Loan Pricing Model: Design and Implementation

by

Ashish Sharma

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the requirements for the Degrees of

Bachelor of Science in Computer Science and Engineering

and Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 22, 1996

Author _____

Department of Electrical Engineering and Computer Science

May 22, 1996

Certified by _____

Professor Peter Szolovits

Thesis Supervisor

Accepted by _____

F.R. Morgenthaler

Chairman, Department Committee on Graduate Thesis

Loan Pricing Model: Design and Implementation
by
Ashish Sharma

Submitted to the
Department of Electrical Engineering and Computer Science

May 22, 1996

In Partial Fulfillment of the requirements for the Degrees of
Bachelor of Science in Computer Science and Engineering
and Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

As the secondary market for commercial loans is becoming more liquid, these instruments are becoming similar to other financial instruments in that they are traded on the market. With loans now becoming tradable assets, the existence of metrics to measure the risk associated with them and to evaluate fair prices becomes crucial. The loan pricing model quantifies the various risks associated with a loan and generates a market price for the loan. The pricing model is designed as an object-oriented system. Given that the financial algorithms undergo frequent changes, the primary goals of the design are to achieve extensibility and reusability. Other goals of the design include achieving high execution speed, good exception handling and high levels of abstraction. The pricing model is implemented as a client server system, with the analytics running in C++ on the client, and the data residing on a remote SQL server.

Thesis Supervisor: Peter Szolovits
Title: Professor of Computer Science and Engineering

Acknowledgments

I would like to thank Asim Qadir for the critical reading of an earlier version of this thesis and his continued support in the form of suggestions and corrections. I thank Professor Peter Szolovits for the many valuable suggestions he made, and for helping me to give the thesis a coherent structure. I would like to thank my parents, Monica and P.D. Sharma, for their support throughout the past years. Last but not least, I would like to acknowledge my fiancée Anindita Mazumdar, who helped me succeed at M.I.T. by being my biggest support and source of energy.

1. Introduction

1.1. Description of the problem and user requirements

As the secondary market for commercial loans is becoming more liquid, these instruments are becoming similar to other financial instruments in that they are traded on the market. With loans now becoming tradable assets, the existence of metrics to measure the risk associated with them and to evaluate fair prices becomes crucial. Various kinds of risk such as interest rate risk, default risk, and liquidity risk need to be evaluated before a fair price can be determined. In summer 1994, the pricing of loans was a very tedious process. Data had to be manually gathered from different systems and entered into a spreadsheet. The spreadsheet would then price the loans. The whole process was extremely time consuming and error prone. However, the fundamental problem with this implementation was that this system was not extensible and reusable.

The finance industry is highly software based because of the focus it has on information and the means to manipulate information. The industry is very dynamic where new ways of doing things are always being invented and existing methods are always being enhanced. In such an environment it is necessary for the existing systems to be extensible. On the other hand, there are some fundamental financial algorithms that are used in finance systems. Since these algorithms are used widely, they should not be reinvented each time. The proposed solution was to redevelop the pricing model in C++. Most of the problems in the spreadsheet implementation like abstraction, exception handling, execution speed, and expressiveness are relatively easy to achieve in an object-oriented system developed in C++. However , the extensibility and reusability of the system depends on how it is designed. This thesis focuses on both these issues and proposes this design as a model approach for similar financial applications.

The loan pricing model was a component of a new system called the Mark To Market Portfolio Management Application (MTM PMA). This PMA performs basically two functions. Firstly, it generates customized reports for the users. Secondly it allows users to price loans. During my summer internship assignments at Bankers Trust, I have been involved primarily with the design and implementation of the pricing model. Thus, the focus of my thesis will be on the pricing model. The model was implemented as a C++ DLL in Microsoft Visual C++, running on Windows. Chapter 2 describes the system architecture of the model and the factors affecting the choice of we made. Since there was already a solution to the problem, the analysis of the existing solution was an important part of the development process. This is described in Chapter 3.

However, The specific financial algorithms used to price the loans are not discussed in this thesis as they were considered proprietary by Bankers Trust. Chapter 4 describes the different design methodologies, and the factors leading to the choice of Responsibility Driven Design. The design and implementation of the system are illustrated in Chapter 5 and 6 respectively. Extensibility and reusability were the primary factors which were kept in mind during this process. For many components of the system there were several methods of designing it. To choose the one that would be most extensible in the future required us to understand the business problem well. Thus, understanding the finance behind the model was essential to the design process. Finally, various product readiness issues such as exception handling are described in Chapter 7.

1.2. Problems with the current implementation of the loan pricing model

The loan pricing model was initially developed as a spreadsheet. It was developed to evaluate risk exposure due to credit which has been extended to customers and to price the associated financial instruments accordingly. The problems with this approach, which the new system attempts to solve, are:

- *lack of abstraction* - the format of a spreadsheet is monolithic and does not allow for the modeling of abstractions. This makes it harder to factor logic appropriately and obtain insights into the business domain.
- *lack of separation of user interface and behavior* - the user is not able to use the model through an interface that shields him/her from the details of the implementation.
- *lack of separation of data and behavior* - the logic is not modularized separately from the data.
- *poor user interface* - The system cannot price an arbitrary set of facilities or run for different dates.
- *lack of extensibility* - small changes can introduce side effects and minor enhancements can result in significant redesign due to the columnar nature of a spreadsheet. This problem stems from the poor tractability of logic in a spreadsheet.
- *lack of expressiveness* - the basic flow of control of a spreadsheet makes it difficult to express some kinds of computations (i.e. a numerical algorithm that requires relaxation of parameters until convergence is achieved).

- *poor security* - users can modify data and behavior. There is no ability to establish audit trails to keep track of how and when data change. The system cannot implement different levels of security for different users.
- *lack of validation* - spreadsheets cannot validate input as it is entered into the spreadsheet. Invalid inputs show up as errors in the results.
- *poor exception handling* - the user is not notified when errors occur. Since it is hard to handle every dependency, invalid results are often generated and not noticed unless the user manually verifies all outputs. Multiple severities of errors are also not supported.
- *poor execution speed* - the model takes days to price a portfolio!
- *heavy reliance on manual entry of input* - this process is prone to many errors and is time consuming.

2. System architecture

2.1. Description of the platform used

The user interface of the PMA is implemented using Microsoft Access. The loan pricing model is implemented as a C++ dynamically linked library (DLL). The database is a Sybase SQL server located on a remote server. LS2 is the primary loan system of the bank. Along with the database server, LS2 provides an interface to access and analyze loan data. The data in the Sybase database is populated from LS2 and other systems. The access interface implements the reporting requirements of the users and allows the users to price the loans as well. The C++ DLL communicates with the database directly for the inputs it needs to price a loan. This is shown in figure 1. Stored procedures stored on the database server serve as the interface between the C++ DLL and the database. The network protocol used for communicating between the client and the database server is TCP/IP and the API used for the database communication is Microsoft ODBC. This is shown in figure 2.

