

The Fusebox Guidebook

Jeff Peters



Proton Arts – Manassas VA USA

The Fusebox Guidebook

Copyright © 2005 by Proton Arts

FIRST EDITION: March 2005

ISBN: 0-9752647-3-7

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage and retrieval system, without written permission from the publisher, except for the inclusion of brief quotations in a review.

07 06 05 04 05 9 8 7 6 5 4 3

Interpretation of the printing code: The rightmost double-digit number is the year of the book's publication; the rightmost single-digit number is the number of the book's publication. For example, the printing code 04-1 shows that the first publication of the book occurred in 2004.

Published in the United States of America

Trademarks

All terms mentioned in this book that known to be trademarks or service marks have been appropriately capitalized. Proton Arts cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark. ColdFusion is a registered trademark of Macromedia, Inc. Fusebox is a registered trademark of Fusebox, Inc.

Warning and Disclaimer

This book is designed to provide information about ColdFusion programming. Every effort has been made to make this book as complete with respect to its topic and as accurate as possible, but no warranty of fitness is implied. The information is provided on an as-is basis. The author and Proton Arts shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book or from the use of the programs that may accompany it.

This book was written and set with OpenOffice.org 1.1.4

Dedication

For the bloggers

This book was inspired by a request from the blogosphere. Technical professionals are gaining an increasing amount of knowledge from the Net. This is an amazing and powerful change in our world, but it's nice to know there's still demand for the printed word.

— Jeff Peters

Other Proton Arts Books
by Jeff Peters

ColdFusion Lists, Arrays, & Structures
Fusebox 4 & FLiP:
Master-Class ColdFusion Applications
What's New in Fusebox 4.1

Table of Contents

| | |
|--|----|
| Introduction..... | 1 |
| What is This Book?..... | 1 |
| Who Should Read It?..... | 1 |
| Fusebox History..... | 3 |
| Fusebox Basics..... | 6 |
| Objectives..... | 6 |
| Standardization..... | 6 |
| Team Communication..... | 6 |
| Self-communication..... | 7 |
| Reduced Coding Time..... | 7 |
| Reduced Debugging Time..... | 7 |
| Reliability..... | 8 |
| Reusability..... | 8 |
| Focus..... | 8 |
| Coding Standards..... | 9 |
| Methodology..... | 10 |
| Coding Standards..... | 12 |
| Standard: Centralized Controller..... | 13 |
| What It Is..... | 13 |
| How It Works..... | 13 |
| Benefits..... | 16 |
| Standard: Unified Variable Scope..... | 18 |
| What It Is..... | 18 |
| How It Works..... | 18 |
| Benefits..... | 19 |
| Standard: Organized Circuits..... | 20 |
| What It Is..... | 20 |
| How It Works..... | 20 |
| Benefits..... | 20 |
| Standard: Application Roadmap..... | 22 |
| What It Is..... | 22 |
| How It Works..... | 22 |
| Benefits..... | 24 |
| Standard: Small, Focused Files (fuse types)..... | 26 |
| What It Is..... | 26 |
| How It Works..... | 26 |
| Benefits..... | 26 |
| Standard: Code the Documentation..... | 32 |
| What It Is..... | 32 |

| | |
|---------------------------------------|----|
| How It Works..... | 32 |
| Benefits..... | 33 |
| Standard: Encapsulation..... | 36 |
| What It Is..... | 36 |
| How It Works..... | 36 |
| Benefits..... | 37 |
| Methodology Standards..... | 39 |
| Standard: Wireframe..... | 40 |
| What It Is..... | 40 |
| How It Works..... | 40 |
| Benefits..... | 41 |
| Standard: Prototype/Frontend..... | 43 |
| What It Is..... | 43 |
| How It Works..... | 43 |
| Benefits..... | 44 |
| Standard: Client Communication..... | 46 |
| What It Is..... | 46 |
| How It Works..... | 46 |
| Benefits..... | 47 |
| Standard: Architecture..... | 49 |
| What It Is..... | 49 |
| How It Works..... | 49 |
| Benefits..... | 50 |
| Standard: Code the Documentation..... | 51 |
| What It Is..... | 51 |
| How It Works..... | 51 |
| Benefits..... | 54 |
| Standard: Unit Testing..... | 55 |
| What It Is..... | 55 |
| How It Works..... | 55 |
| Benefits..... | 55 |
| Summary..... | 57 |
| Appendix A – Coding Summary..... | 60 |
| Appendix B – Methodology Summary..... | 61 |
| Appendix C - Fusebox Versions..... | 62 |
| Appendix D – Fusebox Glossary..... | 64 |

Introduction

Fusebox has existed as a web development framework since 1998. Until now, there has never been a concise summary of Fusebox. Based on feedback from other Proton Arts publications, this guidebook is designed to correct that lack.

What is This Book?

This book is a guide to the Fusebox web development methodology and the Fusebox Lifecycle Process (FLiP). It is not a “how-to” book so much as a “what” book.

There are no code samples in this book, as it is designed to address Fusebox from a conceptual point of view. Fusebox applications are presently developed in ColdFusion, PHP, Lasso, and ASP (as far as I have discovered). The concepts discussed in this book are relevant to all those platforms.

As a language-agnostic guide, it is appropriate for anyone who is interested in Fusebox and what it offers to the program manager, the developer, and even the curiosity seeker.

Who Should Read It?

Many developers have approached me with the request for suggestions to convince their managers

that Fusebox is the solution they've been seeking. While this book is intended to be informative from a managerial point of view, it is not an evangelical volume. I firmly believe that Fusebox is convincing enough in itself to not need any proselytizing. However, it's hard to evaluate something if you don't know what it is. From that perspective, this book is intended to help managers make informed decisions about Fusebox and FLiP. As a manager, the 70% failure rate in software development projects¹ is of major concern.

Developers may also like the information contained here, in that it is an overview of what Fusebox is about. Some developers like to understand a concept by taking apart the code to see how it all works; others are more comfortable with getting a birds-eye view before they start working with the details. It is this latter group that will most benefit from reading this book.

Finally, the book is aimed at curiosity seekers in general. I've often heard opinions of Fusebox offered by those who have never used it, and these opinions sometimes indicate a clear misunderstanding of what Fusebox is and why it should (or shouldn't) be used. If you're curious about the reality of Fusebox, this is the place to start.

1 Standish Group, 2003 and prior.

Fusebox History

A group of ColdFusion developers including (among others) Gabe Roffman, Joshua Cyr, Michael Dinowitz, Robi Sen, and Steve Nelson started discussing common solutions to the problem of reinventing the wheel with each new site development. The CF-Talk mailing list at HouseOfFusion.com became home to the discussion. Threads on a better way to organize applications progressed, and ideas began to form within the group. Example frameworks were discussed, but a catalyst was needed to finalize the system.

That catalyst arrived when Steve and Gabe (both independent consultants at the time) started sharing an office in Charlottesville, Virginia in the spring of 1998. They began to swap notes and ideas about their own development practices and how they related to some of the CF-Talk ideas. Discussions of the state problem and design patterns led to ideas about organizing applications, controlling system actions, and separating functionality of pages.

Before long, Steve finalized his idea for a centralized controller page within each directory of a site. Gabe noticed the resemblance of this model to the breaker panel of a house. This "fuse box" ColdFusion application system was documented in a white paper as the first specification for Fusebox version 1.0. Joshua Cyr wrote an example calendar application based on that specification.

Impressed with the system, Robi Sen hired Gabe and Steve to create the first full-scale Fusebox implementation at eBags.com. This e-commerce site was a magnificent success and is still going strong today as the largest online retailer of luggage.

Others got involved in the Fusebox concept, including Hal Helms as one of its earliest proponents. His columns in *ColdFusion Developer's Journal* spread the word, and interest began to accelerate. The first Fusebox conference was held in Charlottesville, Virginia, in 2000, attracting about 100 dedicated developers. By this time, Hal had created the Fusedoc specification and was expanding on Fusebox's basics with some ideas he collectively called Extended Fusebox (XFB).

