copy.pdf" download
  title="Download FREE Lifetime License (Limited-Time Offer)" >
  EasyPDF™ Medication Manager
</a>
```

Result:

PDF files download correctly instead of opening inline in Microsoft Edge.

Matomo Analytics once again records PDF downloads correctly.

No changes to Matomo configuration or tracking code were required.

Lesson Learned:
Do not rely on browser default behavior for PDF links. Always include the HTML5 download attribute on EasyPDF™ Forms download links to ensure consistent browser behavior and reliable Matomo Analytics download tracking.

© Marty Karr

A.3 Creating a Professional PDF Title Page Illustration Cover

Subject

Creating a Professional PDF Title Page Illustration Cover

Problem

After creating a custom title page separately from the manuscript and inserting it into the completed PDF, Acrobat displayed the title page correctly at the selected magnification (Fit Width). H owever, advancing to the next page caused Acrobat to automatically change the zoom level, resulting in an inconsistent viewing experience throughout the remainder of the document.

Cause

The title page and manuscript pages were created using different applications and later combined into a single PDF. Although the pages appeared to be the same physical size, they retained different internal page definitions (such as page geometry and page boxes). Acrobat recalculated the page magnification when transitioning from the title page to the remaining pages.

Solution

Rather than creating the title page as a separate PDF and inserting it into the completed manuscript, insert the title page artwork directly into the word processor document (OpenOffice Writer in this case) as the first page. Export the entire manuscript as a single PDF so all pages are generated using the same PDF export engine.

Before

- Title page created separately.
- Separate PDFs combined into a single document.
- Acrobat changed magnification when advancing beyond the title page.
- Inconsistent viewing experience.

After

- Title page inserted directly into the manuscript.
- Entire document exported as one PDF.
- All pages share consistent page geometry.
- Acrobat maintains a consistent viewing magnification throughout the document.

© Marty Karr

Result

The zoom change between the title page and the remaining manuscript pages was completely eliminated. The document now opens and navigates consistently, providing a more professional reading experience.

Lesson Learned

Rather than creating the cover or title page separately and inserting it into an existing PDF, insert the artwork directly into the source manuscript before exporting the document to PDF. Generating the entire publication as a single PDF ensures consistent page geometry, prevents unexpected page magnification changes, and produces a more polished, professional publication.

A.4 Local Preview Delays Caused by Matomo Tracking

Symptom

After adding the Matomo tracking code to every page, local website previews from CoffeeCup HTML Editor (or opening pages directly from a local `file:///` path in Edge) m ay pause for several seconds before displaying the page.

The page eventually loads correctly, but the delay can make it appear that CoffeeCup HTML Editor or the browser is malfunctioning.

Cause

The standard Matomo tracking code executes immediately when the page loads and attempts to retrieve the tracking script from the Matomo server.

When previewing pages locally using the `file:///` protocol, the browser still attempts to initialize the tracking code even though the page is not being served from a web server. This unnecessary network activity can significantly delay local previews.

The issue does not affect the live website.

Solution

Execute the Matomo tracking code only when the page is served using HTTP or HTTPS.

Example

```
if (location.protocol !== "file:") {
    var _paq = window._paq = window._paq || [];
    _paq.push(['trackPageView']);
    _paq.push(['enableLinkTracking']);
    (function () {
        var u = "https://www.easypdfeforms.com/matomo/";
        _paq.push(['setTrackerUrl', u + 'matomo.php']);
        _paq.push(['setSiteId', '1']);
```


© Marty Karr

```
        var d = document,
            g = d.createElement('script'),
            s = d.getElementsByTagName('script')[0];
        g.async = true;
        g.src = u + 'matomo.js';
        s.parentNode.insertBefore(g, s);
    })();
}
```

Benefits

- Restores fast local previews.
- Prevents unnecessary network requests during development.
- Prevents local testing from appearing in Matomo analytics.
- Leaves live website tracking completely unchanged.