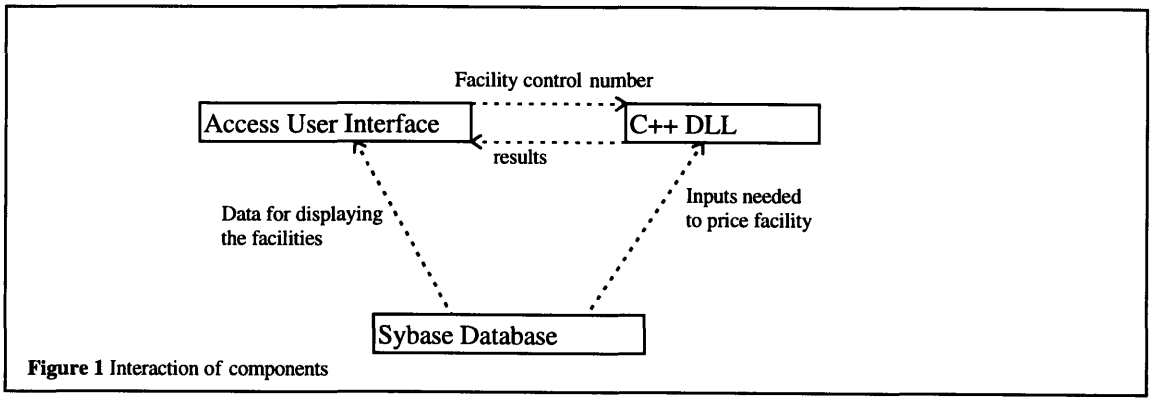


Figure 1 Interaction of components

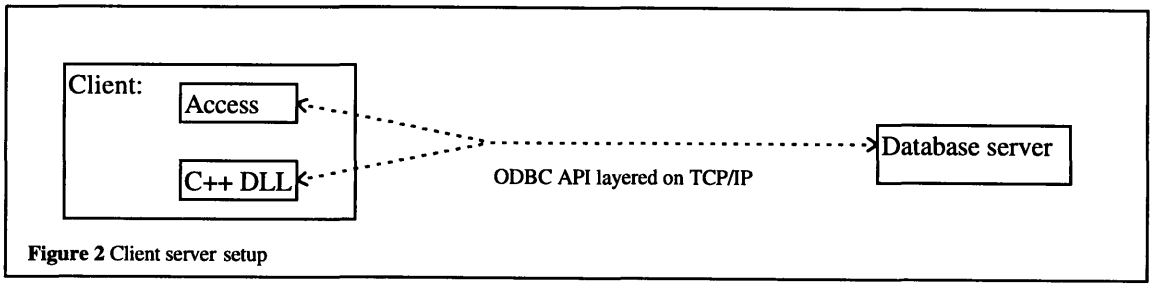


Figure 2 Client server setup

2.2. Definition of a PMA and the platforms used for it

Applications for the business domain at Bankers Trust can be broadly divided into two categories - product control applications (PCAs) and portfolio management applications (PMAs). The product control applications are mostly large repositories for the official records of the bank. They run on secured servers and the access to them is limited. PMAs on the other hand, are applications made to run on the PC. They provide the end users the ability to effectively use the data from the PCA. In a sense, the PCAs are the servers and the PMAs are the clients.

In addition to the differences in the software and the platform the standards for PCAs and PMAs are very different. PCAs are generally secured databases running Sybase on a RISC processor. Analytic PCAs are fewer in number and are typically implemented in C running on a Unix server. The operating system for the PCAs is either Unix or Windows NT, although some earlier PCAs have been implemented on the VAX mainframes. PMAs are developed for all the other applications that the users need. Most analytic PMAs are written in spreadsheets using Microsoft Excel, and running under Microsoft Windows. Visual Basic and Microsoft Access are also widely used. C++ is used for very few PMAs today. Among other things, PCAs implement strict security and the data in these systems is official. PMAs are not as secure and the results generated by the PMAs are used primarily by the few people who use it.

The Mark to Market PMA tries to narrow the gap between the traditional PCAs and PMAs. It is developed with the design concepts of a PCA, although it meets the requirements of PMA users. Specifically, the system is designed as a client server system with the goal of making it extensible and reusable, which makes it similar to a PCA. This PMA is also shared by more users than other PMAs are. The security of the system is not implemented to the level of a PCA but the system may be extended to provide a higher level of security. Therefore the MTM PMA is different from both the PCAs and the PMAs currently developed in the bank.

2.3. Bankers Trust Architecture

Technology departments at different banks are very different. Some banks have a central technology department and some have a technology department for each business area. For the banks which have separate technology departments associated with each business line, often the software developed is not shared by the different departments. Bankers Trust has different technology departments for each business line. However there is a department called Technology and Strategic Planning (TSP) which defines technology guidelines for the whole bank. TSP defines this in the Bankers Trust Architecture (BTA). In the context of what software to use, BTA defines

a cost effective technology base as its objective. The architecture principles are to use a minimum set of infrastructure components and to off- load integration to strategic vendors. The uniform computing model from BTA 2.0 [1] is shown in figure 3. In addition to this, BTA specifies a list of approved products in the form of templates [1]. All technology groups at Bankers Trust are expected to use the software which has been approved in the BTA. Microsoft is an approved vendor in the BTA and the programming language of choice for PMAs is C++.

There are pros and cons of having to choose from a selected list of products. Off loading integration to a strategic vendor ensures the compatibility between the different components of the system. For example, the interface between Microsoft Access and Microsoft Excel is well defined since they are both products made by Microsoft. Using the same software applications in every technology department ensures compatibility between applications and facilitates global function/class libraries for all the departments. Often this also makes the system easier to maintain and support. The firm also enjoys the benefits of economies of scale in the costs of licenses, support etc. However, the big disadvantage of using standard software is that it is not always possible to choose the application most suitable for the project. This has affected our choice of platform in a major way.