As Fusebox grew and the input of its supporters expanded, ideas for a new version were swirling about, and the community grew restless for progress. The Fusebox Council (Hal Helms, Steve Nelson, Nat Papovich, Jeff Peters, John Quarto-vonTivadar, and Erik Voldengen) was formed to help codify a new version. That version was Fusebox 3, released in the fall of 2001.

Fusebox 4 was developed primarily by John Quarto-vonTivadar and Hal Helms. It provided a refinement of the organizational aspects of Fusebox, and moved us into the realm of XML-based application configuration files. Version 4.1 adds some new verbs to the Fusebox vocabulary, and is covered in the Proton Arts publication *What's New in Fusebox 4.1*. Today the Fusebox standard,

housed at www.fusebox.org, is going strong under the support of Team Fusebox.

Since its inception, Fusebox has been based around an open exchange of ideas, a community. This community has been pivotal to Fusebox's success and growth. The Fusebox core files have been translated into several other web development languages, including PHP (David Huyck, Mike Stark et al) and Lasso (Michael McKellip and Rich Tretola). Make no mistake, we have come a long way since that first basic concept, but the fundamental ideas of Fusebox remain intact.

Fusebox Basics

Fusebox has always involved leveraging the best practices of developers to the benefit of everyone using it. There are some basic standards that have been kept in mind by the developers of Fusebox over the years. These standards include general objectives for Fusebox, coding standards, and a standard methodology.

Objectives

The general objectives for Fusebox are all related in the desire to create solid, well-designed applications quickly. The only way to achieve this goal is to use a proven set of techniques. These techniques, detailed later in this book, are based on the following objectives.

Standardization

Fusebox is, at its heart, a standard. While there have been several implementations of the Fusebox core files over the years, it is the cooperative standard of Fusebox that represents its greatest power. Fusebox seeks to empower developers through standardized practices.

Team Communication

Several of the aspects of Fusebox, from its file naming conventions to its use of the Fusedoc standard for documentation, are aimed at improving

communication between developers. Fusebox seeks to empower developers through improved team communication.

Self-communication

Many web developers operate as one-man development shops. In this environment, it is critical to be able to understand what you were thinking at the time a piece of an application was developed. Just as aspect of Fusebox seek to improve team communication, these same aspects can improve a developer's understanding of their own past mindset. Fusebox seeks to empower developers through improved self-communication.

Reduced Coding Time

Rather than starting out a development project by coding, Fusebox developers start with several steps of planning and architecture. Performing these steps serves to define the project's code needs so well that the actual coding time required is reduced greatly. Fusebox seeks to empower developers through reduced coding time.

Reduced Debugging Time

Fusebox includes the concept of unit testing small chunks of code. Because unit testing identifies the syntax errors and other common bugs at a point where they are easy to identify and correct, less overall time is spent on tracking down and

eradicating bugs in the integrated system. Fusebox seeks to empower developers through reduced debugging time.

Reliability

Because of its well-designed nature, a Fusebox application tends toward reliability in comparison to the average web application. Following the rigor of a methodology improves the quality of an application. Fusebox seeks to empower developers through increased reliability.

Reusability

Any given problem has a limited number of solutions available. It is often discovered that portions of solutions have similar behaviors. By reducing code modules to small, well-defined chunks, Fusebox increases the ability to reuse code, both in the context of the same application, and in the context of moving the same (or significantly similar) code into another application. Fusebox seeks to empower developers through increased reusability.

Focus

I save the best for last. One of the greatest objectives of Fusebox is to allow developers to focus on the skills necessary to handle the task in front of them. This means that database tasks can be handed to a DBA, interface tasks can be handed

to a coder skilled in interface issues, etc. It also means a one-man shop can focus on one type of task at a time, for example writing all the queries at once, then writing all the interface modules. Fusebox seeks to empower developers through focus of skills.

These objectives for Fusebox are achieved through the implementation of coding standards and a methodology known as the Fusebox Lifecycle Process (FLiP). We'll take a brief overview of these aspects of Fusebox before looking at detailed discussions of them in the coming chapters.

Coding Standards

Versions of Fusebox from 3.0 forward have included the concept of a set of “core files” that handle basic Fusebox processes. However, even before there were core files, there were coding standards. From the beginning, Fusebox has been a set of standard practices for coding that made application code easier to understand.

This means Fusebox coders can easily look at each other's applications and understand what's going on inside. It also means that Fusebox applications can be integrated with one another, in whole or part. Developers can use pieces of applications developed by other Fuseboxers in their own applications (assuming the appropriate license permissions exist).

Finally, it means that Fusebox applications are very

flexible from the developer's perspective, and Fusebox developers are very flexible from the manager's perspective. Using Fusebox means that developers who are new to a project can get up and running with a minimum of fuss and learning curve. We'll look at the Fusebox coding standards in detail later in the book.

In addition to the benefits of standardized coding, Fusebox includes a standard application development methodology.

Methodology

The FLiP methodology is aimed at alleviating some of the most prevalent problems in application development. It works by taking the typical perspective of application developers (a very technically-oriented programmer perspective) and standing it on its head. Instead of thinking about technical considerations first, Fuseboxers think about the customer's needs first. The technical details result from the customer's needs, instead of driving the customer to use a solution the developer thinks will fit.

This means that Fuseboxers enjoy some very real benefits. First, by not falling victim to the tendency to continually use technical methods they've used before, they can be free to explore appropriate technologies for each project. Second, they can avoid the statement that every application developer has heard, and none likes: "That's nice, but it's not

what I wanted.” In fact, Fuseboxers enjoy some of the least-surprising application deployments of any category of developers.

Finally, the FLiP methodology emphasizes thorough documentation and testing. These two aspects work hand-in-hand to ensure the application works according to its design, and doesn't “blow up” in unexpected ways once completed.

We've had a chance to look at the objectives of Fusebox and an overview of what it's about. The next two chapters will look at the coding standards and methodological standards in a bit more detail.

Coding Standards

Fusebox coding standards provide a foundation for developers to be able to compare applications, discuss techniques and coding styles, and create modules that can be easily reused in other applications.

Each of the coding standards in this chapter has been implemented throughout the history of Fusebox. In early versions, the developer was responsible for seeing to all the conventions. In later versions, the core files take some of the burden from the developer. In all cases, the purpose behind the convention remains consistent regardless of which version of Fusebox is being discussed.

For each standard, we'll look at what the convention is, how it is implemented in Fusebox, and the benefits it provides.

Standard: Centralized Controller

The single largest characteristic of a Fusebox application is the use of a centralized controller to manage application flow. This is where Fusebox gets its name: the “fusebox” controls the flow of “power” to the “circuits”.

What It Is

In Fusebox, the centralized controller determines which modules of code, known as *fuses*, are used for a particular page request, known as a *fuseaction*. The value of the *fuseaction* determines which fuses will be used to build the page to be delivered to the browser.

How It Works

Fusebox began with the idea of using a switch construct to assemble fuses based on the value of the *fuseaction*. For example, a *fuseaction* might have a value of `cart.showContents`. The switch construct would evaluate the part after the dot (`showContents`), and include the appropriate fuses. We'll look at the purpose behind the first half of the *fuseaction* later.

Perhaps the `showContents` *fuseaction* needs only one fuse to accomplish its purpose. In this case, the switch construct would be defined to include that one fuse file.

A different *fuseaction*, such as `cart.deleteItem`,

might require more than one fuse. In this case, the switch construct would be defined to include each of the required fuse files.

Compared to other typical methods of writing ColdFusion templates, the centralized controller approach trades large cumbersome templates for smaller, well-focused ones.