Lesson Learned

Third-party analytics scripts should generally be disabled during local `file:///` development unless they are specifically required for testing. Doing so improves development performance and keeps analytics data free of local testing activity.

© Marty Karr

A.4 Local Preview Delays Caused by Matomo Analytics Tracking Script

Symptom

After adding the Matomo tracking code to every page, local website previews from CoffeeCup HTML Editor (or opening pages directly from a local file:/// path in Edge) may pause for several seconds before displaying the page.
The page eventually loads correctly, but the delay can make it appear that CoffeeCup HTML Editor or the browser is malfunctioning.

Cause

The standard Matomo tracking code executes immediately when the page loads and attempts to retrieve the tracking script from the Matomo server.
When previewing pages locally using the file:/// protocol, the browser still attempts to initialize the tracking code even though the page is not being served from a web server. This unnecessary network activity can significantly delay local previews.
The issue does not affect the live website.

Solution

Execute the Matomo tracking code only when the page is served using HTTP or HTTPS.
Example

```
if (location.protocol !== "file:") {
var _paq = window._paq = window._paq || [];
_paq.push(['trackPageView']);
_paq.push(['enableLinkTracking']);
(function () {
var u = "https://www.easypdfeforms.com/matomo/";
_paq.push(['setTrackerUrl', u + 'matomo.php']);
_paq.push(['setSiteId', '1']);
var d = document,
g = d.createElement('script'),
s = d.getElementsByTagName('script')[0];
g.async = true;
g.src = u + 'matomo.js';
s.parentNode.insertBefore(g, s);
})();
}
```

Benefits

- Restores fast local previews.
- Prevents unnecessary network requests during development.
- Prevents local testing from appearing in Matomo analytics.
- Leaves live website tracking completely unchanged.

Lesson Learned

Third-party analytics scripts should generally be disabled during local file:/// development unless they are specifically required for testing. Doing so improves development performance and keeps analytics data free of local testing activity.

© Marty Karr

A.5 Why I Switched from Button Fields to the Document Link Tool for the TOC

Problem

For more than thirty years, I created interactive Acrobat PDF forms using transparent button fields whenever I needed users to click something. Naturally, when I began developing the EasyPDF™ Acrobat JavaScript Reference, I followed the same approach by placing transparent button fields over each Table of Contents (TOC) entry.

Although the button fields were configured with no border and no fill color, the underlying TOC text did not display correctly beneath the button overlays, making the technique unsuitable for a professional reference manual.

Cause

A button field is still a form field, regardless of whether its border or fill color is visible. Acrobat renders form fields as interactive objects that reside above the page content. As a result, transparent button fields may interfere with the appearance of the underlying document.

The Document Link Tool, however, was specifically designed for document navigation. Unlike button fields, document links are not form fields. They are simply invisible clickable regions that allow readers to navigate within a PDF without introducing unnecessary form objects.

Solution

Replace transparent button fields with Document Links for all clickable TOC entries.

Create an invisible document link over each TOC entry that leads directly to actual content and assign the link a Go to a page view action.

For organizational headings that merely group related topics (such as major chapter or section headings), no document link is required unless the heading itself begins a page containing instructional content.

Button Fields vs. Document Links

Button Fields

• Designed primarily for interactive PDF forms.

• Ideal for executing JavaScript, calculations, printing, submitting forms, resetting fields, and other interactive operations.

• Become part of the document's form field structure.

• May interfere with underlying page content when used solely for document navigation.

© Marty Karr

Document Links

- **Designed specifically for PDF document navigation.**
- **Ideal for Tables of Contents, indexes, cross-references, and "See Also" references.**
- **Do not create additional form fields.**
- **Provide simple, efficient navigation with minimal document overhead.**

Benefits

- **Eliminates unnecessary button fields from the document.**
- **Prevents button overlays from interfering with TOC text.**
- **Produces a cleaner, more maintainable PDF.**
- **Reduces unnecessary form objects and tab stops.**
- **Uses the Acrobat feature specifically intended for document navigation.**

Lesson Learned

One of the easiest mistakes long-time Acrobat form developers can make is using the same tools for every type of PDF.