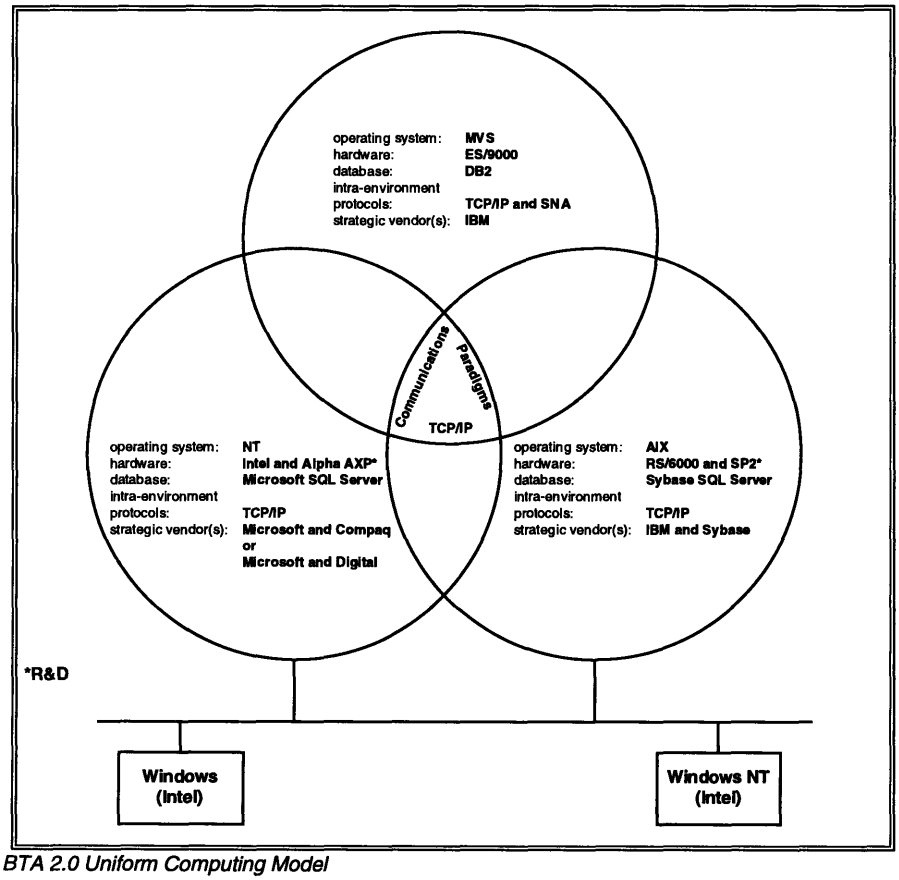


Figure 3 *BTA 2.0 Uniform Computing Model*

2.4. Factors affecting the choice of the system architecture

The loan pricing model was implemented using object oriented methodology, the reasons for which are discussed in the next chapter. For this methodology the two commercially popular languages are Smalltalk and C++. From an object oriented perspective Smalltalk is a superior language as it is a non typed language where the building blocks are objects. C++ on the other hand is not fully object oriented. It is an extension of C with the functionality of object oriented programming being a subsequent addition. The major advantage of C++ over Smalltalk is that programs written in C++ generally run faster than the corresponding ones written in Smalltalk. Given the numerically intensive nature of the loan pricing model, C++ was chosen as the language for development.

Given C++ as the language for programming the model, the two major commercial vendors of C++ compilers (under OS/2 and MS Windows) are Microsoft and Borland. The development environment of Borland C++ was superior to that of Microsoft Visual C++. Borland C++ was more stable and their class libraries provided more functionality. However Microsoft was a BTA approved vendor and BTA was in favor of Microsoft Visual C++. Also, from the point of view of integration, it was preferable to use Microsoft Visual C++, since we were using Microsoft products for other parts of the PMA as well.

The model was implemented as a DLL because one could foresee it being used by many systems in the future. In addition to the pricing model, the PMA also had reporting requirements, for which other applications would be more suitable. Therefore the user interface was not developed as part of the model. We used Microsoft Access for the user interface, although the BTA recommended solution was Microsoft Visual Basic. This decision was made primarily due to the reporting requirements of the PMA. Microsoft Access is a database product which provides many functions to generate automated reports. Since there was a lot of database I/O involved, Microsoft Access was chosen. In retrospect, it was probably not a very good decision because of the way Access interfaces with other types of databases. When Access edits a view on a table from another database, one works directly on the table. There is no way to catch values in intermediate stages and manipulate them. This turned out to be a major drawback of the user interface, a drawback which would not have existed if Visual Basic was used as the tool to develop the user interface.

Since we were using Microsoft Visual C++ and Microsoft Access , there was really no choice other than Microsoft Windows for the operating system. The only other operating system used on client workstations at Bankers Trust is OS/2. The advantage of OS/2 is that is a much more stable operating system with multi threading capabilities. The disadvantages of OS/2 were that firstly there were no comparable tools for developing the user interface under OS/2. Secondly, the users had a distinct preference for Windows, as they were already using Windows. Since OS/2 has the capability of running Windows code under WIN/OS2, developing the system for Windows would allow both sets of users to use the model. The third factor in favor of Windows was that the Bankers Trust plans to use Windows NT as the operating system for its servers. In the future, it is probable that the loan pricing model is installed on a RISC server from where many users will be able to access it. From this perspective it was preferable to use Microsoft Visual C++ running under Windows.

3. Analysis of the existing solution

Since a spreadsheet version of the pricing model was already in use, the problem needed to be approached in a non-traditional manner. The various components of the existing solution and its usage had to be analyzed. The issues can be broadly categorized as follows

- *Validation of the object hierarchy:* This process involved working with the primary developer of the pricing model to validate the abstractions and interactions made in our object model of the solution. This validation was mostly from a business perspective to ensure that our objects modeled the business as closely as possible. This was done in an incremental manner where we started by discussing a preliminary version of the model (the final object hierarchy is on page 16). For example, we verified if separate classes for interest payments, fee payments and amortization payments were necessary. In this case it was necessary, primarily because these payments may occur at different frequencies.
- *Analyzing the data requirements:* The data for the pricing model came from various sources. Since this process needed to be automated, the sources needed to be defined precisely and automated feeds into our database had to be arranged for as many of the sources as possible. At the time the model was being developed most of the data came from LS2. However, some data came from other systems of the bank and some data came from external vendors like Bloomberg. Of these systems some of them were scheduled to be replaced by LS2 in the near future. For these systems it was not worth arranging for automated feeds as LS2 would provide the data in the near future. The data from external vendors, an example of which is the data on the yield curve, needed to be automatically downloaded from the external source.
- *The team structure:* In addition to the developers of the system there were three groups of people actively involved in the development of the PMA. The PMA aimed to be an official application of the bank , i.e. the values generated by the process were to be official bank records once they were copied into the loan system database. This required that the central controllers of the bank be involved to monitor the security issues of the system. The other two groups were users of the pricing model and users of the reporting component of the PMA. If the system was developed in isolation from the users then the interface may or may not appeal to them. Thus, users input during the development process is important.

4. Design methodology used

4.1. Reasons for Choosing object oriented methodology

The loan pricing model is a model which is evolving continuously. The reasons for choosing an object approach to the problem, some of which are mentioned in Object Oriented Analysis and Design [2] are

- *Modeling of the underlying financial concepts* - the designer thinks in term of behavior of objects, not low level detail. In a model where the concepts and their interactions are well defined, this methodology is preferable to the procedural approach. Encapsulation hides the detail and makes complex classes easier to use. Once implemented, the class is like a black box where one has to understand the behavior of the class and how to communicate with it.
- *Maintainability* - Loan pricing will undergo significant changes in the future. This creates a maintainability problem. However if the behavior of the underlying concepts is correctly captured in the objects, then it is a much easier task.
- *Reusability* - the loan pricing model implements some fundamental financial concepts like yield curve calculators. Yield curves are used widely for the purpose of calculating the present value of future cash flows. This is done by discounting the future cash flow by an interest rate which is calculated off the appropriate yield curve. The pricing model also implements some basic pricing techniques like binomial trees and some related abstractions like payment streams. The binomial model , which is described in detail in section 5.1.1, is a discrete time representation of a Markov process and is used to price assets whose past state is not relevant in predicting its future price. These classes would be useful for any similar application written in the future.
- *Evolutionary design* - object oriented methodology provides the flexibility to extend the design incrementally. This is important given that we could not understand the model completely upfront.
- *Faster Design* - Applications developed in an object oriented environment are created from preexisting components, which in this case is the class library provided [2]. For example, the class library provided the fundamental time structures and list structures which only needed modification to build payment streams.
- *More realistic modeling* - object oriented analysis models the enterprise or application area in a way that is closer to reality than conventional analysis. The analysis translates directly into design and implementation. In conventional techniques, the paradigm changes as we go from

analysis to design and from design to programming. With object oriented techniques, analysis, design and implementation use the same paradigm and successively refine it [2].

4.2. Reasons for Choosing Responsibility Driven Design (RDD)

We chose Responsibility Driven Design (RDD) as the design methodology for our system. The other alternatives we considered were Booch's Object Oriented Design methodology, the Use Case method proposed by Jacobson, and the Fusion Method. Following is a short description of these methods and the reasons for choosing RDD over the other methodologies.