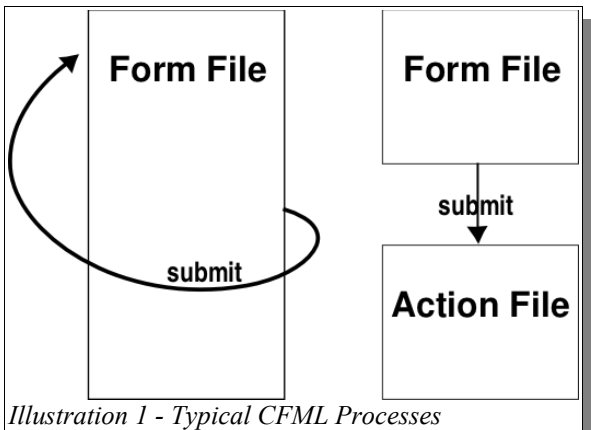
For example, a typical approach is to create a template with a form, and the actions to be performed in response to the form, all in a single template with conditional logic to determine whether the template should display the form or respond to its submission, and finally to request another template when the form processing is complete.

Another typical approach is to use separate files for forms and actions, without any controller mechanism. This means the form template requests the action template, which in turn requests some other template, and so on.

Using the Fusebox approach, the same processing would be performed with several small files: first, the controller (fusebox) file, with a switch construct for two actions, like `showMemberForm` and `addNewMember`. Next, there would be a fuse file for the form, named something like `dsp_MemberForm.cfm`, and another fuse for the action to add a new member, named something like `act_AddMember.cfm`. The form file would only contain the code to display the form, and the action

file would only contain the code to process the output from the form. There might also be additional files, such as a query file used to provide data for a droplist on the form. The relationships between these files and how the program flow is controlled among them is all handled by the controller.

Comparing the two approaches visually looks something like Illustration 1 and Illustration 2.



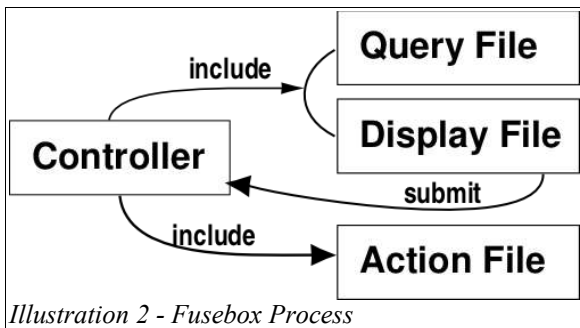


Illustration 2 - Fusebox Process

The difference between the two processes is this: with the typical processes shown in Illustration 2, a **developer never knows** which method was employed without examining the actual code. An application might use the all-in-the-form-file approach, or it might use the form-and-action approach, or it might (more commonly) use a mix of the two. This makes typical applications difficult to interpret and maintain.

The Fusebox process, on the other hand, doesn't vary. Every page request in a Fusebox application goes through the controller. The controller then retrieves the other template(s) required to perform the requested action. Any given action may need one or several templates ("fuses") to get the job done. Once the job is done, program flow returns to the controller for the next request.

Benefits

The benefits of having a centralized controller are

several. First, a Fusebox application's program flow is very easy to understand without looking at the actual fuse files. The program flow is all contained in the controller file. In fact, the controller file in Fusebox 4.0 and later is written in XML, and is completely agnostic of the language used to write the fuse files.

Bringing new developers into a Fusebox environment is greatly simplified, as the learning curve for the application does not depend on reading and interpreting all of the application's code. A developer can quickly understand “the big picture” without seeing a single line of actual program code: the controller tells the whole story.

Using a centralized controller also means that fuse files end up being smaller, more focused pieces of code. Very little (if any) of the program flow logic is contained in fuse files—that's the controller's job. This improves the separation of presentation and business code, which improves the robust nature of the application, and its overall maintainability.

Standard: Unified Variable Scope

Web developers often confront the issue of the origin of data to be used in a template. Data submitted from a form can be found in the Form scope, while data submitted through a URL request can be found in the URL scope. Finally, data submitted by requesting a template as a custom tag (using the CFMODULE command in ColdFusion, for example), can be found in the Attributes scope.

This means that a developer writing a template to respond to a set of data needs to know where the data is coming from in order to write proper code. If the data is coming from a form, and the developer specifies URL scope, an error will occur.

It also means that if the source of the data changes, the code for template that responds to the data must be changed. This makes the responding code fragile: it can be rendered useless by a change in another part of the system.

What It Is

To overcome these problems, Fusebox uses a unified variable scope for data that is passed into a fuse. That scope is the Attributes scope. All data passed into a fuse can be found in the Attributes scope, regardless of its origin.

How It Works

The Fusebox core files copy any data that exists in

the Form or URL scopes and copies it into the Attributes scope. No preparation or action is required of the developer.

Benefits

Having all the variables placed in a unified scope simplifies programming for the developer, and also increases the flexibility of the resulting fuse.

Developers know that incoming data can be found in the Attributes scope, so they don't need to look around for the proper scope to use.

Additionally, if the application's process changes, and the source for the inbound data changes from a form to a URL submission, the responding fuse does not need to be recoded. The application will also be able to simultaneously respond to custom-tag-style requests with the same code. The application is much more robust in this form.

The Fusebox core does not dispose of the variables found in the Form and URL scopes, either, so they may be used in those scopes at the architect's discretion. For example, a fuse's job may be to provide some validation logic for data submitted by a user. The architect doesn't want to allow automated submission via the URL, so the fuse is written to use Form-scoped variables only. The unified variables scope is a tool that provides flexibility in most cases, but is not a constraint that must be followed if the application's design is better off using a specific variable scope.

Standard: Organized Circuits

Web applications that do not follow a development methodology tend to be jumbled messes of code that are very difficult to follow and interpret from the inside. Fusebox alleviates this problem by using organized circuits.

What It Is

Fusebox applications are comprised of circuits. Each circuit is a collection of code that is functionally related in some way.

How It Works

During the architecture phase of creating a Fusebox application, all the actions that can be performed by the application are identified. These actions are then sorted into small collections of related actions. The collections become the circuits of the application. Each will be contained in its own directory within the finished application.

Benefits

Organization is one of the hallmarks of well-crafted applications, and it has been one of the hallmarks of Fusebox from the very beginning. Gathering common functionality together into a circuit makes that portion of the application very easy to create, understand, and maintain. When the entire application is organized in this manner, the benefits extend throughout.

Organized circuits provide focus to the architect when the application is being designed. It is highly effective to be able to “zoom in” on the requirements of a small segment of the application when creating the design.

Similarly, organized circuits provide a great deal of leverage for development teams. Some teams develop specialists in particular types of circuits (shopping carts, for example) who are very skilled at customizing that type to a specific need.

Circuits help define the boundaries for developers. When working on creating or maintaining code, a developer can easily see and understand the code that might be related to the task (i.e., is within the circuit) and the code which is “out of bounds” (not within the circuit).

Teams that have not yet migrated to formalized version control systems will find circuits to be a helpful means of distributing code among team members without risking collisions.

Circuits also help provide definable milestones during the development process. Daily builds can be communicated in terms of circuits completed (and even fuseactions within circuits, and fuses within fuseactions). This hierarchical nature is inherent to Fusebox.

Standard: Application Roadmap

When dealing with most applications, a developer has no choice but to read through the actual code to understand how the program flow works. Fusebox creates an advantage by providing a roadmap of the application.

What It Is

In versions of Fusebox prior to 3, the roadmap functionality is contained in a switch construct within the index file(s).

In Fusebox 3, the roadmap is contained in the `fbx_switch` file(s), and is still a switch construct in the native application language. This file is often referred to as the “fusebox file”, as it performs the central controller function.

In Fusebox 4, the roadmap is contained in the `circuit.xml` file(s), and is written in the Fusebox XML vocabulary for circuit configuration files. While this file doesn't actually perform the central controller function in FB4 (that's handled by the core files), it does define how the controller behaves.

