Software Architecture With ColdFusion:
Design Patterns and Beyond

Topics Outline Prepared by Simon Horwith for CFUnderground 6

Some Terms:

Architecture – the manner in which the components of a computer or computer system are organized and integrated (Webster)

OOP – Object Oriented Programming: representing the parts of an application as Objects that interact with each other. Objects typically represent real world entities. OOP languages typically support Classes, Objects, Messages, Inheritance, and Polymorphism.

UML – Unified Modeling Language: an industry standard vocabulary for defining software structure. There are many types of UML diagrams: use case, class, collaboration, sequence, activity, deployment, and more.

Design Pattern – A recurring solution to a recurring problem

Anti-Pattern – Recurring mistake made in software designs

Aspect Oriented Programming (AOP) – Software components represent "crosscutting concerns" which are then "woven" – it is the next generation school of thought for software development.

Notes on Terms:

Some Topics:

An application has many tiers. Typical tiers are:

- Client-side presentation - presents UI to user
- Server-side presentation – feeds data to client-side presentation tier
- Server-side Business Logic – perform actions
- Server-side domain model – the entities
- Integration Tier – talks to other systems

Why Patterns?
Design patterns define ways to represent components in a system. Breaking a component into several smaller components, which are broken into several smaller components, etc. – becoming more specialized as they become smaller. This allows changes in one component to not effect other components and allows components to be reused more easily. This is commonly known as encapsulation and loose coupling.

What makes an application a good one?
There are four categories by which we measure success as developers:

- Scalability – how well does an application perform as load (requests) is increased?
- Reliability – does the application behave as expected all of the time?
- Extensibility - how easily are changes accommodated?
- Timeliness – is the application delivered in a reasonable amount of time?

OOP and patterns helps with all of these things… largely by use of centralized, decoupled, reusable functionality and components.

What entities do we commonly have to consider?
CFM pages
Custom Tags
CFCs
Database
File System
RIA Interfaces
External Applications
Other technologies

Additional Notes:

Design Patterns

There are thousands of design patterns and dozens of ways to categorize them all. Common general categories are Presentation Tier, Business Tier, Messaging, and Anti-Patterns. I like to also categorize them as: Factory, Delegate, Hub, Façade, Filter, and Controller… because many patterns are based on these concepts. Always remember: there are many patterns that solve a problem – which is best depends on requirements and how you define success. Also remember that not all patterns are applicable (relevant) in CFML development, though the majority are.

Pattern	Purpose	Description	
Model-View-Controller (MVC)	Separate presentation tier into separate components	Controller receives and processes user input, updates Model, and selects view. View presents UI and uses Model data. Model stores data and performs business logic.	
Front Controller	A single central component performs common functions	Front controller receives all requests and performs common functions like logging, passes data to page (request) controller which talks with Model. Page controller then selects a View.	
Decorator	Allows functionality to be dynamically added to a Front Controller.	Decorator object is a configurable filter that intercepts requests and hands control and data (including a chain definition) to first decorator. That decorator hands control to next decorator in the chain, etc.	
Composite Entity	Efficiently represent Domain Model Entities by aggregating data	Model is broken into many small objects in the back, aggregated data is	

	from many smaller objects.	presented to the View.	
Domain Object Model	Use objects to represent the business logic/application concepts	Objects in Model are connected to each other in ways that represent business relationships.	
Data Transfer Objects (DTO)	Improve performance by reducing the number of objects passed between application tiers	Smaller data container objects are passed from the business tier to the view.	
Data Transfer Hash	Improve performance by reducing the number of objects passed between application tiers	Structures (Hash variables) are passed from the business tier to the view	
Row Set DTO	Improve performance by returning recordset data to the presentation tier.	Queries are returned directly to the presentation tier for use rather than returning Objects.	
Data Access Object (DAO)	Separate persistence from code that processes data	Business object or Presentation tier Object has a DAO that it uses to access and update data. DAO talks with persistence layer via other data objects.	
DAO Factory	Hide DAO creation/selection from presentation/business tiers.	Business or presentation tier object requests a DAO from a factory. Factory decides how to create the DAO	
isDirty	Prevent unnecessary writing to the database	DAO tracks whether or not data has changed and when passed to Model, the Model determines whether or not to write to the	

