Design Patterns and ColdFusion

Sean A Corfield
Chief Systems Architect
Broadchoice, Inc.

Patterns describe problems and solutions

"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."

-- Christopher Alexander, architect

Alexander is not a software architect - he is a building architect!

1. Hierarchy of Open Space - big space in front, small space behind
2. Intimacy Gradient - enter into public areas, navigate into private areas

Who am I?

- VP Engineering, Broadchoice
- Previously:
  - Mgr, Adobe Hosted Services
  - Sr Architect, Macromedia IT
  - Sr Consultant, various web co.
- Architect / Developer:
  - ColdFusion since 2001
  - Java since 1997 (Groovy since 2008!)

Also manager of Bay Area ColdFusion User Group!

What is this about?

- Common problems
- Common solutions
- Trade offs

Show of hands:
1. Who's using ColdFusion 8? 7? 6.1? 5 or earlier?
2. Who's using CFCs? Who thinks they're writing OO code? Patterns?

I want to remove some of the mystery around patterns by showing you how pervasive they are - you're probably already using some patterns without even realizing it. I hope that you will come out of this session with a better understanding of what patterns are how they can help you in your day-to-day programming work - and that you will be inspired to learn more about patterns and extend yourself to become a better developer.

Our roadmap

- Some simple, common patterns
- Patterns and languages
- Selecting and using patterns
- Frameworks and patterns
- Recommended reading

Simple, common pattern examples

A common problem

- My web site has the same stuff at the top and bottom of each page and I don't want to duplicate that in every page!

A common solution

- Put the common stuff in separate files (header.cfm and footer.cfm) and include them on every page.

This is just one possible solution - a simple one.

Trade offs

- No duplication of header and footer code now (+)
- Still have boilerplate text (includes etc) in every page (-)
- If some pages need a different header / footer, you either make new header / footer files for that page or add conditional logic to the header / footer files (-)

Again, there are other possible solutions, more sophisticated solutions, that have different trade offs.

Including header and footer

```
<cfinclude template="header.cfm" />

<h1>Welcome to my site!</h1>

<p>Let's learn about design patterns in ColdFusion!</
p>

<cfinclude template="footer.cfm" />
```

Another solution might be to use content variables and a layout but we're keeping things simple here.

Patterns provide a common vocabulary

"An important part of patterns is trying to build a common vocabulary, so you can say that this class is a Remote Facade and other designers will know what you mean."

-- Martin Fowler

If we use different names for the same things, we cannot communicate effectively with each other

Pattern: Composite View

```
<cfinclude template="header.cfm" />

<h1>Welcome to my site!</h1>

<p>Let's learn about design patterns in ColdFusion!
</p>

<cfinclude template="footer.cfm" />
```

- A composite view is a view that includes other views (conceptually or literally - as with all patterns, there are many, many ways to implement this!)

From Core J2EE Patterns - specifically gives the example of JSP includes.

What is a pattern?

- Patterns have four parts:
  - Name - the common vocabulary
  - Problem - “forces” that determine when the pattern is applicable
  - Solution - a **template** for solving the problem
  - Consequences - “pros **and cons**”

It's important to remember that a pattern is all four parts together. A pattern is not just a solution. Patterns may help us figure out one or more possible solutions to any given problem. The consequences should allow us to figure out better solutions in specific circumstances.

A common problem

- I need a single instance of an object and I need it easily accessible everywhere in my application!

A common solution

- Create it at application startup and put it in application scope so you can get at it anywhere.

Trade offs

- Initialization is all in one place (+)
- Easy access to objects application-wide (+)
- Application scope is referenced directly everywhere which breaks encapsulation to some degree (-)

We can resolve the negative and avoid having objects refer to application scope by “injecting” dependencies – passing in the other objects they need – when those objects are created.

APPLICATION SCOPE USAGE

- This is a common idiom in ColdFusion:
- In Application.cfc's onApplicationStart():

```
application.logger = createObject("component","LoggingService");
```

- Global access:

```
application.logger.info("This is a common idiom.");
```

Show of hands: who is using Application.cfc?

Still using Application.cfm? You have to use conditional logic and locking to perform initialization at application startup.