How It Works

Regardless of version, the roadmap functionality in Fusebox is similar. The switch construct, or circuit definition in FB4, defines which fuses will be used to perform a particular fuseaction, and the order in

which they will be used.

In Fusebox 3 and prior, a fuseaction would look something like this (ColdFusion example provided):

```
<cfcase value="showForm">
  <cfset xfa.submitButton =
"cart.totalUp">
  <cfinclude
template="qryStateCodes.cfm">
  <cfinclude template="dspMyForm.cfm">
</cfcase>
```

This block says, “to perform the fuseaction named **showForm**, use the templates **qryStateCodes.cfm** and **dspMyForm.cfm**, in that order.”

We can also tell from the first line inside the case that there is a submit button on the form, and when the submit button is clicked, the application will execute the fuseaction called **cart.totalUp**.

In Fusebox 4 and later, a fuseaction would look something like this:

```
<fuseaction name="showForm">
  <xfa name="submitButton"
value="cart.totalUp" />
  <include template="qryStateCodes" />
  <include template="dspMyForm" />
</fuseaction>
```

Note that this second example is not ColdFusion. Nor is it PHP, Lasso, or any other web development language. This example is valid regardless of the language used to implement the application, because it is written in the Fusebox XML vocabulary for

circuit configuration files.

Benefits

The roadmap feature of Fusebox applications offers benefits at several points in the application's lifecycle.

First, since the circuits and fuseactions are defined before the code is written, the roadmap allows walk-through testing of the application's functional logic even before programming starts. This provides the opportunity to catch overlooked items before they become more expensive to fix.

Second, the roadmap provides a one-stop place for a developer unfamiliar with the application to understand its overall flow before looking at any code. The use of Exit Fuseactions (XFAs; the lines with **xfa** in them in the examples) within the fuseaction definitions allows a reader to follow program flow. Proper naming of XFA variables indicates what the user will do to cause the XFA to be fired (e.g., **xfa.submitButton** in the examples).

The roadmap also allows a developer to pick out the small collection of fuses that are required for the fuseaction. This prevents such chores as searching through entire directories of code, trying to find the appropriate files to examine.

The roadmap can also act as a checklist during the coding process. As each fuse file gets coded and unit tested, it can be marked off in a printout of the roadmap. When all the fuses in a fuseaction are

checked off, the fuseaction is marked as complete. When all the fuseactions in a circuit are checked off, the circuit is marked as complete. This provides a very detailed project management checklist during the coding phase.

Finally, since the roadmap files are written in XML in Fusebox 4 and later, XSLT and other XML techniques can be used to automatically generate project checklists, etc., directly from the roadmap files.

Standard: Small, Focused Files (fuse types)

Many software shops use naming conventions to assist with identifying files. Fusebox goes a bit further by defining the types of fuses that will make up an application.

What It Is

This standard simply specifies that each fuse file will be tasked and named according to a simple set of rules. These rules guide a Fusebox architect in designing the application, and cause fuses to be small and focused in their scope.

How It Works

Fusebox defines three basic fuse types: display fuses, query fuses, and action fuses. Display fuses are defined as code that presents something to the user. Query fuses are defined as code that interacts with the database. Action fuses are defined as code that performs processes other than providing information to the user. These fuses are identified by the prefixes `dsp_`, `qry_`, and `act_`, respectively. (The underscores are optional; some Fuseboxers use them and others don't.)

Benefits

The benefits to defined fuse types are several. First, by defining what type of code can belong in a fuse,

the amount of code in any given fuse is limited. Earlier we looked at an illustration of a typical ColdFusion template that contained code to display a form and process it. That same template in Fusebox would typically be created through the use of three fuses: a query fuse to populate the dropdown on the form, a display fuse for the form itself, and an action fuse to handle the form submission.

Let's look at some code for a slightly more complex form, and how it would compare with the same code using the Fusebox approach.

```
<!-- assignSupervisor.cfm -->
<cfif not IsDefined("form.submit")>
  <cfquery name="Employees" dsn="myDB">
    select empID, name from employees
  </cfquery>
  <cfquery name="Supervisors" dsn="myDB">
    select empID, name
    from employees
    where deptHead = true
  </cfquery>
  <form name="myForm"
    action="assignSupervisor.cfm">
    Employee:
    <cfselect name="empID"
      query="Employees"
      display="name"
      value="empID"><br>
    Supervisor:
    <cfselect name="supID"
      query="Supervisors"
      display="name"
      value="empID"><br>
    <input type="submit">
  </form>
```

```

<cfelse>
  <cfquery name="assignSupervisor"
           dsn="myDSN">
    update Employees
      set SupervisorID = #form.supID#
      where empID = #form.empID#
  </cfquery>
  <cfoutput>Employee #form.empID#
supervisor set to
#form.supID#.</cfoutput>
</cfif>

```

Now here's the Fusebox approach. First, we have two query fuses--one for the employees:

```

<!--- qryGetEmployees --->
  <cfquery name="qryGetEmployees"
           dsn="#request.DSN#">
    select empID,
           name
    from employees
  </cfquery>

```

And one for the supervisors:

```

<!--- qryGetSupervisors --->
  <cfquery name="qryGetSupervisors"
           dsn="#request.DSN#">
    select empID,
           name
    from employees
    where deptHead = true
  </cfquery>

```

Then, the display fuse:

```

<!--- dspAssignSupervisorForm.cfm --->
  <form name="myForm"
        action="#self#">

```

```
<input type="hidden"
      name="fuseaction"
      value="#xfa.submitButton#">
Employee:
<cfselect name="empID"
          query="qryGetEmployees"
          display="name"
          value="empID"><br>
Supervisor:
<cfselect name="supID"
          query="qryGetSupervisors"
          display="name"
          value="empID"><br>
<input type="submit">
</form>
```

And finally the action fuse:

```
<!-- actAssignSupervisor.cfm -->
<cfquery name="assignSupervisor"
         dsn="myDSN">
    update Employees
       set SupervisorID = #form.supID#
       where empID = #form.empID#
</cfquery>
<cflocation
template="#self#?fuseaction=#xfa.success#"
">
```

Note is how much smaller the fuse files are. The code to accomplish this set of tasks has been broken up into individual tasks: retrieve data, display form, and update database. This makes the code in each fuse far more focused and easy to manage.

The second thing to notice is that there is no flow control logic in the fuse files as there is in the first

example. That's because the flow control is handled, appropriately, by the Fusebox application's central controller, not by fuse files. Fusebox intrinsically separates not only database, display, and business logic, but also program flow. A Fusebox developer working on a fuse never needs to worry about program flow.

Third, the organization of program code into fuses makes identifying files in a repository much simpler. Consider the following file lists:

assignSupervisors.cfm
departments1.cfm
departments2.cfm
employees.cfm
raises.cfm

actUpdateSupervisor.cfm
actAddEmployee.cfm
dspAssignSprvvrForm.cfm
dspEmployeesReport.cfm
qryGetEmployees.cfm
qryGetSupervisors.cfm

The lists on the left is practically indecipherable. Is “assignSupervisors.cfm” a form, or does it perform updates, or both? What makes “departments1.cfm” different from “departments2.cfm”? What do “employees.cfm” and “raises.cfm” do?

The list on the right, on the other hand, is much easier to read. There are two action fuses: one to update a supervisor and one to add an employee. There are two display fuses: one to display the Assign Supervisor form and one to display the Employees Report. There are two query fuses: one to get employees and one to get supervisors.

Finally, there is the matter of local standards vs. a published one. Many development shops use some form of naming convention for their files, but these

local standards offer no benefit outside the local shop. The fact that Fusebox is a widely adopted standard means that new hires are already familiar with the “local” convention when they arrive on the job. Fusebox can save innumerable hours in the training of new employees.

Standard: Code the Documentation

Every developer has been instructed to document the code at some point in their career. The problem is, very few have been instructed on *how* to document the code. Fusebox uses the Fusedoc standard for documenting fuses.

What It Is

Fusedoc is a documentation standard conceived and designed by Hal Helms. In its 1.0 version, Fusedoc was defined by an Extended Backus-Naur Form (EBNF) notation. In its 2.0 version, Fusedoc is an XML vocabulary.

How It Works

Every fuse in a Fusebox application has a Fusedoc at the top of the file. The Fusedoc is actually written before the code. The coder writes the fuse's code according to the specifications in the Fusedoc. As Steve Nelson once put it, in Fusebox we don't document our code, we code our documentation.

A Fusedoc includes three primary sections: first, a Responsibilities section, in which the fuse's task is described in plain English; second, a Properties section, in which the history of the fuse and any notes and miscellaneous properties of the fuse are documented; and third, an IO section, in which the data the fuse needs before it can run (input) and the

data the fuse needs to create (output) are defined.

Benefits

Fusedoc provides a foundation for both the architect and the developers of a Fusebox application.

The architect is the creator of the Fusedocs in an application. The discipline of writing Fusedocs forces the architect to examine the interaction among fuses, and refine the application's design to make sure all requirements are met.

Writing Fusedocs is also a talent that develops with experience. An architect becomes more skilled at communicating the design requirements through Fusedocs as coded fuses come back from the coders. For a typical architect's first Fusebox application, the fuses wind up missing some important pieces when they come back from coding. This is due to information not communicated to the coder through the Fusedoc. The architect learns to refine Fusedocs to include all the information a coder needs to create the fuse, and nothing more.

For the coder's part, Fusedocs are a great boon. Fusebox coders are completely freed from the need to know anything about the overarching application, the client's needs and/or mood, or anything else outside the scope of the individual fuse. This makes it possible to simply pick up the fuse file with its Fusedoc (called a "fuse stub") and write the code. Excellent coders who like to put on the headphones, crank up the music, and code are empowered to do just that. It's an incredibly productive environment.

Additionally, the use of Fusedocs means that a Fusebox application actually overcomes the longstanding problem described in Frederick Brooks' *The Mythical Man-Month*. Brooks describes the problem by stating that adding programmers to a software development project doesn't allow the project to be completed more quickly, because it takes time to train and familiarize the added developers, which costs time taken from the original development team, thus offsetting the benefit of the added labor.

In Fusebox, though, by the time the project gets to the coding stage, it IS possible to reduce the coding time by bringing additional developers in, because they don't need to know anything other than Fusedoc and the development language being used. It is theoretically possible, assuming we can farm out work to as many developers as we have fuses, to code a Fusebox application of any size in one day or less. Steve Nelson has even implemented an online Fusebox piecework system where coders can check

out fuses, code them, and get paid on a piecework basis.

Additionally, Fusedocs make maintenance of an application much easier. A coder can very easily understand the design purpose of a fuse as well as what the code actually does. Design intent is a critical aspect of maintenance that is missing from most applications' internal documentation, causing maintenance developers to expend time and effort to research the problem outside the code base.

Standard: Encapsulation

Encapsulation is a concept in software engineering that refers to the degree to which the inner workings of a module are hidden from other parts of the system. If the rest of the system can interact with the module through a simple set of controls, then the module is said to be encapsulated. Fusebox encourages the encapsulation of fuses through the use of Exit Fuseactions (XFAs).

What It Is

An XFA is simply a variable that holds a fuseaction as its value. Fuses that employ fuseactions as part of their job don't use explicit fuseactions; rather, they reference XFAs. The values for XFAs are set in the circuit configuration file, removing flow control from the fuse code.

How It Works

Each fuseaction defined in the circuit configuration file (circuit.xml) may contain `<xfa>` elements. These elements establish XFA variables for the fuseaction. For instance, a fuseaction with this code:

```
<fuseaction name="showLoginForm">
  <xfa name="loginButton"
    value="login.validateUser" />
  <include template="dspLogin.cfm" />
</fuseaction>
```

defines a fuseaction to show a login form. An XFA

is defined with the name **loginButton**. This indicates that the included display fuse, **dspLogin.cfm**, has an exit point at a button with the label **login**. When this button is clicked, the application will execute the fuseaction **login.validateUser**, as indicated by the value of the XFA. If at some point in the application's maintenance lifecycle the decision is made to use a different fuseaction to validate a user, we don't need to change anything in the **dspLogin.cfm** fuse; we just change the value of the XFA to the new fuseaction.

Benefits

Exit fuseactions benefit the application's architect, the coders, the maintainers, and the project manager.

From the architect's perspective, XFAs provide a way to disconnect the coding requirements from awareness of the overall system. Instead of needing to provide sufficient information to coders for them to understand the system (or at least a complete circuit) before they can write code, the architect simply passes along the containers for that information in the form of XFAs. This a big savings in terms of time taken up by team awareness activities.

From the coder's perspective, XFAs enable focus on the real job of a fuse, rather than externalities. This makes the coder's job easier by removing concerns

about unnecessary aspects of the system.

From the maintainer's perspective, XFAs allow the complete application flow story to be told in the configuration files, instead of being wound throughout the code in fuses. The maintainer only has to look through the configuration files (which are named identically in every Fusebox application) to find out how the application's flow works. This greatly reduces the amount of time required to isolate a problem and deal with it. It also means a new feature can be completely coded and tested, then placed in the application and "turned on" by simply referencing the new feature's fuseaction in the appropriate XFA.

From the project manager's perspective, XFAs are one of the key factors in Fusebox that make it possible to bring in new programmers without incurring enormous costs in training time and "getting up to speed". Coders who are familiar with the language being used can be assigned fuses that are completely agnostic with respect to the larger system.

Methodology Standards

Fusebox began as a set of best practices, and evolved into an application framework. Many of the best practices that have been used with Fusebox have been organized into Fusebox's companion methodology, the Fusebox Lifecycle Process (FLiP).

Just as there are standards for coding associated with the Fusebox framework, there are also standards for development associated with FLiP. These standards include concepts such as wireframing, creating a prototype/front-end, communicating with the client, designing the application's architecture, coding practices, and unit testing.

Though the Fusebox framework does not require developers to use the FLiP process, there are many benefits to be realized in so doing. Also, there is no requirement to use the Fusebox framework when following the FLiP process, though certain steps will require modification if Fusebox is not the target framework.

Standard: Wireframe

Unlike many traditional lifecycles, which start with defining a data model such as a relational database, the FLiP process focuses on the business model and the customer's concept of the user experience. The wireframe is the first step in codifying the application's behavior based on the customer's requirements.

What It Is

In Fusebox parlance, a wireframe is a simple, non-graphical, clickable map of the application's actions from the user's perspective. The architect explores, along with the customer, everything the application is expected to do. The visual appearance of the application is not under consideration at this point.

How It Works

A Fusebox wireframe is defined by a simple text file that is interpreted by a wireframe editor tool. The wireframe editor enables both the interactive display of the wireframe and editing of the wireframe file. This allows very rapid editing of the wireframe, often with the architect working side-by-side with the customer.

Illustration 3 shows the map page of a wireframe, and Illustration 4 shows the first page of the same wireframe in interactive mode.

WireFrame Viewer/Editor 3.80

"FlagPoll"

Map Edit History Generate Code Admin Help Credits

Quick Map of "FlagPoll"

| Page | Exits and Destinations |
|---|---|
| Welcome I welcome the user and display the current poll question. | Agree --> Poll Results Disagree --> Poll Results Current Results --> Poll Results |
| Poll Results I display the results of a poll. | Home --> Welcome |
| Admin I let the administrator enter a new poll question, and display a list of polls taken. | Add Question --> Question Update Notice Question Link --> Poll Results |
| Question Update Notice I display a message confirming the addition of a new question. | |

Illustration 3 - Wireframe Map

WireFrame Viewer/Editor 3.80

"FlagPoll"

Map Edit History Generate Code Admin Help Credits

Welcome

- I welcome the user and display the current poll question.

Exits

- Agree
- Disagree
- Current Results

Illustration 4 - Wireframe Page

Benefits

Wireframing is a very rapid method for launching the application design process. By essentially ignoring database design until the user requirements

are extremely well-understood, we avoid getting bogged down in technical trappings before they are essential. The phases of FLiP are designed to progress from least to most expensive in terms of the cost to make a change to the application. Thus, during the wireframing phase, making a change is a matter of a few seconds to change the wireframe file and check the change with the customer. Conversely, once the application's code has been written and tested, introducing a change is a much more expensive proposition.

The wireframe is the beginning of the design process in FLiP, which is an ongoing set of conversations with the customer, each taking a slightly different form. This initial conversation is quick, agile, and devoid of artistic considerations. The idea is to capture the customer's concept of what the application is supposed to do. Because we are avoiding everything other than functionality at this point, the process is very quick and precise.

Standard: Prototype/Frontend

Most software development lifecycles jump into implementation details too quickly. Whether it's succumbing to the urge to start writing code, or basing the entire application's design on a relational database model, technical people tend to frame the solution to a problem in terms of the technology with which they are familiar. Fusebox turns this trend on its head, refusing to look at technical issues until the customer's needs are fully explored.

What It Is

In FLiP terms, a prototype is a clickable model of the application that looks and feels like the real thing, but has no application code or database behind it. Due to the loaded nature of the term “prototype” in a variety of industries and the fact that much of the prototype winds up as operational HTML in the production system, we have begun to refer to this phase as the front-end.

How It Works

This step of FLiP is the one where the designers are most involved. Through an iterative communication process with the customer, the designer creates a representation of how the application will look and behave.

The most-asked question during the front-end phase is, “Do you mean, like this?” This is the question the designer asks the customer with every change

made to the front-end. For example, the customer asks, “Can we make the menu text larger, and use a plainer font?” The designer makes changes that he believes conform to the customer's wishes, then asks, “Do you mean, like this?” Perhaps the customer likes the changes; perhaps the customer wants something else. Fuseboxers recognize that most customers don't know what they want until they see it. Thus, we provide a way for them to see it before we actually build it.

Benefits

Like wireframing, creating the front-end is a communication tool between the developer/designer and the customer. Whether or not the customer can clearly express it, all customers have expectation with respect to how their application will look and behave.

The wireframe step has defined most of the functional requirements of the application; the front-end phase defines the appearance and finalizes the functional aspects. There will be points that come to light during the front-end phase that were not discovered during wireframing. This is perfectly normal. We certainly prefer to discover these points before the application is coded, and changes become more expensive.

Using a front-end as a requirements analysis tool is also a big time-saver. Instead of writing a voluminous requirements document that may or may not prove useful to a development team, we create

the actual front-end of the application. The HTML code that is created during this phase (or Flash movies, or whatever we're using for the front-end) will ultimately wind up in the application, so the front-end development is doing double-duty. The web is really the first software development environment that has provided the ability to do this in such a seamless manner. Prior to HTML, prototypes were simply visual tools that were used as references to develop the actual front-end.

Standard: Client Communication

The front-end phase of the FLiP cycle is a critical communication between the developer and the customer. But it is seldom possible to have a customer who is always available during the entire phase. For this reason, FLiP uses a technique that allows recorded dialog with the customer throughout the front-end phase.

What It Is

The FLiP process recommends the use of the DevNotes tool for customer communication. DevNotes provides a threaded dialog at the bottom of every page of the front-end, allowing the customer to comment on the design process from anywhere that a web browser is available.

How It Works

The DevNotes tool, originally created by Hal Helms, is a custom tag (in its ColdFusion implementation) that can be inserted into the OnRequestEnd.cfm template so that it is included at the bottom of every page in an application. Each participant in the design process identifies himself through the DevNotes interface, and can comment and reply to existing comments. Illustration 5 shows a sample page with the DevNotes interface.

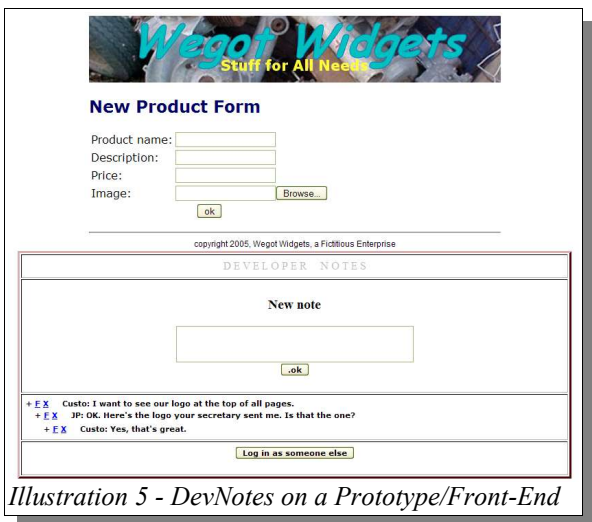


Illustration 5 - DevNotes on a Prototype/Front-End

As issues are identified in through DevNotes, changes are made to the front-end, and the changes are noted and agreed upon, also in the DevNotes interface.

DevNotes are stored in a database, and deleted notes are never actually deleted from the database; they are marked as deleted. This ensures that the entire conversation with the customer is retained as documentation of the process.

Benefits

DevNotes optimizes the ability to accept customer input during the front-end phase. They help maintain a continuous dialog with the customer

throughout the phase. By removing the requirement that the customer be physically present for consultation, DevNotes maximizes the input the customer has to this critical process.

From the developer's perspective, DevNotes provides critical documentation of the customer's wishes with regard to the application's appearance and behavior. Besides the documentation provided by the front-end itself, the developer retains valuable information about the process required to get to the end point, housed in a database that can be used to generate post-mortem reports. This type of information not only provides project documentation for the current project, but also invaluable codified experience that can be leveraged against future projects.

DevNotes also provide assistance in determining the end of the front-end phase of the FLiP cycle. When all the customer's questions have been answered satisfactorily, and the developer has all the information needed to build the application, the two parties sign off on the front-end design. This step is known as prototype freeze.

Standard: Architecture

Far too many software development projects get well into their development cycle without any guiding architecture for the code. Fusebox projects never begin coding before the architecture is firmly established.

What It Is

The architecture phase of FLiP is the step in which control of the project shifts from a collaboration between the developer/ designer and the customer into the hands of the developer/architect. During this phase, the architect uses the front-end to design the application architecture in terms of Fusebox elements: circuits, fuseactions and fuses.

How It Works

The architect works with a printed copy of the front-end, marking up each page in terms of dynamic content, variables to be passed, and exit points. Once this markup is finished, it serves as the foundation for the creation of fuses, fuseactions, and circuits within the Fusebox framework.

Many Fusebox architects use a tool called Mind Mapping to plan out the Fusebox framework. Using such a tool allows automated generation of the basic framework, ready for coding to begin.

Benefits

The architecture phase of FLiP results in an application framework that is designed based on the front-end specified by the customer. This means the entire design of the application is based on its purpose. The architect organizes the application's circuits based on the functionality depicted in the front-end.

Once the application framework has been designed, it can be readily understood by anyone who is familiar with Fusebox, and learned fairly easily even by developers who haven't used Fusebox before. The organization of the application's fuses makes their functionality easy to understand.

Another result of the architecture phase is the creation of the circuit configuration files, which act as roadmaps for each circuit. Even before any actual code is written, the program flow can be read from the circuit configuration files. This provides another opportunity for the architect to backcheck the system logic before releasing fuses for coding.