		database.	
Lazy Load	Avoid unnecessary database access	Client requests data from object, object calls DAO methods just to get that data – no extra data ever returned.	
Procedure Access Object (PAO)	Leverage Stored Procedures	Properties are set on the PAO, execute() method then passes those properties to a stored procedure which then sends new values back (if appropriate) to the PAO	
Service to Worker (Dispatcher)	Decouple navigation from Front Controller	Front Controller gets request, performs common functions, then passes request to dispatcher object. Dispatcher determines what actions to perform and uses it's internal Model for persistence and logic. Dispatcher chooses view and uses it's Model to populate it.	
Composite View	Build a view from several sub-views	Dispatcher builds structure of "leaf views" then forwards request to a composite view which replaces generic leaf view names with actual views.	
View Helper	Avoid over specialized views	View uses view helper objects to translate Model data into more usable data model. Either view or helper transforms that data into	

		something useful (HTML, XML, etc.	
Asynchronous Page	Cache remote data efficiently	Subscriber object retrieves data from publisher when it's changed or at intervals. Subscriber transfers that data to the Model. View reads from the Model	
Caching Filter	Minimize page generation	Cache filter intercepts requests and determines whether or not to return a cached version.	
Resource pool	Decrease repetition of object instantiations	All objects requested from a pool. New instance is created and returned if necessary, otherwise an available object is returned (and locked until released).	
Serializable Entity	Persist objects to database	Data passed to a DAO – the DAO serializes the object data as binary stream and stores it in the database	
Table Inheritance	Offer simple mapping between database and objects	Every object has a corresponding database table.	
Tuple Table	Store objects in a database so that they're readable and extensible	Tuple table DAO stores data properties as name/vale pairs in database – one row per pair.	
Business Delegate	Encapsulate how to locate, connect to, and interact with business objects in the presentation tier	Methods of business delegate are called – delegate locates and connects with	

		appropriate service. Business delegate manipulates return values and returns them to the client	
Business Delegate Factory	Simplify use of business delegates	Factory controls what business delegate handles request – creates new ones as needed.	
Service Adaptor	Simplify working with foreign data formats	Service adaptor connects with remote resources and formats return data in easy to use native format.	
Session Facade	Increase performance by presenting remote business objects in an optimal way	Request goes to the remote façade, façade works with local objects and returns data	
ACID Transaction	Perform actions on resources while ensuring they stay in proper state	All business and data access logic routes through transaction manager – it is a gateway to persistence that ensures permanent data changes are Atomic (single action), Consistent (with business rules), Isolated (don't interfere), and Durable (committed)	
Lockable Object	Implement simple locking in shared resource	Before any data modification can occur, object's lock() method is called. Lock is released when actions complete.	
Lock Manager	Create central point for managing locks	DAO and delegates request a lock on a resource from a manager.	

		Manager accepts a primary key and/or identifier.	
Version Number	Provide simple way to track when an object changes	Objects have a version number which is incremented when properties change. Other objects use this to deal with concurrency	
Content Aggregator	Allow single handler to process data that is similar but in a variety of formats	Content aggregator receives data in any one of a number of formats and formats it one way before passing it along.	
Anti-Patterns			
Excessive Layering	Unnecessary layers make application inefficient.		
Leak Collection	Objects, locks, and other resources aren't properly released		
Overstuffed Scopes	Objects with too short or too long life spans are stored in the wrong memory scope		
Magic Template	CFM or Custom Tag does all the work of Model, View, and Controller		
Everything is a CFC	CFCs are great and everything – but too many of them is overkill		

Notes on Patterns and Anti-Patterns: