Applying design patterns for web-based derivatives pricing

V. Papakostas, P. Xidonas, D. Askounis & J. Psarras
School of Electrical and Computer Engineering,
National Technical University of Athens, Greece

Abstract

Derivatives pricing models have been widely applied in the financial industry for building software systems for pricing derivative instruments. However, most of the research work on financial derivatives is concentrated on computational models and formulas. There is little guidance for quantitative developers on how to apply these models successfully in order to build robust, efficient and extensible software applications. The present paper proposes an innovative design of a web-based application for real-time financial derivatives pricing, which is entirely based on design patterns, both generic and web-based application specific. Presentation tier, business tier and integration tier patterns are applied. Financial derivatives, underlying instruments and portfolios are modelled. Some of the principal models for evaluating derivatives (Black–Scholes, binomial trees, Monte Carlo simulation) are incorporated. Arbitrage opportunities and portfolio rebalancing requirements are detected in real time with the help of a notification mechanism. The novelty in this paper is that the latest trends in software engineering, such as the development of web-based applications, the adoption of multi-tiered architectures and the use of design patterns, are combined with financial engineering concepts to produce design elements for software applications for derivatives pricing. Although our design best applies to the popular J2EE technology, its flexibility allows many of the principles presented to be adopted by web-based applications implemented with alternative technologies.
Keywords: financial applications, financial derivatives, pricing models, design patterns, J2EE patterns, web-based applications, multi-tiered architectures.

1 Introduction

Financial derivatives have become extremely popular among investors for hedging and speculating. Their growing use has triggered an increased interest in financial engineering and the emergence of several computational models for evaluating them and determining their characteristics.

Numerous software systems and applications have been developed for implementing such models. Some are in-house applications for large financial institutions and investment banks while others are available as software products. Despite the plethora of computational models that are present in the relevant literature, the existence of books and publications on the design and construction of software systems for implementing them is limited. Even these are usually constrained to conventional object-oriented design, circumstantial use of design patterns and traditional programming languages like C++ or Visual Basic.

The objective of the present paper is to propose an innovative design of a web-based application for real-time financial derivatives and portfolios pricing. The modelled application quotes derivatives and underlying assets prices from market data feeds and applies pricing models for computing derivatives and portfolios theoretical values and characteristics. In addition to rendering pricing information on web pages, it can send notifications (e.g. emails) when prices or attributes of derivatives or portfolios satisfy certain conditions (e.g. permit arbitrage or require portfolio rebalancing).

Design patterns play central role in our design, upon which it is almost entirely based. Both generic [5] and web-based applications specific patterns (J2EE patterns [1]) are applied. The proposed design aims to facilitate the introduction of new derivative instruments, additional valuation models and alternative market data feeds to the system on subsequent phases after its initial release.

2 Background work

Joshi [7] and Duffy [3] apply the concepts of object-oriented programming and adopt design patterns for evaluating financial derivatives. London [9] assembles a number of pricing models implemented in the C++ programming language. Zhang and Sternbach [12] model financial derivatives using design patterns. van der Meij *et al* [11] describe the adoption of design patterns in a derivatives pricing application. Marsura [10] presents a complete application for evaluating derivatives and portfolios using objects and design patterns. Eggenschwiler and Birrer [2], and Gamma and Eggenschwiler [4] describe the use of objects and frameworks in financial engineering. Koulisianis *et al* [8] present a web-based application for derivatives pricing implemented with the PHP technology, using the *Problem Solving Environment* methodology.

3 Derivatives pricing models

It is possible to estimate the value that a financial derivative contract should theoretically have from the underlying asset price and the contract characteristics. If the difference between the market price and the theoretical value of the contract is significant, an investor can achieve guaranteed profit (arbitrage). For this reason, derivatives pricing has become the field of extensive study for the past three decades.

A number of pricing models (analytical and numerical methods) have emerged and applied for derivatives pricing [6]. Black–Scholes equation provides analytical formulas for calculating theoretical prices of European call and put options on non-dividend paying stocks. Binomial trees are particularly useful in cases that an option holder has the potential for early exercise. Monte Carlo simulation is primarily applied when the derivative price depends on the history of the underlying asset price or on multiple stochastic variables.

4 Multi-tiered architecture

The present paper proposes the design of an application for derivatives pricing that is web-based. The use of the internet introduces certain complexity into our model. A multi-tiered architecture has been adopted for our design. Each tier in a multi-tiered architecture is responsible for a subset of the system functionality [1]. It is logically separated from its adjacent tiers and loosely connected to them. It is important to emphasise that a multi-tiered architecture is logical and not physical. This means that multiple tiers may be deployed on a single machine or a single tier may be deployed on multiple machines, especially if it contains CPU intensive components.

5 Design patterns

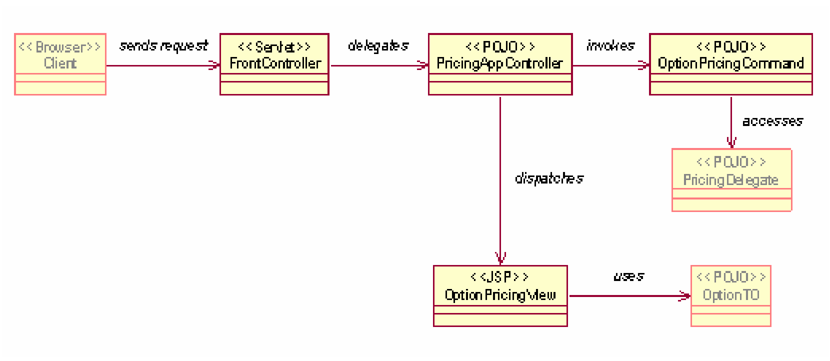
5.1 Presentation tier

5.1.1 Front Controller

The *Font Controller* pattern forms the initial point of communication for handling user requests, aiming to reduce the administration and deployment tasks for the application [1]. One *Front Controller* is used for all user requests. It is incarnated by the *FrontController* class, which is a servlet.

5.1.2 Context Object

The *Context Object* pattern encompasses state in a protocol independent way, in order to be utilized by different parts of the application [1]. One *Context Object* is used for each type of user request. Requests for futures pricing are modelled by the *FuturePricingRequestContext* class, requests for options pricing by the *OptionPricingRequestContext* class, etc. The *Factory* pattern is applied for their creation.



<<Browser>> Client
sends request
<<Servlet>> FrontController
delegates
<<POJO>> PricingAppController
invokes
<<POJO>> OptionPricingCommand
accesses
<<POJO>> PricingDelegate
dispatches
<<JSP>> OptionPricingView
uses
<<POJO>> OptionTO

Figure 1: Presentation tier design patterns.

5.1.3 Application Controller

The *Application Controller* pattern centralizes the invocation of actions for handling requests (action management) and the dispatch of response data to the proper view (view management) [1]. Our design suggests the use of the *ApplicationController* interface for modelling *Application Controller* functionality. The *PricingAppController* class, which implements this interface, coordinates *Commands* and *Views* related to pricing requests. The *ManagementAppController* class does the same for requests related to instrument management, such as adding a new financial instrument to the application. The *Factory* pattern is again applied for their creation.

5.1.4 View Helper

The *View Helper* pattern uses views to encapsulate the code that formats responses to user requests and helpers to encapsulate the logic that views require in order to obtain response data [1]. In our design, *Views* are incarnated by a number of JSP pages. *PortfolioDetailsView* displays information related to portfolios definition, *OptionPricingView* displays the results of an option pricing request, etc. *Business Delegates* are used as *Helpers*.

5.1.5 Command

The *Command* pattern encapsulates the action required as the result of a request through the invocation of the corresponding functionality [5]. One *Command* is used for each type of user request. Requests for futures definition invoke class *FutureDefinitionCommand*, requests for portfolio pricing class *PortfolioPricingCommand*, etc. As a result, there is one-to-one correspondence among *Context Objects* and *Commands*. The *Factory* pattern is applied for their creation.