Although integral to the architecture phase of FLiP, writing a Fusedoc for each fuse file is such an important part of the FLiP that we'll look at it as a separate standard.

Standard: Code the Documentation

Most programming environments teach coders to document their code using comments interspersed throughout the source code module. Conversely, Fusebox specifies that each module (fuse) will be documented before it is written.

What It Is

The standard for documenting fuses is called Fusedoc. It is a documentation standard that was created by Hal Helms and originally based loosely on Javadoc.

How It Works

Fusedoc, as of version 2.0, is an XML vocabulary designed to document fuses in a Fusebox application. A Fusedoc specifies the Responsibilities, Properties, and Input-Output of a fuse.

As the architect is designing the application based on the front-end, a Fusedoc is created for each fuse file. In the case of display fuses, HTML from the front-end is added to the file after the Fusedoc. In the case of query fuses, a query simulation (QuerySim) may be added after the Fusedoc. The result is a file that tells the coder all the information needed to write the fuse. This is called a fuse stub.

Listing 1 shows a simple fuse stub for a query fuse.


```
<!---
<fusedoc fuse="qryGetCatalog.cfm"
          version="2.0"
          language="ColdFusion">
  <responsibilities>
    I return a recordset of the current
    catalog items.
  </responsibilities>
  <properties>
    <history type="create"
            email="jeff@grokfusebox.com" />
  </properties>
  <io>
    <in>
    </in>
    <out>
      <recordset name="qryGetCatalog">
        <number name="productID"
              precision="integer" />
        <string name="productName" />
        <number name="price" />
      </recordset>
    </out>
  </io>
</fusedoc>
--->

<cf_queriesim>
qryGetCatalog
productID,productName,price
1|Antique Hexaform Device|475.00
2|Autodiscerator|35.00
3|Autospork|19.95
</cf_queriesim>
```

Listing 1 - Fuse Stub

Benefits

Fusedocs are the tool that allows Fusebox to overcome Fred Brooks' Mythical Man-Month. By capturing everything the developer needs to know in order to write a specific fuse, the Fusedoc removes the need to teach the entire development team about the application, its design, its intentions, and so forth. The developer's only task is to complete the code required for the assigned fuse.

Clearly this means that the success of the application's integration phase is dependent on the quality of the Fusedocs. If the architect doesn't do a good job when writing the Fusedocs, the results that come back from coding will not be what the architect expects. Fortunately, with the advent of Fusedoc 2.0, it is much easier to create correct, well-defined Fusedocs. There are also a number of software tools available for use with popular IDEs to assist with creation of Fusedocs.

Standard: Unit Testing

Debugging is one of the most expensive aspects of software development. Fusebox reduces debugging costs through the heavy use of unit testing.

What It Is

Each fuse in a Fusebox application is tested by the developer before it is returned to the architect. This testing ensures that the fuse meets the requirements set for it by the architect, and that it will integrate successfully into its circuit.

How It Works

Unit testing in Fusebox is implemented through the use of test harnesses. A test harness is a piece of code provided by the architect. The test harness establishes the environment that would normally be established by Fusebox, then includes the fuse to be tested.

Test harnesses can be simple to very complex, depending on the degree of testing the architect wishes to perform.

Benefits

Unit testing relieves the vast majority of debugging issues when developing a Fusebox application. Fuses are small, well-focused pieces of code, so debugging a fuse using a test harness is a very simple task, as opposed to the process of isolating

and correcting bugs in an integrated application.

The use of Fusedocs is also leveraged for unit testing, in that a fuse's Fusedoc can be used as input to a test harness generator. Using such a tool, the architect can create test harnesses for an entire application in a matter of seconds.

Once each fuse has been unit tested and returned to the architect, the application can be integrated and use-tested. It is not uncommon for a Fusebox application to run bug-free through its first integration test, thanks to the rigor imposed by unit testing.

Summary

Fusebox and the Fusebox Lifecycle Process (FLiP) are proven techniques for building successful, easy-to-maintain, scalable applications. These techniques were originally codified using the ColdFusion development environment, and have been translated into several other web application environments.

The major benefits seen by Fusebox developers and development teams are the result of a collection of standard, both in coding (Fusebox itself) and methodology (FLiP).

Fusebox uses a centralized controller pattern to control program flow. This removes flow control from basic program modules (fuses) and eases interpretation of program flow.

Program modules are organized into meaningful collections (circuits) based on the tasks the application needs to perform (fuseactions).

Each circuit's tasks are enumerated in a circuit configuration file, which acts as a roadmap of the program flow in that circuit. This approach allows fuses to be decoupled from the application design.

Fuses are small, focused pieces of code that isolate display, query, and action code. Naming conventions make finding and maintaining code extremely easy.

Every fuse is documented using the Fusedoc standard. This enables a coder to write the program

code for a fuse with no additional knowledge of the application at large.

Fuses communicate flow control with the overall application through the use of exit fuseactions. These are specialized variables that route fuseactions from the circuit configuration file to the program code and back, resulting in encapsulated fuse code.

FLiP standards also benefit the development effort by keeping the project on track and well-documented, and moving through a progression from inexpensive to more expensive stages.

Wireframing the application jump-starts the developer's understanding of the customer's business model.

Creating a front-end or prototype of the application before writing code ensure the customer will get the desired end result, and refines the model for the architect's benefit.

DevNotes provide a web-based means for constant communication between the customer and the developer, and a permanent repository of the issues discussed.

Applying skilled architectural techniques to the completed front-end organizes the application into a coherent Fusebox framework, including circuits, fuseactions, and fuses.

Fusedocs allow the architect to communicate specific instructions to the coder, removing the need

to train each developer in the overall nature of the system and destroying the “Mythical Man-Month”.

Every fuse is unit tested before it is accepted by the architect, greatly reducing the number of bugs and consequently the time required to debug the integrated application.

These standards work in concert as a toolkit for the developer to create robust, easy-to-maintain applications that meet the customer's requirements the first time out of the gate. The 70% failure problem has been solved, at least where web applications are concerned.

Appendix A – Coding Summary

This table summarizes the coding standards explained in this book.

| Standard | Implementation |
|------------------------|------------------------|
| Centralized Controller | All calls to index.cfm |
| Organized Code | Circuits |
| Application Roadmap | circuit.xml |
| Small, Focused Files | Fuses |
| Solid Documentation | Fusedoc |
| Encapsulation | Exit Fuseactions |

Appendix B – Methodology Summary

This table summarizes the methodology standards explained in this book.

| Standard | Tool(s) |
|------------------------|--------------------|
| Wireframe | Wireframe Editor |
| Prototype/Frontend | Plain HTML |
| Client Communication | DevNotes |
| Architecture | Mind Mapping tools |
| Code the Documentation | Fusedoc |
| Unit Testing | Test harnesses |

Appendix C - Fusebox Versions

The following table shows how some Fusebox characteristics have been implemented in various versions of Fusebox. Note that Fusebox 4 adds a large number of features that do not have analogs in previous versions, such as tags to handle CFCs and Java classes.

| Characteristic | Fusebox 1 ² | Fusebox 2 | Fusebox 3 | Fusebox 4 |
|-----------------------------|------------------------|-------------------------|------------------------|---------------------------------|
| Centralized Controller | All calls to index | All calls to index | All calls to index | All calls to index |
| Application Roadmap | cfswitch in index | cfswitch in index | cfswitch in fbx_Switch | circuit.xml |
| Common Variable Scope | cf_formurl2 attributes | cf_formurl2 attributes | cf_formurl2 attributes | Fusebox core |
| Global Variable Definitions | index | appGlobals or myGlobals | fbx_Settings | plugin (4.0) fusebox.init (4.1) |
| Circuit Definitions | N/A ³ | index | fbx_Circuits | fusebox.xml |
| Documentation | N/A | Fusedoc 1.0 | Fusedoc 2.0 | Fusedoc 2.0 |
| Flow control | Hard-coded in fuses | Exit Fuseactions | Exit Fuseactions | Exit Fuseactions |

-
- 2 Fusebox versions weren't officially numbered until
 3. 1 is essentially "where it started" and 2 is "how it grew before 3", in retrospect.
 - 3 Early Fusebox didn't have a circuit/fuseaction distinction.

| Characteristic | Fusebox 1 | Fusebox 2 | Fusebox 3 | Fusebox 4 |
|----------------------------|----------------------|----------------------|----------------------|--|
| Content block manipulation | N/A | cf_bodycontent | cf_bodycontent | Content Component Variables (Fusebox core) |

Appendix D – Fusebox Glossary

action fuse - a template that does not present any data to the user, or interact with the database. Usually has act_ as its prefix.

application configuration file - the XML file where circuits, parameters, global fuseactions, and plugins are defined (fusebox.xml.cfm; Fusebox 4.x).

architect - the person responsible for the development of a Fusebox application. Oversees the entire project, and is responsible for circuit design and Fusedocs.