Responsibility Driven Design (previously called the Class-Responsibility-Collaboration - CRC methodology) designs the application from the perspective of what the responsibilities of the objects are. In Designing Object-Oriented Software [3] responsibilities are stated to include two key items -

- The knowledge an object maintains
- The actions an object can perform

Collaborations represent requests from a client to a server in fulfillment of a client responsibility. An object can fulfill a particular responsibility by itself, or it may require the assistance of other objects. RDD, as the name suggests, designs the application keeping in mind what the responsibilities of the classes are and what other objects the class has to collaborate with to fulfill all its responsibilities. This methodology is the simplest of the object oriented methodologies and can be easily implemented with a minimum of tools. The only thing needed for the design process are CRC cards, which are diagrams which capture the methods of the classes and the interaction between classes.

Another design methodology is presented by Grady Booch in Object Oriented Design [4]. This methodology, which he calls simply object oriented design (OOD), is captured by four basic diagrams -

- *Class diagrams* are used to show the existence of classes and their relationships in the logical design of a system; a class diagram represents all or part of a class structure.
- *Object diagrams* are used to show the existence of objects and their relationships in the logical design of the system. A single object diagram represents a snapshot in time of an otherwise transitory event or configuration. An object in an object diagram denotes some instance of a class.

- *Module diagrams* are used to show the allocation of classes and objects to modules in the physical design of the system.
- *Process diagrams* are used to show the allocation of processes or processors in the physical design of the system. These diagrams are important for systems that execute processes asynchronously.
- *State transition* diagrams are used to show the state space of an instance of a given class, the events that cause a transition from one state to another, and the actions that result from that change.
- *Timing diagrams* are used to show the dynamic interactions among various objects in an object diagram.

The process that Booch describes for designing a system consists of the following three steps that may be iterated over many times.

1. The first step in the process of OOD involves the identification of the classes and objects at a given level of abstraction; here, the important activities are the discovery of the key abstractions and the invention of important mechanisms.
2. The second step involves the identification of semantics of these classes and objects; here the developer must view each class from the perspective of its interface.
3. The third step involves the identification of the relationships among these classes and objects; here we establish how the objects interact with the system.

RDD and the methodology presented by Booch have a lot of overlapping ideas. However the methodology described by Booch (OOD) is much more detailed and rigorous. This makes it more time consuming than RDD. OOD requires that one understand the problem fully before one designs the system. In other words, OOD is more systematic, which is evident from the fact that object oriented analysis is mentioned as an ideal front end to the design process. These factors make OOD effective in designing very large systems. RDD on the other hand is more effective in allowing the designer to incrementally add to the existing model. Therefore, RDD is better suited for an evolutionary design. Here the design evolves along the way, object oriented analysis and object oriented design taking place together. Given the time constraints of the project it was necessary to design the system as fast as possible. As the project was a medium sized one, with very few people on the development team, we chose RDD over the second approach, even though it is less systematic.

The third methodology considered was the one described in Object Oriented Software Engineering : A used case approach [5]. This methodology is often referred to as Objectory. The framework of Objectory is derived from a framework called "design with building blocks". The system is viewed as a number of connected blocks, each block representing a system service. Once the blocks have been specified, they are designed using a top-down approach keeping in mind the criteria of ensuring that the system being developed can support changes to its functionality and can be adapted to new technology. The underlying architecture of this design methodology is comprised of four different models - the requirements model, the analysis model, the design model, and the implementation model.

The requirements model consists of actors and use cases supported with a domain object model and interface descriptions. Actors model the prospective users who will interact with the system and a use case specifies a flow that a specific actor invokes in a system. An actor is a user type or category and hence one person can play the role of many actors. The description of use cases is further enhanced using interface descriptions called MMIs (man-machine interactions). Thereafter the object model is developed which gives an easy to understand picture of the system. This method of approaching the problem emphasizes greatly on the outside interactions of the system. The analysis model is developed from the requirements model. The aim is to get a robust and logical structure that will be maintainable during the system life cycle. The functionality is modeled using objects which are categorized as interface objects, entity objects and control objects. The design model refines the analysis model further and also takes into consideration the current implementation environment. This is followed by the implementation phase which consists mainly of the source code written to implement the blocks.

Objectory approaches the problem from a different angle than that used by RDD and OOD. This use case approach focuses more on the external interactions of the system. In our system the internal interactions between the objects are much more complex than the external interactions. The user interface is simple and well defined. Objectory is a useful methodology for designing systems with highly complex user interactions and user interfaces. For this reason, we felt that the RDD/OOD approach was more suitable for our project.

The last methodology considered was the fusion method, described in Object Oriented Development, The Fusion Method [6]. This method attempts to integrate and extend the best features of the most successful object oriented technologies : OMT\Rumbaugh, Booch, CRC and

Objectory. Two problems not addressed by the above methodologies that the fusion method talks about are -

- Team working not addressed by most methodologies
- Management changes are required since the object oriented approach is fundamentally different from the functional decomposition approach.

The fusion method attempts to address these problems by precisely dividing the process of system development into phases and indicate what should be done in each phase. It also provides criteria that tell the developer when to move on to the next phase. Secondly it provides management tools for software development. The outputs of different phases are clearly identified and there are cross checks to ensure consistency within and between phases. The three phases are analysis, design and implementation. The phases are not described in detail here as the components are similar to the ones used in the methodologies mentioned above. The difference lies in the fact that the place of each component is precisely defined.

The fusion method adds value in the management domain of the system and for the coordination of teams on large projects. These additions were not very applicable in our case since there were only two members developing the loan pricing model. Therefore there were no outstanding management issues that needed to be addressed. The methodology had much more detail than we needed for the current project size. Because of these reasons, RDD was chosen over the fusion method.