Interactive forms and reference manuals have different navigation requirements. Button fields remain the preferred solution for form interaction and JavaScript execution, while the Document Link Tool is the better choice for navigating a Table of Contents or other document references.

Sometimes the best solution is not discovering a new Acrobat feature, but recognizing when an existing feature is better suited to the task at hand.

© Marty Karr

EasyPDF™ Acrobat Encyclopedia

A.6 – Investigating Acrobat's Unexpected Bookmark Creation

Problem

While maintaining the EasyPDF™ Acrobat JavaScript Reference, Acrobat unexpectedly created bookmarks that I had never intentionally added. Because the bookmarks appeared during normal document maintenance, it was initially unclear whether they were being generated by inserting new pages, modifying the Table of Contents (TOC), recreating Document Links, or some combination of these operations.

Before accepting the behavior as another Acrobat quirk, I wanted to determine whether routine TOC maintenance was responsible.

Investigation

To isolate the suspected cause, I modified the final TOC page by recreating the existing Document Links for Appendices A.1 through A.5. After confirming the links functioned correctly, I added a new TOC entry for Appendix A.6, assigned its page number, and created a new Document Link to the appendix.

By limiting the changes to the TOC page, the objective was to determine whether recreating existing Document Links or creating a new Document Link would cause Acrobat to generate additional bookmarks.

Results

Contrary to expectations, modifying the final Table of Contents page by recreating the existing Document Links for Appendices A.1 through A.5 and adding a new TOC entry and Document Link for Appendix A.6 did not generate any unexpected bookmarks.

To further investigate the behavior, the temporary one-page Appendix A.6 test page was later replaced with the completed two-page Appendix A.6. No changes were made to the Table of Contents or its Document Links.

Immediately after replacing the appendix pages, Acrobat automatically generated new bookmarks for the two inserted pages.

These results demonstrated that routine TOC maintenance—including recreating existing Document Links and creating new Document Links—was not responsible for the unexpected bookmark creation. The testing instead suggests that the behavior is more closely associated with inserting or replacing document pages than with maintaining the Table of Contents itself.

Although the exact trigger remains unidentified, the investigation successfully eliminated one suspected cause while narrowing the search to another.

© Marty Karr

Workaround

If Acrobat automatically generates unwanted bookmarks during page insertion or replacement, simply deleting the bookmarks may not permanently remove them from the document.

After deleting the unwanted bookmarks, use File → Save As Other → Reduced Size PDF to rewrite the document structure. This rebuilds the PDF and permanently removes the deleted bookmark objects.

Saving the document normally may not completely remove the internally stored bookmark information.

Benefits

- Eliminated routine TOC maintenance as a suspected cause of the unexpected bookmark creation.
- Confirmed that recreating existing TOC Document Links does not automatically generate bookmarks.
- Confirmed that creating a new TOC Document Link does not automatically generate bookmarks.
- Narrowed the investigation by eliminating one entire category of document editing operations.
- Reinforced the importance of isolating and testing one editing operation at a time when troubleshooting Acrobat.

Lesson Learned

One of the easiest mistakes during software troubleshooting is assuming that two operations occurring at nearly the same time are necessarily related.

Initially, the unexpected bookmark creation appeared to coincide with updating the Table of Contents. Controlled testing demonstrated that recreating multiple existing TOC Document Links and adding a new Document Link did not reproduce the behavior. However, replacing the temporary one-page Appendix A.6 with its completed two-page version immediately caused Acrobat to generate new bookmarks.

Although additional testing may eventually identify the precise trigger, this investigation demonstrated the importance of isolating one editing operation at a time. Eliminating suspected causes can be just as valuable as identifying the actual one, and careful, methodical testing often reveals software behavior that would otherwise be incorrectly attributed to unrelated operations.

© Marty Karr