CCV – See content component variable

circuit - a collection of fuseactions. Circuits are groups of program behavior arranged due to common behavior. For example, all fuseactions having to do with management of user accounts might be placed in a circuit called Users. Deciding which fuseactions belong in which circuits is part of the architect's job.

circuit configuration file - the XML file where fuseactions are defined for a circuit (circuit.xml.cfm; Fusebox 4.x).

content component variable - a variable that holds some generated content for use within a Fusebox site.

core file - the file that handles the "heavy lifting" central to Fusebox processing. There are four core files in Fusebox 4: the Runtime, the Loader, the

Transformer, and the Parser.

daily build - the result of each day's writing of code. Daily builds can be plugged into the prototype to demonstrate progress to the client.

designer - the person responsible for the visual design of a Fusebox application.

DevNotes - a small utility used to provide discussion and feedback capability at the bottom of prototype pages.

display fuse - a template that provides information to the user and/or allows user interaction. Display fuses do not interact with the database. Usually has `dsp_` as its prefix.

exit fuseaction - a variable that contains a fuseaction that may be used to exit a given fuse. For example, if a fuse uses a button named Save, the architect might create an XFA named XFA.btnSave, with the value "users.saveUser" stored in it. This allows fuses to be coded with variable destinations, e.g., `fuseaction=#XFA.btnSave#`, instead of being hard-coded.

exit point - A place where processing leaves a fuse, such as a form action or redirection.

FLiP - Fusebox Lifecycle Process: a web development methodology that includes wireframes, prototypes, DevNotes, formalized architecture, distributed coding, and unit testing.

front end - sometimes used to refer to a prototype.

Because a Fusebox prototype becomes the front end for the finished application, this term is often used when talking to clients.

fuse - an individual code file that contributes to processing a fuseaction. One or more fuses may contribute to a fuseaction. Typically divided into display, action, query, and sometimes url categories.

fuse file - see fuse

fuseaction - a collection of fuses that perform an action within a Fusebox application. For example, a fuseaction might consist of a query fuse to fetch data from a database and a display fuse to present the data to the user. Every request in a Fusebox 4 application employs at least one fuseaction.

Fuseboxer - a person who uses Fusebox.

fusecoder - a person who writes the code for a fuse, as specified in its Fusedoc.

Fusedoc - a specification for documenting the inputs and outputs to be used by a fuse. Also refers to the documentation block at the top of a fuse file. The architect is responsible for creating the Fusedoc, and a fusecoder is responsible for writing the fuse's code according to its Fusedoc.

home circuit - The top-level circuit in a Fusebox application. Also called the main circuit.

prototype - a static representation of the final application. A prototype looks and acts in every way, except the presence of production data, as the

final application will. Uses query sims to mimic the presence of a database.

prototype freeze - the point in time at which the client accepts the prototype as the plan for development.

query fuse - a template that interacts with the database. Usually has qry_ as its prefix.

query sim - a query simulation. Query sims use Hal Helms' QuerySim tag to generate ColdFusion recordsets. They are used in place of live queries until the database has been constructed.

target circuit - in the context of a fuseaction, the circuit in which the requested fuseaction is located.

test harness - a template that establishes a valid Fusebox environment for the purpose of testing a fuse file.

unit test - a test of an individual fuse file. Unit tests are performed using test harnesses.

wireframe - a text-based representation of the application's business logic. Wireframes are used to develop the application concept prior to prototyping.

XFA - see exit fuseaction

Alphabetical Index

| | |
|--------------------------------|--|
| architecture..... | 7, 21, 40, 50p., 67 |
| Architecture..... | 50, 62 |
| CF-Talk..... | 3 |
| circuit.. | 13, 21pp., 25p., 37p., 50p., 56, 58p., 61, 63, 65, 67p. |
| Circuit..... | 21p., 61, 63, 65 |
| David Huyck..... | 5 |
| DevNotes..... | 47pp., 59, 62, 66p. |
| Erik Voldengen..... | 4 |
| Extended Fusebox | 4 |
| FLiP..... | 1p., 9pp., 40p., 43p., 47, 49pp., 58p., 66 |
| Frederick Brooks..... | 35 |
| front-end..... | 40, 44pp., 59 |
| fuse.. | 4p., 13p., 16pp., 22pp., 37pp., 50pp., 55pp., 63, 65pp. |
| Fuse..... | 1pp., 16pp., 21pp., 31pp., 37, 39pp., 44p., 50pp., 55pp. |
| fuseaction..... | 13p., 22, 24pp., 30, 37pp., 50, 58p., 65pp. |
| Fuseaction..... | 25, 37, 61, 63 |
| Fusebox 3..... | 5, 23p. |
| Fusebox Lifecycle Process..... | 1, 9, 40, 58, 66 |
| fusebox.org..... | 5 |
| Fusedoc..... | 4, 6, 33pp., 51p., 55, 57p., 60pp., 65, 67 |
| Gabe Roffman..... | 3 |
| Hal Helms..... | 4p., 33, 47, 52, 68 |
| Jeff Peters..... | 4 |
| John Quarto-vonTivadar..... | 4 |
| John Quarto-vonTivadar | 5 |
| Joshua Cyr..... | 3, 4 |
| methodology..... | 1, 6, 8pp., 21, 40, 58, 62, 66 |
| Methodology..... | 10, 40, 62 |
| Michael Dinowitz..... | 3 |
| Michael McKellip..... | 5 |
| Mike Stark..... | 5 |
| Nat Papovich..... | 4 |

| | |
|--------------------------------|----------------------------------|
| program flow..... | 15pp., 23, 25, 31, 51, 58 |
| prototype..... | 40, 44, 46, 49, 59, 66pp. |
| Prototype..... | 44, 62 |
| Rich Tretola..... | 5 |
| roadmap..... | 23, 25p., 51, 58 |
| Roadmap..... | 23, 61, 63 |
| Robi Sen..... | 3, 4 |
| scale..... | 4 |
| scope..... | 18pp., 27, 34 |
| Scope..... | 18, 63 |
| Steve Nelson..... | 3, 4 |
| unit testing..... | 7, 40, 56p., 67 |
| Unit testing..... | 56 |
| Unit Testing..... | 56, 62 |
| What's New in Fusebox 4.1..... | 5 |
| wireframe..... | 41, 43, 45, 66, 68 |
| Wireframe..... | 41, 62, 68 |
| XFB..... | 4 |
| XML..... | 5 |

Illustration Index

| | |
|--|----|
| Illustration 1 - Typical CFML Processes..... | 15 |
| Illustration 2 - Fusebox Process..... | 16 |
| Illustration 3 - Wireframe Map..... | 42 |
| Illustration 4 - Wireframe Page..... | 42 |
| Illustration 5 - DevNotes on a Prototype/Front-End..... | 48 |