5. Design

5.1. Application architecture components

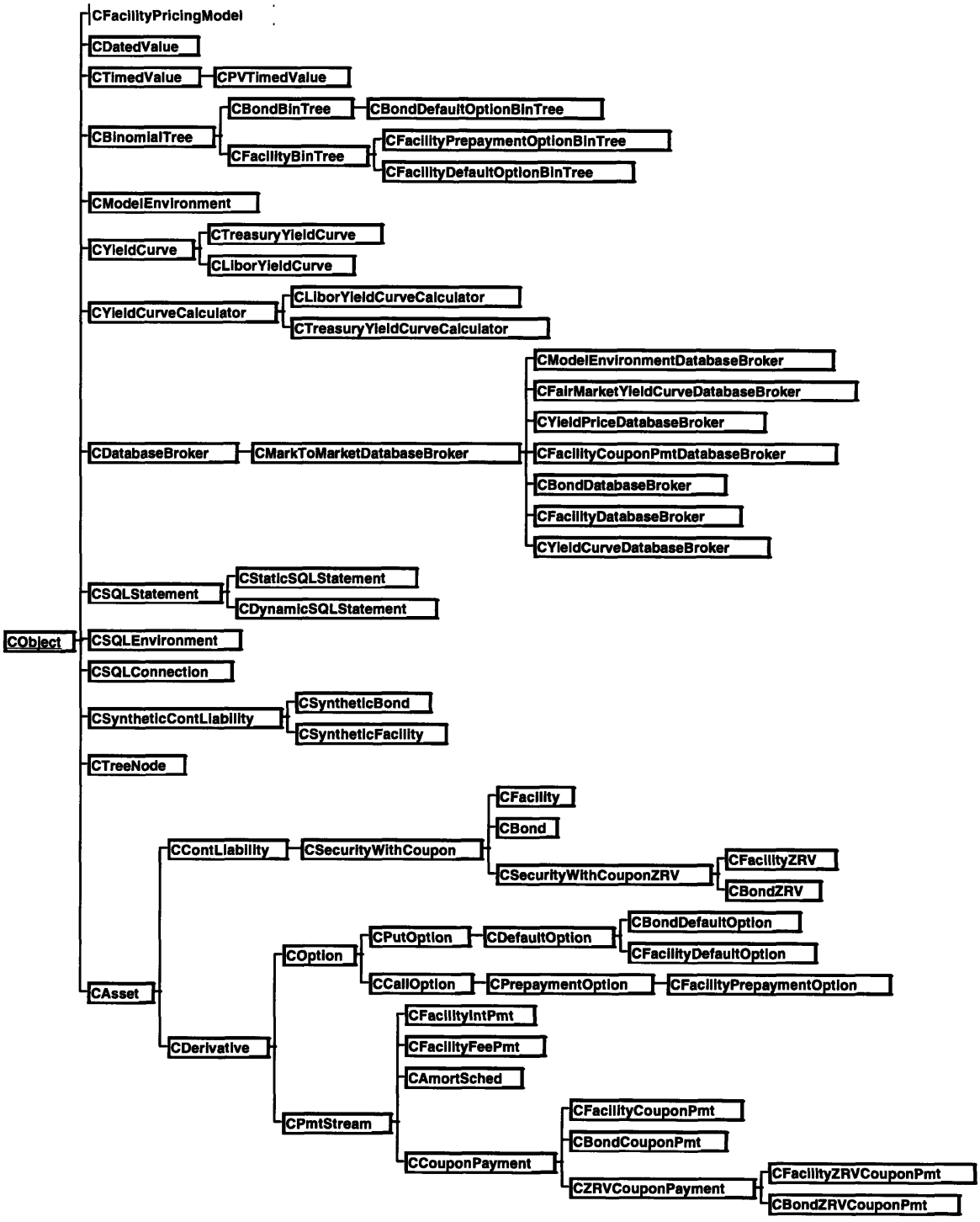
The application architecture components can be divided into three major categories - the analytics (C++ DLL which prices the loan), the database and the user interface.

5.1.1. Analytics

The analytics is the most extensive part of the loan pricing model and is implemented in C++. The class hierarchy is shown in figure 4 on the next page. The classes can be divided into three major categories -

- *Classes which are business domain independent classes* - These classes implement general structures which help the business specific classes price the loan e.g. time structures and the model environment.
- *Classes which are finance related but independent of loan pricing* - These classes implement finance related processes like binomial trees (described in detail later in the section). They also implement generic assets such as options, payment streams, and bonds
- *Classes which are specific to loan pricing*- these classes contain information and algorithms which are specific to the pricing model. They are the synthetic contingent liability classes. A contingent liability is an asset that shows up as a liability on a portfolio. The assets pertinent to the pricing algorithms were separated into two categories - the asset and the synthetic asset. These categories separate the generic properties of the asset from the specific details of the pricing model.

This design of the model leverages reusability as much as possible. The above categories describe three tiers of reusability. Classes in the first category can be used by any application. Classes in the second category can be used by any application attempting to solve a problem in the area of finance. The classes in the third category are specific to the problem and are useful for loan pricing only. We tried to minimize the amount of code in the third category. The design attempts to make as much of the code reusable as possible. For example, a bond was separated into two classes - CBond and CSyntheticBond. The CBond class implements methods which are generic to all bonds and would be meaningful to any application using bonds. In contrast, the CSyntheticBond class contains the specific algorithms that we use to price certain bond related values. Hence the generic bond characteristics are separated and reusable.


CFacilityPricingModel
CDatedValue
CTimedValue
CPVTimedValue
CBinomialTree
CBondBinTree
CBondDefaultOptionBinTree
CFacilityBinTree
CFacilityPrepaymentOptionBinTree
CFacilityDefaultOptionBinTree
CModelEnvironment
CYieldCurve
CTreasuryYieldCurve
CLiborYieldCurve
CYieldCurveCalculator
CLiborYieldCurveCalculator
CTreasuryYieldCurveCalculator
CDatabaseBroker
CMarkToMarketDatabaseBroker
CModelEnvironmentDatabaseBroker
CFairMarketYieldCurveDatabaseBroker
CYieldPriceDatabaseBroker
CFacilityCouponPmtDatabaseBroker
CBondDatabaseBroker
CFacilityDatabaseBroker
CYieldCurveDatabaseBroker
CSQLStatement
CStaticSQLStatement
CDynamicSQLStatement
CObject
CSQLEnvironment
CSQLConnection
CSyntheticContLiability
CSyntheticBond
CSyntheticFacility
CTreeNode
CFacility
CBond
CContLiability
CSecurityWithCoupon
CSecurityWithCouponZRV
CFacilityZRV
CBondZRV
CBondDefaultOption
CPutOption
CDefaultOption
COption
CFacilityDefaultOption
CCallOption
CPrepaymentOption
CFacilityPrepaymentOption
CAsset
CFacilityIntPmt
CFacilityFeePmt
CDerivative
CAmortSched
CFacilityCouponPmt
CPmtStream
CBondCouponPmt
CCouponPayment
CZRVCouponPayment
CFacilityZRVCouponPmt
CBondZRVCouponPmt

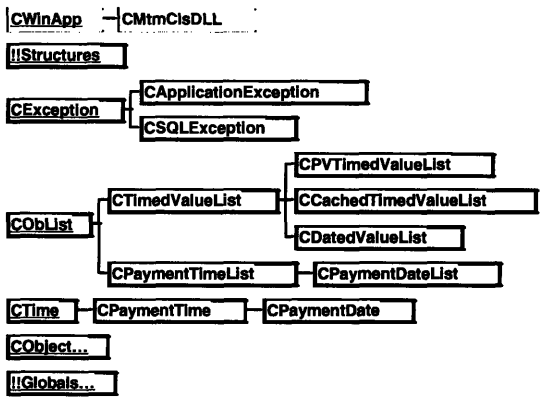


Figure 4: Class hierarchy

The Business Domain Independent Classes

The helper classes are divided into the following groups of classes

Time structures: The loan pricing model requires extensive calculations using times and dates. Methods are implemented to allow the programmer to manipulate times in the CPaymentTime class. In many cases, times needed to be stored only in days and not in hours, minutes and seconds. The CPaymentDate class provides for that. The CTimedValue class associates a value with a time object , a class which is useful for the building block of a stream of payments. The CPVTimedValue class is a type of CTimedValue class with the difference that it stores the payment time, value as well as an associated discounted payment value. This is calculated given the interest rate and the time expired since the business run date. This structure is used widely since many payments are discounted to present value.

List structures: These classes implement data structures which assist in modeling data streams. The various classes (Subclasses of CObList in the above figure) arise due to the fact that C++ is a typed language and has to keep track of the types of the elements in a given list. Payment streams are used to represent cash flows associated with financial assets and to represent interest rate curves. The timed value list classes implement lists whose units are CTimedValue or CPVTimedValue objects. The lists also provide the functionality for interpolating values between elements of the list. The payment date and payment time lists simply store a stream of times or dates. These classes are not as frequently used as the timed value list class.

Database brokers and SQL classes: The interface to the database is abstracted in the database broker classes. Each business class needing input from the database has a database broker class attached to it. Only via the database broker can the asset get the database inputs it needs. This design decision was made to make the database as independent from the pricing process as possible. In the future if the database interface changes then the only things that will need change are the database broker classes and the SQL classes.

