

Exploring FM Software Architecture

Financial markets (FM) software is one of the most interesting fields in modern software development. Organizations use FM software to administer and manage foreign exchange, commodity, equity, and interest rate transactions. Because of the risk involved in these transactions, the software has to be both robust and responsive to changing user requirements. FM software highlights the challenges to good architecture, design, and implementation.

Investigating this field provides an opportunity to think about solutions for a recurring software problem: Many systems become inflexible, degrade, or fail because of poor encapsulation and excessive coupling. System components become so tightly bound to each other that no single piece can be modified without altering the behavior of the application or damaging other modules.

The nature of FM software makes it ideal for exploring this problem; robustness and responsiveness is valuable to any system, but is virtually indispensable in FM software. Encapsulation controls coupling, reducing the likelihood of software failure and making it easier to respond to changing requirements.

Preface: Coupling and Encapsulation

Coupling is a measure of how much each component is directly dependent on the behavior of other components. If one component changes, how much does the rest of the system have to change to stay in synch? Lower coupling allows each component to respond to changing business needs without requiring synchronized efforts throughout the system. Too much coupling leads to maintenance problems; however, too little is also undesirable. Insufficient coupling causes some problems to be detected late in the system development cycle. One method for avoiding these problems or catching them earlier is the use of well defined software interfaces. These interfaces, called the API, are a form of coupling that reduces the cost and risk of software development. The API is the road-map.

The tension between these two goals - decoupling to reduce maintenance cost and coupling to increase velocity and safety - is the motivation behind encapsulation, one of the core tenets of object oriented programming. To put encapsulation in the context of common experience, consider your plumbing: A hot water heater is a very simple device - one line for the water supply, another for the hot water output, and a power supply. The user interface has only two features; a thermostat and a power switch. Internally the hot water tank is far more complicated, but the homeowner does not want to be exposed to that complexity. The hot water tank encapsulates the water heater and hot water storage subsystems.

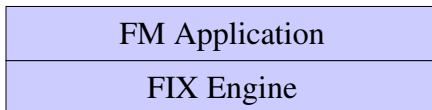
Further down the line, there is another encapsulation layer at the real user interface; a handle or two for selecting the temperature, and a spout for the water. The person washing dishes doesn't care about the hot water tank, its internal complexity, or the

plumbing - they just want warm water.

That is encapsulation: Each layer envelopes a complex problem, and exposes a friendly interface. FM software thrives on these benefits; the bottom layer deals with FIX protocol, the top layer displays a friendly user interface. Each module in between reduces maintenance cost and increases the velocity and safety of adding new features.

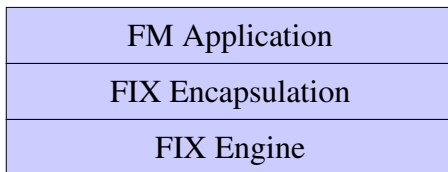
Evolving an Architecture

The architecture of an FM system evolves directly from the core principals of encapsulation. Starting with the simplest practical system, each additional component solves a clearly defined problem. The objective is to reach the least constrictive architecture that is sufficient for the task at hand. In the first iteration there are exactly two components; the FIX engine that allows communication with other organizations, and the application.



That is a good starting point; a simple design is great for learning FIX and the chosen FIX engine. While this is fine for experimenting, it is possible to do better for the production system. The coupling between the chosen FIX engine and the application will be too high - any change in the engine will require changes throughout the application. Within the application, coupling will lead to high maintenance costs and make it difficult to respond quickly and safely to new business requirements.

The first step in the solution is to add a container for the FIX engine. This container, called an abstraction layer, reduces coupling between the application and the FIX engine. When a new version of the FIX engine is released, the upgrade cost is reduced. If the engine vendor raises the price or goes out of business, the migration cost is held to a minimum.



The application is now protected from the expense and failures frequently associated with low level software upgrades. This doesn't mean that each upgrade will be as simple as an oil change, but it does mean the problems will be trapped in a single tier. With the engine issue in hand, the next step is to ensure that the system can only create rational messages.

This system can handle spot, forwards, and options, but there is nothing in the architecture to prevent mixing these types irrationally - adding a strike price to a spot

trade, for example. The front end developers have to focus on making the user interface friendly and intuitive; they should not have to think about the rules for forming a proper message. Adding a rational message encapsulation tier keeps the message details isolated, and allows the front end developers to respond to the business more rapidly.