Application / onApplicationStart() is a pattern

- Name - Singleton
- Problem - some classes require that only a single instance exists and that it is accessible from a well-known place
- Solution - provide a way to create the single instance and a global way to access that single instance
- Consequences - controlled access; ability to use subclass; complexity of ensuring only one instance...
- ...in some languages!

Subclass - or *any* class that provides the same API!

Java has no global variable scope and no built-in infrastructure for initialization at startup (static members don't have quite the same semantics). C++ also has different static member semantics. Both languages require specific code to provide on-demand initialization and prevent multiple instances.

Java Singleton (one possible solution)

```
import java.util.HashMap;
public class Singleton {
  private static HashMap map = new HashMap();
  protected Singleton() {
     // Exists only to thwart instantiation
  }
  public static synchronized Singleton getInstance(String classname) {
     Singleton singleton = (Singleton)map.get(classname);
     if(singleton != null) {
        return singleton;
     }
     try {
        singleton = (Singleton)Class.forName(classname).newInstance();
     }
     catch(Exception e) {
        e.printStackTrace();
     }
     map.put(classname, singleton);
     return singleton;
  }
}
```

This is a slightly biased comparison – this Java code is more than just a singleton: it is really a factory that creates a single instance of any class you need. It uses a cache of objects ("identity map" pattern) to ensure it only creates one instance of each object. But it does not really solve the problem of unique instances since you can create, say, a MyClass instance independently of the Singleton. "synchronized" is required – the equivalent of <cflock> in CF.

A common problem

- I have several components that make up my user security logic but I don't want to expose them to other code, in case I want to change how I implement it!

A common solution

- Create a new component that has a nice, simple, high-level API (set of methods) and have it call all your security components for you.

Trade offs

- Application code no longer needs to know about the internals of your security system (+)
- Application code can still get at the "low-level" components if it needs to (+/-)
- Some redundancy of methods between the API and the underlying components (-)

Service objects and APPLICATION scope

- Authentication system contains
  - User, Credentials, Group, Permission components
- AuthenticationService provides an API:
  - login(username,password), logout(), getUser()
- AuthenticationService object stored in application scope

```
success = application.authService.login("jane@doe.com","xxx");

if ( success )
     name = application.authService.getUser().getFirstName();
```

That's another pattern!

- Name - Facade
- Problem - as a subsystem grows, it becomes more complex with a large number of smaller objects
- Solution - introduce a single component that provides a simple API to the set of components within the subsystem
- Consequences - shields clients from the inside of the subsystem, reducing complexity and coupling; does not prevent access to subsystem if needed
- Often only one instance of a facade is needed (singleton)

In this pattern we see another common pattern – a pattern of patterns: that many patterns depend on other patterns. Whilst some patterns are small and self-contained, many patterns involve several objects and interactions and their implementation usually poses problems that other patterns can help solve.

A common problem

- I need to apply standardized security logic to every request in my application!

Show of hands: who uses an application framework like Fusebox, Model-Glue or Mach-II? You'll recognize this...

A common solution

- Make all requests go through a common piece of code and apply that logic centrally, using URL parameters to indicate what the request is really for:
  - Before: /catalog.cfm and /product.cfm
  - After: /index.cfm?page=catalog and /index.cfm?page=product

Trade offs

- Full control over every request (+)
- All URLs look very similar (+/-)
- index.cfm may get complicated (-)

As before, we can resolve the negative by applying additional design patterns to help simplify the code (by refactoring and encapsulating the logic elsewhere - such as a framework).

Flying without a framework

```
<cfif not structKeyExists(session,"user")>

        <cfinclude template="login.cfm">

<cfelse>

        <cfinclude template="#page#.cfm">

</cfif>
```

login.cfm displays a form where users login (and also contains the processing code for logging in, setting up the user (object) in session scope etc).

Another pattern (of course)

- Name - Front Controller
- Problem - need to apply consistent logic across all requests in an application (security, layout etc)
- Solution - route all requests through a single file that decides how to process the request
- Consequences - centralized control, easy to change how requests are handled in a consistent manner, moves logic out of individual pages into controller (which can become complex)

Pretty much all the application frameworks implement this pattern - ColdBox, Fusebox, Mach-II, Model-Glue.

Patterns and languages

Patterns are not code!

"The examples are there for inspiration and explanation of the ideas in the patterns. They aren't canned solutions; in all cases you'll need to do a fair bit of work to fit them into your application."

-- Martin Fowler

"... the code examples ... are deliberately simplified to help understanding, and you'll find you'll need to do a lot [of] tweaking to handle the greater demands you face."

-- Martin Fowler

If you look in the books, magazines and blog articles, you will almost always see explanations of patterns accompanied by code. A lot of people need to see code to "get it" - show of hands: who feels they learn best by seeing example code?

Unfortunately, with patterns, this can be misleading at best and sometimes downright dangerous. Patterns inherently have trade offs and each specific implementation of a pattern has additional trade offs. The example code for any given pattern may just not be a good solution for you - even if the pattern itself *is* applicable to your situation!

Patterns and languages

- **Design** patterns are usually generic
  - Can be applied during the **design** phase
  - Can be applied to any implementation language
- Some patterns are language-specific
  - Some languages can implement certain patterns directly
  - Some languages require pattern implementations to be constructed

Patterns are not code – they're about design – which means they are broadly applicable but not universally applicable. Composite View is from the Core J2EE Patterns book but is clearly applicable to ColdFusion – but certainly not all the patterns in that book are applicable to ColdFusion.

An object-oriented bias?

- Singleton and Facade refer to objects
- Most design patterns **are** focused on OO design
- With a procedural language, inheritance, encapsulation and polymorphism become "constructed patterns"
  - X11 (written in C) uses structs and pointers to functions - and conventions - to implement object-oriented techniques

"Although design patterns describe object-oriented designs, they are based on practical solutions that have been implemented in mainstream object-oriented languages like Smalltalk and C++ rather than procedural languages ... or more dynamic object-oriented languages." -- GoF

In other words, many patterns have been identified from common solutions in languages like C++, Java, C# etc (whether Smalltalk is "mainstream" is a matter of opinion!).

Patterns and ColdFusion

- Most **design** patterns can be applied to ColdFusion
- ColdFusion, Java, C#, Smalltalk, C++, Groovy and Ruby all have slightly(!) different language features
- Some language-specific patterns may not apply to ColdFusion - or may apply in different ways
- We should expect to see some ColdFusion-specific patterns evolve - but it will take time for them to get standard names

The difference between a *design* level pattern and a language-specific pattern is not always obvious, especially when patterns are illustrated by implementations in a single language (GoF is good for this since it uses Smalltalk and C++ which are very different languages - Fowler less so since it uses two similar languages: Java and C#).

Which way to go?

- ColdFusion...
- ...like Java?
- ...or like Ruby?

We're at a fork in the road. We've been "raised" to think of ColdFusion (MX+) as a rapid application development layer on top of Java. Indeed, we hear "ColdFusion *is* Java". We've been conditioned to think that what's good for Java is good for ColdFusion - we think we must write OO code and we must use design patterns and we must follow Java's lead.

Well, guess what? Not all Java patterns - and I mean that in the broadest sense - are applicable to ColdFusion. In my opinion, we should look to dynamic languages like Smalltalk and Ruby for inspiration...

Ruby / Ruby on Rails patterns

- Famously uses Active Record design pattern for managing persistence of objects and is core to how Rails works
- Dynamic finder methods

```
find_by_email_and_password("jane@doe.com","xxx")
find_by_colA_and_colB_and_colC(arg1,arg2,arg3)
```

- Dynamic methods are not possible in Java / C#
- Dynamic methods are possible in...
  - ...Ruby (obviously) - using :method_missing...
  - ...Smalltalk - using #doesNotUnderstand...
  - ...Groovy - using invokeMethod()...
  - ...ColdFusion - using onMissingMethod()

Show of hands: Who has heard of Ruby on Rails? Who has used it?

Active Record is broadly applicable (in other languages) but Rails has made it really popular.

Name of method is parsed at runtime to determine what the arguments mean.

onMissingMethod()

```
<cffunction name="onMissingMethod">

  <cfargument name="missingMethodName">

  <cfargument name="missingMethodArguments">



  <cfif left(missingMethodName,8) is "find_by_">

      <!---
      parse col and col and col...
      SQL query with arguments matched to columns
      --->

  <cfelse>
      <!--- no such method: throw exception --->
  </cfif>

</cffunction>
```

The full solution is too complex to show on a slide but this should give you an idea of the structure. onMissingMethod() is invoked when a call is made to a function that is not explicitly defined in an object. You can then look at the function name and the arguments and decide what to do. Rails uses this technique to implement a number of helper methods but the “pattern” doesn’t have a name yet.

Patterns and interfaces

- Often used in design pattern code examples (e.g., Java) or UML diagrams that illustrate concepts
- Smalltalk has no interfaces (but was used to illustrate Gang of Four patterns)
- Interfaces are not needed but are useful as a descriptive tool in explaining pattern implementations

If you read about patterns, you'll see interfaces used extensively. You'll see Java code examples with interfaces all over the place. You can be forgiven for thinking that you need interfaces to implement patterns. The "classic" design patterns book used both C++ *and* Smalltalk to illustrate patterns and neither language has interfaces (C++ has a construct that is similar to an interface but Smalltalk has no such construct). Ruby does not have interfaces either.

Patterns and interfaces (continued)

- ColdFusion 8 introduces <cfinterface>
- You can use it to help implement patterns the "Java way"
- Or you can use "duck typing"** and conventions to implement patterns the "Smalltalk way" (or "Ruby way" or "Groovy way")

** Duck typing: using weak typing (returntype="any" and type="any") and passing in any object that provides the necessary methods to satisfy the "contract"

<cfinterface> specifies the methods a component provides (API) but not how they behave. <cfcomponent> *implements* the interface to specify the behavior. An interface can have many implementations. I was an early - and vocal - advocate of adding interfaces. I submitted the original ER and rallied the community to vote for it. But duck typing is more powerful and appropriate for ColdFusion - onMissingMethod() lets you implement any methods dynamically.

Selecting and using patterns

Patterns provide guidance

- Patterns have consequences that help us decide which solution is more appropriate in a particular situation

  "Using, say, an object-relational mapping tool still means that you have to make decisions about **how** to map certain situations. Reading the patterns should give you some **guidance** in making the choices."

  -- Martin Fowler

This is all about choosing the "best" solution for your particular situation. Or at least the "better" solution given a number of possible choices. There are no absolute best solutions - the choice depends on the forces (problem specifics).

Patterns are not just solutions

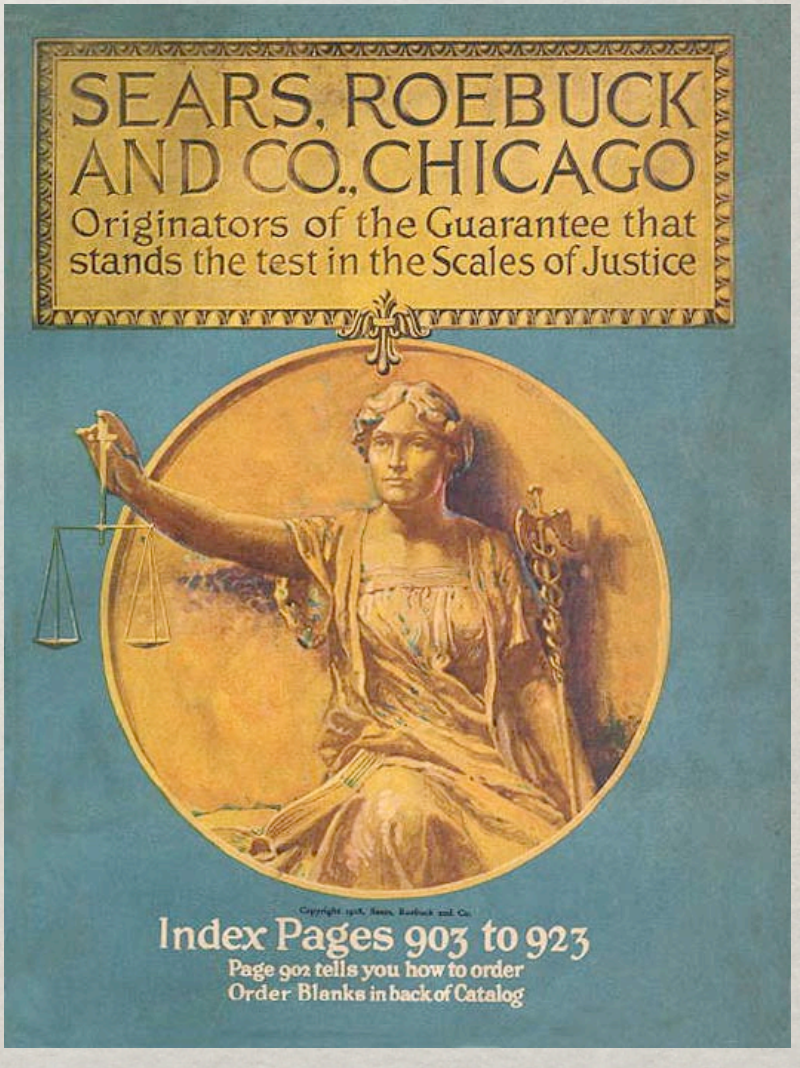
"When people begin to look at design patterns, they often focus on the solutions the patterns offer. This seems reasonable because they are advertised as providing good solutions to the problems at hand.

However, this is starting at the wrong end. When you learn patterns by focusing on the solutions they present, it makes it hard to determine the situations in which a pattern applies. This only tells us **what** to do but not **when** to use it or **why** to do it."

-- Alan Shalloway

Pattern catalogs

- Not like the SEARS catalog...
- You don't just put together a collection of patterns and... voilà, instant application!
- Instead, think of academic catalogs that organize and classify items so that you can find things easily



When you first start using design patterns, there is the temptation to treat these books like a grocery list of ingredients that are necessary for a successful application. Remember that design patterns are about forces and applicability as much as they are about templates for solutions. They can be inspiration when you're stuck on a problem – they can provide guidance when you're not sure which way to solve a problem. They are not "building blocks".

Types of patterns

- The “classic” software design patterns book has:
  - Creational, Structural, Behavioral
- Core J2EE Design Patterns has:
  - Presentation Tier, Business Tier, Integration
- Martin Fowler's enterprise application patterns book has ten categories including three for object-relational alone

I'm not going to go too deep into classification of patterns but I want to give you a sense for how they are typically organized within the books (catalogs), partly because each book handles it differently.

Design Patterns (Gang of Four)

- Creational patterns
  - Abstract Factory, Builder, Factory Method, Prototype, **Singleton**
- Structural patterns
  - Adapter, Bridge, Composite, Decorator, **Facade**, Flyweight, Proxy
- Behavioral patterns
  - Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method, Visitor

The classic design patterns book is very generic and therefore classified patterns based on how you build stuff, how you organize stuff and how you implement your workflows. These patterns are very broadly applicable (although a number of them are not actually very common in ColdFusion applications – because we don't often encounter the problems they solve).

Core J2EE Patterns

- Presentation tier
  - Intercepting Filter, Context Object, **Front Controller**, Application Controller, View Helper, **Composite View**, Dispatcher View, Service To Worker
- Business tier
  - Business Delegate, Service Locator, Session Facade, Application Service, Business Object, Composite Entity, Transfer Object, T O Assembler, Value List Handler
- Integration tier
  - Data Access Object, Service Activator, Domain Store, Web Service Broker

The Core J2EE Patterns book is organized very differently and will look much more familiar to ColdFusion developers because we (mostly) already think in terms of the tiers mentioned in the book. However, several of these design patterns have appeared as a response to certain problems which are Java-specific (in particular addressing the overhead of communications in client-server architectures based on EJB).

Frameworks and patterns

Frameworks are full of patterns

- Frameworks are designed to solve common problems
- Application frameworks usually implement several patterns
  - Front Controller - everything goes through index.cfm
  - Model-View-Controller - segregation of the presentation and business tiers
  - Identity Map - a cache accessed by a unique key
  - Context Object - encapsulating information about a request, e.g., event
  - ...

Since patterns are applicable to common problems and offer templates for solutions to those problems, it should be no surprise that application frameworks – which all solve similar problems – are implementing a number of common patterns (often in very different ways).

Some specific patterns in frameworks

- ColdSpring
  - Chain of Responsibility - each “aspect” calls methods on the next “aspect” until the underlying business object is reached
  - Identity Map - cache of objects accessed by “id” (bean name)
  - Proxy - same API as your business objects but intercepts method calls to execute before, after or around “advice”
- Model-Glue 2.0 aka “Unity”
  - Adapter - to provide a common API to different persistence engines
- Tartan
  - Command - encapsulates behavior in an object with an execute() method

Patterns in Reactor

- Abstract Factory
  - To manage objects for different databases
- Active Record
  - Each business object knows how to save and load itself
- Factory Method
  - Data access objects, gateways, metadata and records
- Transfer Object
  - A lightweight object for moving data between application tiers
- ...and others

Patterns in Transfer

- Data Mapper
  - Maps your business objects to & from tables in the database
- Decorator
  - Extend the functionality of generated objects by adding your own methods or "overriding" the generated methods
- Factory Method
  - Business objects are created by name (and may be decorators or "basic" generated objects)
- Identity Map
  - Object cache, each object is referenced by its primary key
- ...and others

Recommended reading

Design Pattern Books (I)

- There are a number of good books about design patterns
- Most of them assume knowledge of OO
- Most of them are "catalogs" that you can dip into as needed

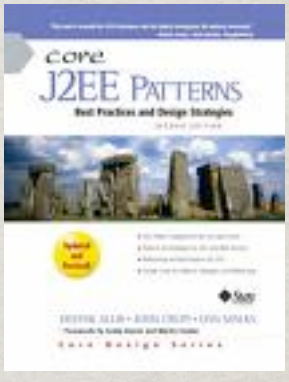
Design Patterns
"Elements of Reusable Object-Oriented Software"
Gamma, Helm, Johnson, Vlissides

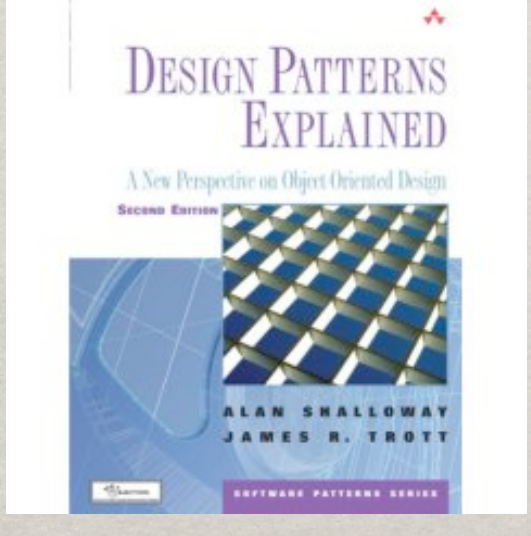
The "Gang of Four" Book containing 23 patterns

Design Pattern Books (II)

Core J2EE Patterns
"Best Practices and Design Strategies"
Deepak Alur, John Crupi, Dan Malik

Fairly Java-specific but quite a few useful lessons for web applications in general





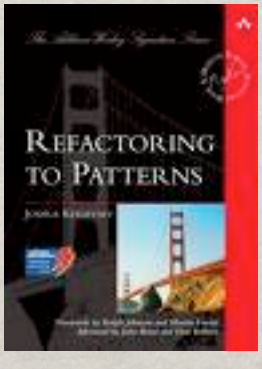
Design Patterns Explained
"A New Perspective on Object Oriented Design"
Alan Shalloway, James Trott

A good introduction to OO and the GoF design patterns

Design Pattern Books (III)

Patterns of Enterprise Application Architecture
Martin Fowler

Catalog of 51 patterns with many variants for each type of software problem



Refactoring to Patterns
Joshua Kerievsky

Practical applications of patterns to improve code structure and maintainability

Thank you!

- Any questions?
- Contact me:
  - sean@corfield.org
  - http://corfield.org/