The SQL classes abstract ODBC and serve as a wrapper for the ODBC concepts used to interface with the database. These classes include the CSQLEnvironment, CSQLConnection and CSQLStatement classes. All the code for setting up the environment, connection and executing queries lies in these classes. This application makes the implementation of the other classes independent of the specific platform. For example, if we were to move the database from Sybase to Oracle, then the pricing portion of the program will be unaltered. Only the database brokers and the SQL classes will need modification in order to adapt to the new platform.

Model environment: This class encapsulates all the constants used in the model. There are no hard coded values in the model. All the values are read from the database via the model environment database broker. Examples of some of the global parameters are the maximum number of iterations allowed for a converging process, the depth of the binomial trees etc. The model environment class also provides access to globally used structures like the yield curve and the SQL environment.

Finance Related Classes Independent of Loan Pricing

Binomial trees: A Markov process is a particular type of stochastic process where only the present state of the process is relevant for predicting the future. The past history of the process and the way in which the present has emerged from the past are irrelevant. Models of stock behavior are usually expressed in terms of what are known as *Wiener processes* [7]. A Wiener process is a particular type of Markov stochastic process. This process has also been used to describe the brownian motion of a particle.

The binomial model is a discrete time representation of the above model. If S is the initial price of the stock, after an interval dt it moves up to Su with the probability p and down to Sd with the probability 1 - p. After another time interval three alternatives are generated. The probability p is assumed constant through every time period as all relevant information about the past is captured in the current price of the asset. It can be shown that in the limit $dt \rightarrow 0$, this binomial

model becomes the geometric brownian model. In the binomial model we assume a single price movement (up or down) in dt. The binomial model can also be viewed as a numerical approximation to the popular Black-Scholes pricing model. The binomial model approaches the Black-Scholes model as $dt \rightarrow 0$ and the depth of the tree$\rightarrow \infty$. However, the Black-Scholes model assumes a constant risk free interest rate and constant volatility of the asset. The binomial model can use different interest rates for different time periods. However, statictically it is observed that the Black-Scholes model generates accurate results using the average interest rate over the whole time span. The binomial model is shown in figure 5.

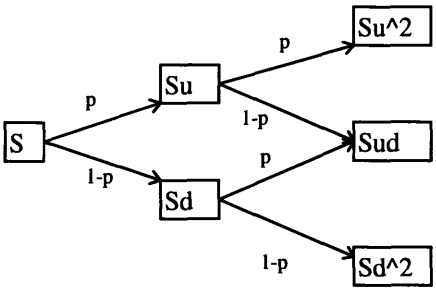


Figure 5 prices movements over two time periods using the binomial model

Given the prices at the last level of the tree it is possible to backtrack and calculate the present value of the asset. In the loan pricing model this technique is used to price the options associated with the loan. The CTreeNode class represents a node on the binomial tree and the CBinomialTree class implements the stochastic process. The CBinomialTree class implements the general binomial tree algorithm. The difference in behavior between the specific options is coded in the respective subclasses of the CBinomialTree class (e.g. CBondBinTree, CFacilityBinTree etc.). The option objects contain the appropriate binomial tree object to which they delegate the stochastic part of the calculation. This design provides encapsulation and reusability for the binomial tree.

Yield curve and yield curve calculators: The yield curve calculator calculates discount factors and other related values from the yield curve. These values are required to obtain the present value of a future payment and are obtained from methods of the CYieldCurveCalculator classes. The actual yield curves are stored separately in the CYieldCurve classes. This was done because the calculations made from the yield curve are not an inherent property of the yield curve and should

therefore be separated from it. In other words we separated data and behavior. As a result the yield curve class, which represents the data aspect of the curve, can be used in different ways by other classes if necessary. The list structures described in the previous section were used to encapsulate the yield curve.

Assets: The class hierarchy of the asset classes is given in figure 4. A broad overview of the main classes is as follows

- Facility and Bond classes: These classes model the loan and the bond in the system (a facility behaves similarly to a loan). They are a subclass of *contingent liabilities* as they show up as liabilities on a portfolio. The fee for holding the risk associated with these liabilities is in the form of coupon payments. Hence these classes are also subclasses of the CSecurityWithCoupon class. CFacility and CBond basically contain the data associated with the respective assets. They store the payments as lists, the spread on the interest rate associated with the assets as values etc. The facility ZRV[1] and the bond ZRV store calculations derived for the ZRV.
- Options: Call options and put options are *derivatives* in that they are financial instruments whose value is based on another security. An option can derive its value from an underlying stock, stock index or future. A call option gives the holder the right to buy the underlying asset by a certain date for a certain price[7]. Conversely, a put option gives the holder the right to sell the underlying asset by a certain date for a certain price. The classes are responsible for pricing the option although a significant portion of the process is delegated to the binomial tree classes.
- Payment Streams: These classes model payments associated with any asset. Hence a payment stream is a *derivative* by the definition given above, as it is the payment stream of some asset. In our model, the payment streams are associated with either the facility or the bond. The names of the subclasses of CPaymentStream are suggestive of what payment stream each one is encapsulating.

Classes Specific to Loan Pricing

Synthetic contingent liability: The goal of the PMA is to generate a price as well as other related results like the dollar value of a one basis point change (a basis point is a hundredth of a percent = 0.01%). All the results required from the PMA are abstracted in the CSyntheticFacility

[1] Zero recovery value. A zero recovery value facility is a facility which cannot be prepaid or defaulted on.

and the CSyntheticBond classes. As mentioned earlier, the loan pricing specific sections were separated to facilitate reuse of the facility and bond classes.

5.1.2. Database

The database for the PMA inherited much of its structure from LS2. It is a relational database where the tables are mostly normalized. There is a table defined for each business entity. There exist dataless keys which are used for joining between tables. An example of this is an ID given to each facility, called a facility ID. This key has no data associated with it and is used for uniquely identifying a facility (a facility is a like a loan). This is done so that the identification of the loan is not dependent on the data associated with it. Every facility also has a facility control number, a number which is the unique external key for a facility. The facility control number is used to identify the facility from outside LS2. For internal joins between tables, the facility ID is used as the key. The distinction between the facility ID and the facility control number is made because if the external identifier of the loan is changed, all the internal tables do not have to be updated. Only the mapping between the facility control number and the facility ID need to be updated.

Security is an important issue for the database. A typical user should not be able to see all the columns of the tables. Therefore the database communicates with the C++ DLL only through stored procedures. Authorizations are given to users to run stored procedures only. It is easier to manage security this way since there are only a few nodes of communication. For example if table T contains the data for traders A and B then giving the traders read access to T would give both of them access to each others portfolio. Separate stored procedures are written for A and B, where A's stored procedures explicitly checks if the data requested is in A's portfolio. Now A is given permissions to run his/her own stored procedures only and is not given read access on the table T. Therefore A cannot see B's position. A similar strategy is used for B. Moreover, stored procedures exist on the server in a secure environment, which makes the implementation of authorizations easier. Another use of stored procedures for retrieving data is that the C++ DLL can run the stored procedures without having to worry about the table layouts. The logic of doing several joins, returning of defaults, etc., is encapsulated in the database. A simple example of this is when one tries to get the estimated selling time from the facility table. The stored procedure first checks to see if it is NULL. If it is, the procedure looks up the mobility rating for the facility in the facility table and using that rating, looks up the estimated selling time in the mobility table. This allows the pricing model to abstract the specific table layout of the database. If the database

and/or the table layouts are changed, the C++ code is not affected. The new database will have to perform the joins to retrieve the correct inputs.

The database could be accessed using static or dynamic SQL statements. If the DLL uses static SQL statements, the SQL command is stored on the database as a stored procedure when the statement is prepared. When the statement is executed this causes the procedure to be executed on the database server. Dynamic SQL statements are not prepared and are executed on the server whenever the DLL requests the database for results. Static SQL statements incur an overhead due to the creation and destruction of the stored procedure associated with the SQL statement. However, if a single statement is executed many times, this is more efficient as the database performs various optimizations and pre compilations that allow it to retrieve the results faster. There is a disadvantage to using static SQL statements. If the communication between the client and the database is prematurely terminated then there are stored procedures that are left on the database that need to be cleaned up. There was no significant pre compilation benefits of the statements executed in the C++ code, since the only statements we had were executing stored procedures on the database. Therefore we used dynamic SQL statements to access the database from the DLL. The database keeps track of who changed what in the database. Every table has a create user id, a update user id, a create time stamp and an update time stamp field. This is another security measure in a multi user environment. The database currently does not have the functionality to maintain audit trails. Audit trails will probably be featured in the next release.

5.1.3. User Interface

The user interface is implemented in Microsoft Access and allows the user to price one or many facilities at a time. The user interface generates automated reports, which was another requirement of the PMA. It also serves as an interface for users to enter and update some data in the database. There is some security implemented at this level that does not allow users to enter invalid data into the fields of the database. The C++ DLL communicates with the user interface through a well defined protocol. In this thesis I will talk of the user interface only in context of the pricing requirement of the PMA, even though the user interface implemented other requirements of the PMA. There were meetings held with the developers of the user interface to decide the communication protocol between the DLL and the user interface and the delegation of tasks between the two. I was not directly involved with the design or the implementation of the reporting aspect of the user interface.

5.2. Application usage

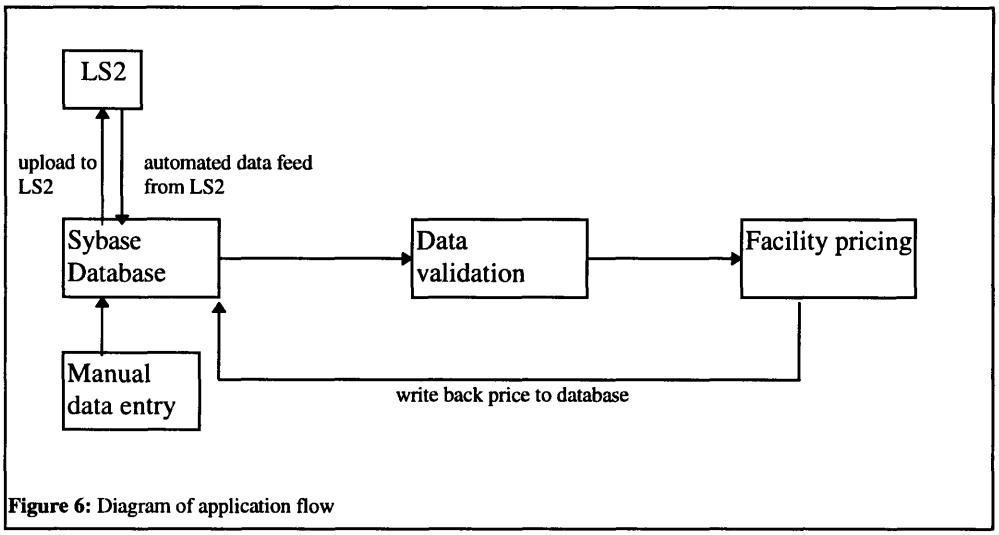


Figure 6: Diagram of application flow

Another way of describing the design of the pricing process of the PMA is through the application usage of this process. This process, as described in Figure 6, starts with the input data being downloaded from LS2 and manual sources. Thereafter the inputs are validated and priced. The pricing model prices the facilities and writes the outputs on the database. Finally the prices generated are uploaded back to LS2 at the end of the month.

5.2.1. Facility inputs and data validation

At the end of each month a snapshot of LS2 data is downloaded onto the Sybase database. This process is run as a simple batch whose task is made easier by the fact that the table layouts in the database server mirror the corresponding ones in LS2. Currently all the inputs required for the pricing are not available in LS2. Some of the data is obtained from other sources, some of which are standard sources like Bloomberg™. All the data from these sources has to be manually compiled and entered into the database. Since LS2 plans to provide this data in a subsequent release, we did not spend any effort to create automated feeds from the other systems.

The next process is to validate all the inputs to the pricing model. This happens at three levels. For the manual inputs to the database there is some rudimentary checking done at the user interface level to make sure that the values entered make sense. The second level is implemented using stored procedures in the database, which validate the inputs required for pricing. There are two stored procedures in the database which, with the help of other stored procedures, check that the inputs are within the bounds desired for them to generate meaningful results. Some of the

checks are simple range checks and ideally these checks should be a part of the table definition. This would have given the database referential integrity and ensured that invalid data does not exist in the database. However this was hard to accomplish due to management issues. Therefore stored procedures were implemented to validate those inputs that the pricing model required. The inputs validated by these procedures can be broadly classified into global validation and facility specific validation.

Global inputs are validated when the model is initialized and are needed for all the facilities. An example of this is the yield curve for the run date. Facility inputs are specific to the facility that is being priced. The global validations are repeated every time a new business run date is set. Facility inputs are checked whenever the price of a facility is requested. These procedures are called from the C++ DLL, which in turn returns three different error codes to the user interface. An error in the global input validation is critical as this means all subsequent pricing attempts will also fail. Errors in the facility inputs are non critical errors as they do not effect subsequent facilities. An input that generates extreme results is flagged as a warning. In the case that the user requests the price for multiple facilities, computation continues for the latter two types of errors. All these errors are logged in a table for inspection after a pricing run.

The reason for separating the validation from the pricing process is to give the users the flexibility to validate the data before starting the relatively more time consuming pricing process. The user can choose only to validate the data and run the pricing model later. In any case validation is always done before pricing a facility. There was an option of implementing the validation procedures in the C++ code for use during the pricing process. This would not have added much work as the infrastructure for exception handling already existed. Validating inputs in the C++ code would be faster as inputs would be checked only along the path of usage and one would avoid the database overhead due to multiple reads of inputs (once for the validation and once for the pricing). The disadvantage of this situation is that this creates a maintenance problem where essentially the same logic is implemented in two places. Since the database overhead did not turn out to be significant, some efficiency was sacrificed for maintainability.

The last level of error checking is not a validation. This case covers those cases where the source of error is not known. An example of this is as follows - given an interest rate, if the model is trying to use a numerical algorithm to converge to a price, there is a limit to the number of iterations it should take. If the interest rate is too low for the given facility the algorithm is not able to converge to a price. If the number of iterations exceeds this limit, the DLL raises an exception

and then the user gets an error that the procedure was unable to converge. Ideally this should not happen as this error should be checked for in the validation procedures. The reason for these types of errors has to be determined by the user. Typically when the reasons for these errors are determined, the appropriate checks are added to the validation procedures. The infrastructure for exception handling is provided in the CException class. There are catch points defined at the level that the code was "tried". Every time an exception is thrown, it is caught at the nearest catch point on the procedure call stack. This mechanism is used to throw exceptions related to calculations and database queries.