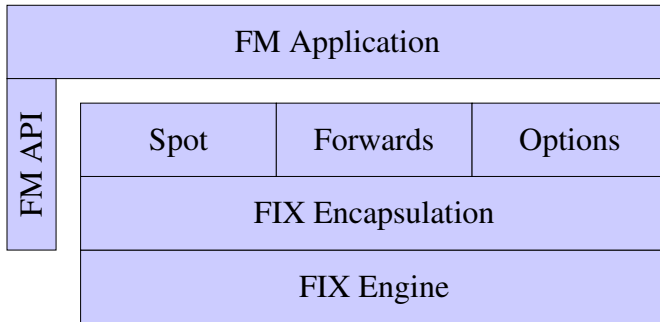
FM Application		
Spot	Forwards	Options
FIX Encapsulation		
FIX Engine		

With the addition of rational message encapsulation, the system is now more responsive to changing business needs. Since irrational messages are no longer possible, it is also more robust. However, this evolutionary step introduces two new challenges; there is a new business issue related to the choice of lateral dividing lines, and a new technical issue regarding coupling between the application and the laterally divided tier.

The more apparent oddity from the business perspective is the unnatural division of message types. To the business, markets are divided primarily into foreign exchange, rates, equities, and commodities, and only secondarily into options, forwards, and spot. Applications - in their architecture, their design, and their implementation - should express the business view of the world. This is a critical issue, but is best handled after the technical problem.

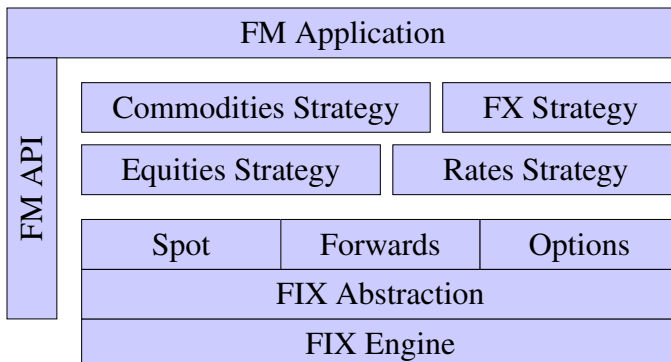
The technical issue is a coupling problem between the application layer and the rational message tier - spot, forwards, and options. The application must now know how to use each of the three components. This leads to a lot of code saying, "In spot mode, do this; in options mode, do that." That additional logic is a breeding ground for bugs, and is not necessary. It can be eliminated using another type of encapsulation called generalization.

Generalization in object oriented programming is a way of making different software components act the same. Going back to the homeowner analogy; a microwave, television, and vacuum cleaner are all of one general type; "Things that can be plugged in." Since they all conform to the same general interface, each can be plugged into one standard kind of outlet - buying a new toaster does not require any rewiring. Similarly, the spot, forwards, and options components can all conform to a general trading interface. If each subsystem implements this interface, they can be dropped into the system without adding any special code in the application layer. When the business wants a new feature in the options subsystem, the application layer does not have to change - there is lower coupling.



Returning to the business issue at hand; the current design of the system reflects the mechanical differences between messages, but does not accurately reflect the business view of the world. It is valuable to retain the mechanical division, but the system should also include a layer that more closely models the business.

This dual role of mechanical and business encapsulation is common in software, and there are a number of established solutions for handling it in a safe and cost effective manner. The Strategy Pattern is well suited to this particular case. This design standard is a way of assigning different behavior to a single class or set of classes depending on the context. Adding foreign exchange (FX), equities, rates, and commodities strategies to the design makes it easy to customize the behavior of the spot, forwards, and futures subsystems.



The strategy components enable a single application to behave appropriately for each market area.. Each business unit can, somewhat independently, guide the development of their section of the application. At the same time, a single overall application architecture allows sharing of development and testing expenses.

This is a good starting point for the system. The mechanical components are neatly encapsulated and the system clearly expresses the business view of the world. Each section of the project has a specific role, and is free to react rapidly to changing business needs with minimal risk of error or of negatively impacting another component in the system. The clear divisions also facilitate a wide range of team sizes - the entire system could be built by two or three senior engineers, by fifteen developers loosely organized

according to the architectural boundaries, or even by a number of separate development teams.

Summary

The goal of this exploration was to develop a FIX based FM architecture focused on proper encapsulation. Just such a system has evolved. It contains four key patterns of encapsulation, each subsystem performs a well defined role with a clear purpose, and the architecture is light enough to allow real world business needs to steer the implementation. The architecture is optimized to enable the developers to focus on the business needs in each component with minimal risk of unexpected side effects.

Copyright 2004, Robert Bushman

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-nd/2.0/> or send a letter to Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.