5.1.6 Factory

The *Factory* pattern is responsible for the creation of objects that implement an interface or extend an abstract class. In our design, a number of classes adopt this pattern, such as *RequestContextFactory* for the creation of *Context Objects*, *ApplicationControllerFactory* for the creation of *Application Controllers*, *CommandFactory* for the creation of *Commands,* etc. *Factories* can be configured declaratively through the use of XML files.

5.1.7 Singleton

The *Singleton* pattern defines classes that are allowed to have only one instance per application [5]. Each class that adopts the *Factory* pattern in our design adopts the *Singleton* pattern in addition.

5.2 Business tier

5.2.1 Business Delegate

The *Business Delegate* pattern encapsulates access to business services, aiming to reduce interconnection between components of the presentation and business tiers [1]. In our design, one *Business Delegate* is defined for each *Session Façade*. The *PricingDelegate* class provides centralised access to the *PricingFacade* class, the *ManagementDelegate* class to the *ManagementFacade* class and the *NotificationDelegate* class to the *NotificationFacade* class.

5.2.2 Service Locator

The *Service Locator* pattern centralises the lookup of services and components [1]. One *Service Locator*, which is incarnated by the *ServiceLocator* class, is used. It also adopts the *Singleton* pattern.

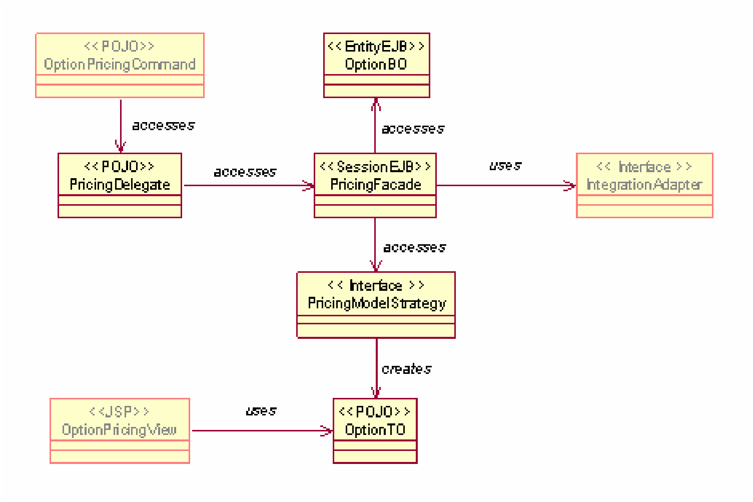


Figure 2: Business tier design patterns.

5.2.3 Session Façade

The *Session Façade* pattern encapsulates components of the business tier and exposes coarse-grained services to remote clients, aiming to reduce the number of remote method invocations among components of the presentation and business tiers [1]. Services related to derivatives and portfolios pricing are aggregated to the *PricingFacade* class. Services related to derivatives and portfolios management are encapsulated in the *ManagementFacade* class. Services for the notification mechanism are accumulated in the *NotificationFacade* class.

5.2.4 Application Service

The *Application Service* pattern underlies components that encapsulate business logic, aiming to leverage related services and objects (*Business Objects*) [1]. Our design adopts the layer strategy in regard to the use of *Application Services*. The *PricingAppService* and *NotificationAppService* classes, which reside on the higher layer, expose pricing and notification services respectively. They require pricing modelling related functionality, which is provided by the *PricingModelStrategy* interface, which resides on the lower layer, along with the *BlackScholesAppService*, *BinomialTreeAppService* and *MonteCarloAppService* classes that implement it. Financial instruments volatility is calculated on a daily basis by the *VolatilityAppService* class.

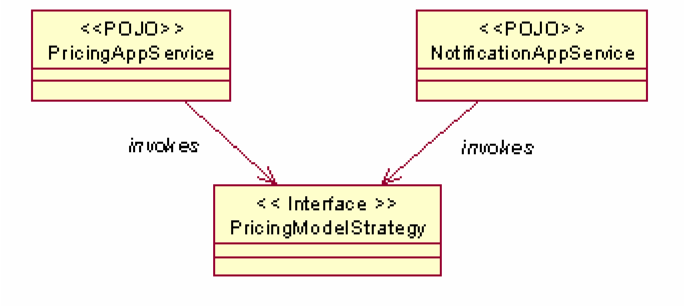


Figure 3: Application Service layering.

5.2.5 Business Object

The *Business Object* pattern encapsulates and administers business data, behaviour and persistence, aiming at the creation of objects with high cohesion [1]. Our design contains a hierarchy of *Business Objects* that correspond to portfolios and financial instruments. They consist of abstract classes *FinancialInstrumentBO*, *DerivativeBO* and concrete classes *PortfolioBO*, *StockBO*, *IndexBO*, *CurrencyBO*, *FutureBO*, *OptionBO*, *EuropeanOptionBO*, and *AmericanOptionBO*. This way, new derivative types can be added with minor modifications.

5.2.6 Composite Entity

The *Composite Entity* pattern aggregates a set of related *Business Objects* into one coarse-grained entity bean, allowing for the implementation of parent objects that manage dependent objects [1]. In our design, the *PortfolioBO* class, which represents a portfolio, is defined as parent object and the *PortfolioInstrument* class, which represents a financial instrument that is member of a portfolio, as dependent object. Although a *PortfolioInstrument* object is linked to a *FinancialInstrumentBO* object, it is a separate object. It holds information such as quantity and (call/put) position of a specific instrument in a portfolio.

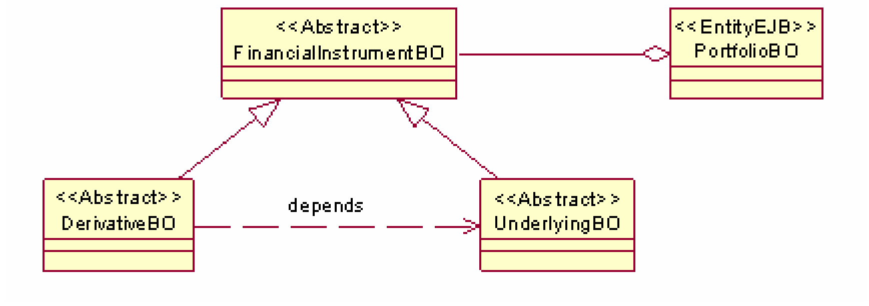


Figure 4: Business Objects for financial instruments.

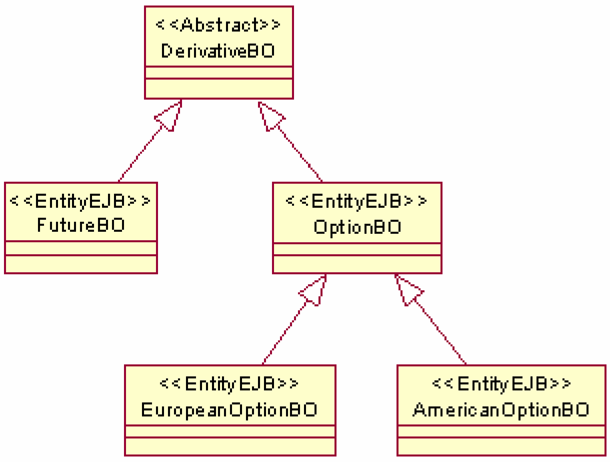


Figure 5: Hierarchy of Business Objects for derivatives.

5.2.7 Transfer Object

The *Transfer Object* (or *Data Transfer Object*) pattern carries multiple data across application tiers [1]. Our design adopts the multiple transfer objects strategy in regard to the use of *Transfer Objects*. One *Transfer Object* is defined for each *Business Object*. This leads to a hierarchy of *Transfer Objects* that correspond to financial instruments and portfolios. They consist of classes

FinancialInstrumentTO, *DerivativeTO*, *PortfolioTO*, *StockTO*, *IndexTO*, *CurrencyTO*, *FutureTO*, *OptionTO*, *EuropeanOptionTO*, and *AmericanOptionTO*.

5.2.8 Strategy

The *Strategy* pattern encapsulates a family of algorithms and makes them interchangeable [5]. Considering our design, such algorithms are the pricing models for derivatives. The *PricingModelStrategy* interface adopts this pattern. It is implemented by the *BlackScholesAppService*, *BinomialTreeAppService* and *MonteCarloAppService* classes, which contain the algorithms for the Black–Scholes, binomial trees and Monte Carlo models respectively. This way, new pricing models can be introduced with minor modifications.

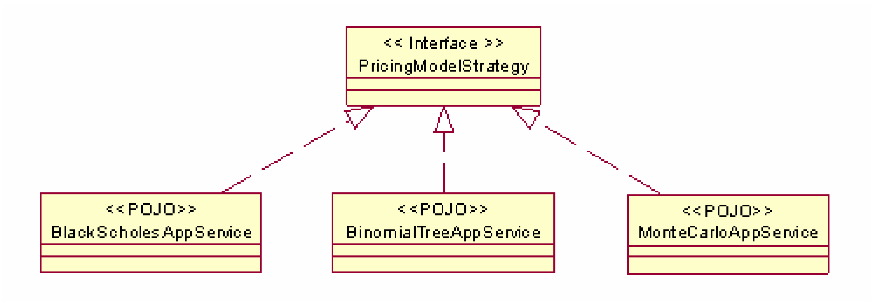


Figure 6: Strategy.

5.2.9 Observer

The *Observer* pattern defines an one-to-many correspondence between an observable object (*Observable* or *Publisher*) and one or more observer objects (*Observers* or *Subscribers*). When the observable object changes state, all the observer objects are automatically notified [5].

The *Observer* pattern is applied on a very significant feature of our proposed design: the notification mechanism. Notifications are sent when the states of derivatives instruments or portfolios conform to certain predefined rules. For example, when the difference between the market and theoretical price of a derivative becomes large enough to permit arbitrage or when the delta of a portfolio in respect to one its underlying instruments exceeds a certain value. In such cases, users should be notified immediately, in order to take advantage of the arbitrage opportunity or perform portfolio rebalancing.

For simplicity, the *NotificationAppService* class is defined as observable object and not each *Business Object* separately. The *NotificationAppService* class is triggered at constant intervals (e.g. every 60 seconds) by a system timer. It monitors derivatives and portfolios states, sending notifications to observer objects. Observer objects implement the *InstrumentListener* interface. The *EmailAppService* and *SocketAppService* classes, which send notifications via email and TCP/IP respectively, have been defined as observers. Additional observers can be easily introduced.

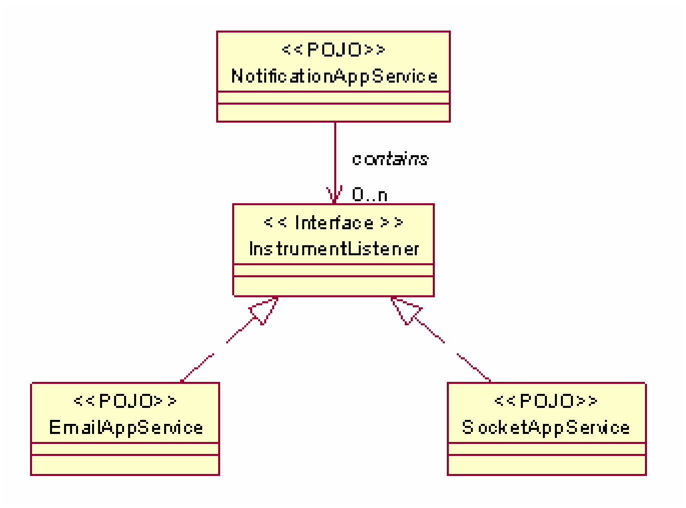


Figure 7: Observer.

5.3 Integration tier

5.3.1 Integration Adapter

The *Adapter* pattern converts the interface of an object or system to another interface that a client is capable of using [5]. The *Integration Adapter* pattern is a special case of the *Adapter* pattern which refers to the integration with third-party systems that perform similar functionality but provide different interfaces, such as market data feeds. In our design, the *IntegrationAdapter* interface adopts this pattern. It is implemented by the *HTMLAdapter*, *XMLAdapter* and *SOAPAdapter* classes, which consume market data available in HTML, XML and SOAP format respectively. These classes may be further sub-classed to allow data consumption from different providers. This way, additional market data feeds may be introduced with minor modifications.

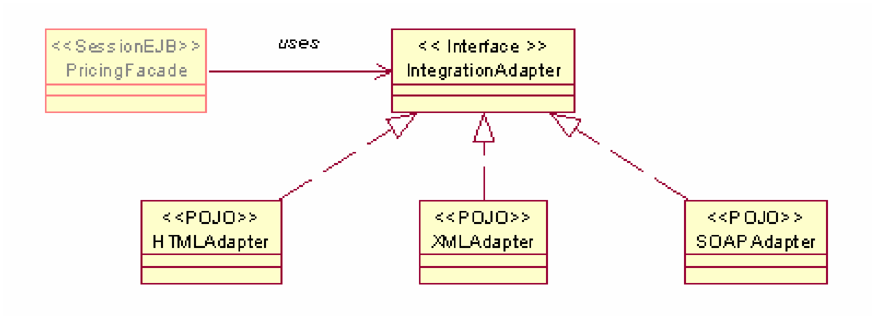


Figure 8: Integration tier design patterns.

6 Conclusions

The present paper aims to combine the theory behind financial derivatives pricing with the latest trends in software engineering, such as the development of web-based applications, the adoption of multi-tiered architectures and the use of design patterns, in order to propose an innovative design of a web-based application for real-time derivatives pricing. Our design is entirely based on the adoption of design patterns, both generic and web-based applications specific, and incorporates some of the principal models for derivatives pricing (Black–Scholes model, binomial methods, Monte Carlo simulation). The introduction of new types of derivatives instruments, additional pricing models and alternative market data feeds is substantially facilitated by our model.

References

[1] Alur, D., Crupi, J., & Malks, D., *Core J2EE Patterns: Best Practices and Design Strategies*, Second Edition, Prentice Hall, 2003.

[2] Birrer, A., & Eggenschwiler, T., Frameworks in the financial engineering domain: an experience report, *Proceedings ECOOP '93*, Springer-Verlag: Berlin, LNCS 707, pp. 21-35, 1993.

[3] Duffy, D., *Financial Instrument Pricing Using C++*, Wiley, 2004.

[4] Eggenschwiler, T., & Gamma, E., ET++SwapsManager: Using object technology in the financial engineering domain, *Proceedings OOPSLA '92*, ACM SIGPLAN, **27(10)**, pp. 166-177, 1992.

[5] Gamma, E., Helm, R., Johnson, R., & Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[6] Hull, J., *Options, Futures and Other Derivatives*, Fifth Edition, Prentice Hall, 2003.

[7] Joshi, M., *C++ Design Patterns and Derivatives Pricing*, Cambridge, 2004.

[8] Koulisianis, M., Tsolis, G., & Papatheodorou, T., A web-based problem solving environment for solution of option pricing problems and comparison of methods, *Proceedings of the International Conference on Computational Science (Part I)*, pp. 673-682, 2002.

[9] London, J., *Modeling Derivatives in C++*, Wiley, 2005

[10] Marsura P., *A Risk Management Framework for Derivative Instruments*, M.Sc. Thesis, University of Illinois, Chicago, 1998.

[11] van der Meij, M., Schouten, D., & Eliëns, A., Design patterns for derivatives software, *ICT Architecture in the BeNeLux*, Amsterdam, 1999.

[12] Zhang, J. Q., & Sternbach, E., Financial software design patterns, *Journal of Object-Oriented Programming*, **8(9)**, pp. 6-12, 1996.