5.2.2. Facility pricing

The application usage of the facility pricing process illustrates the interaction between the objects. Analyzing the flow of data from the inputs to the outputs is another perspective from which to design the system. Using this method of analysis the object model can be iteratively evolved. For example, one can start off with just the basic classes (facility, bond, options, database brokers etc.) and the fundamental output values (transfer price, hedge ratio, etc.). thereafter we can iteratively add more classes and more interactions as we figure out how the output values are actually determined. A detailed analysis is not presented here as it would require one to delve into the financial algorithms used in the process, which is proprietary information. The two main concerns while designing interactions were -

- *Modeling of the interactions as they are in the business domain*: Both the interactions as well as the behavior of the classes should try to model the interactions and behaviors of the financial entities. The design of the pricing model tries to implement a particular method in the class where it makes the most sense from the perspective of the business domain. This approach makes the system extensible. If the objects model reality then it is usually easier to evolve them as the business changes.
- *The use of delegation vs. inheritance*: In object oriented development, inheritance and delegation serve essentially the same purpose. We have made an attempt to use inheritance only when the subclass is "a kind of" its superclass.

The facility pricing components were described in the previous section. Figure 7 shows a few of the interactions involved in the calculation of the transfer price in an attempt to illustrate how the objects interact with one another. The interactions of all the objects are not shown as it is

very complex. Moreover, some of the interactions shown are not complete. The idea is to give a flavor of how the objects fit into the big picture.

From a high level, facility pricing has the following steps

1. Initialize model: Sets up the memory for the pricing model.
2. Price facility: This step can be repeated any number of times to price multiple facilities
3. Free model : This step frees all the memory used by the model

The inputs to the DLL come from two sources - the user interface and the database. The user interface provides the DLL with a business run date and a facility to price. The user interface talks to the DLL via the MTM API (application programming interface) which defines the entry points to the DLL.

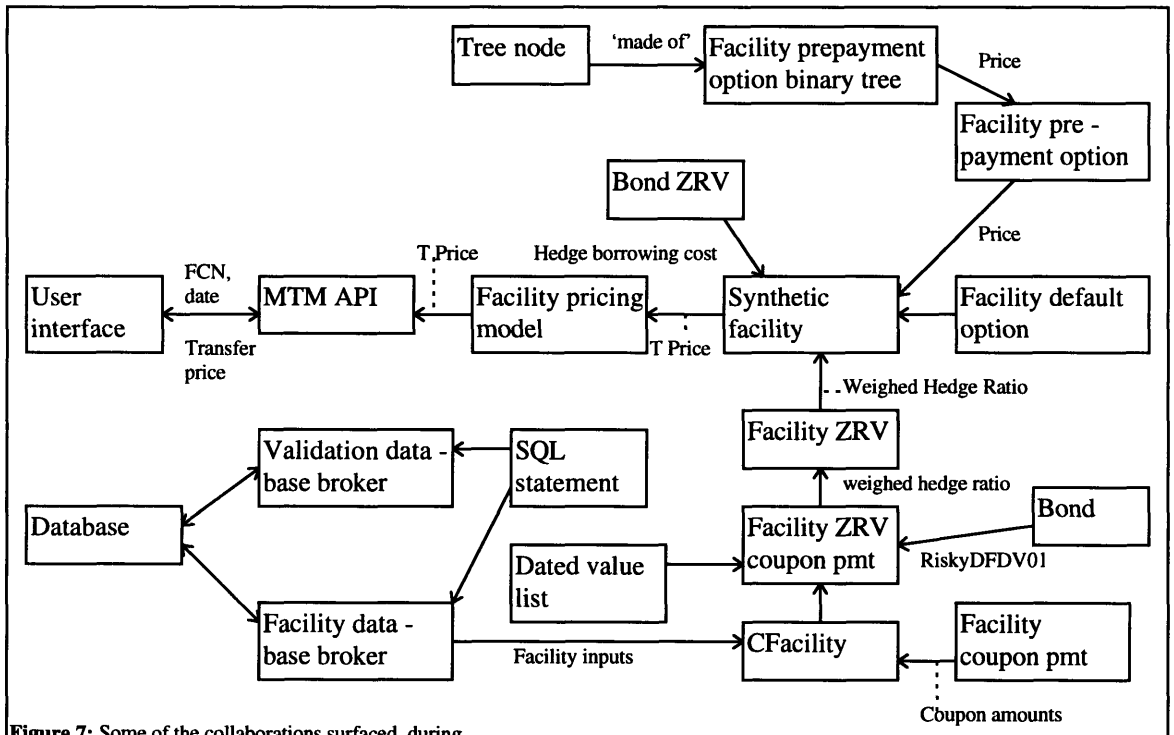


Figure 7: Some of the collaborations surfaced during the pricing of a facility.

The MTM API in turn calls the CFaciliyPricingModel class to obtain the price of a facility. The facility pricing model class does three things before trying to price anything. It initializes the global model environment variable, validates the inputs and reads the inputs into the correct class variables. Among other things the model environment establishes a connection with

the database and generates a handle which is used for communication. The facility pricing model then proceeds to validate all the data for the facility and, if necessary, for the global inputs. This is done with the help of the CValidationDatabaseBroker and the CSQLStatement classes, as shown in figure 7. Reading the inputs is the last prerequisite to pricing. This shown in the figure for the facility inputs. Other inputs like those for associated with bond, yield curve etc. are also read at this time. Again, the communication is established with the help of the appropriate database broker and the CSQLStatement class. The values are populated into the appropriate variables which in this case is an instance of the CFacility class.

The transfer price function is calculated in the synthetic facility class, which in turn depends on the facility prepayment option, facility default option, facility ZRV and the bond ZRV classes for information about their prices, hedge ratios etc. The coupon payment classes behave like payment streams. They model the data and behavior of the coupon payments of their respective underlying assets. The option classes delegate their pricing process to the appropriate binomial tree classes. These in turn depend on the ZRV classes for other inputs (not shown in the diagram). These interactions often lead to circular references. However, in the application there is only a single instance of any finance related class. Therefore, all objects which point to an instance of a particular class point to the same object.

After the outputs are calculated, they are written back to the database. Certain outputs are also passed back to the user interface using memory locations which were specified during initialization. Thereafter the memory used by the model is deallocated.

Although the interactions seem complex, programming these methods was not difficult. This is because the role of every class is well defined and disjoint from the roles of the other classes. Efforts have also been made to implement classes to abstract functions unrelated to the pricing algorithm, an example of which is database communication.

5.2.3. Uploading prices to the database

Whenever a facility is priced, its results are written into a model output table in the database. In addition, the prices are also returned to the Microsoft Access user interface. This is done by putting these answers in memory locations that were passed to the DLL by the Access via the MTM API. At the end of the month a snapshot of this table is stored on LS2 and these data now become official.

6. Implementation

The implementation process is only touched upon in this section. Only the fundamental data representations of the classes are discussed. I have elaborated on those areas where standard algorithms were not used. The specific implementation of the classes is not discussed. Most of the challenge lies in the design and specification of these classes. If this is done correctly, the implementation process is relatively easy.

6.1. Data Structures

The data structures used in the model are as follows:

1. *List structures*: these structures are all subclasses of the COblist class which is a foundation class provided in the class library. The COblist class behaves very much like a linked list of objects. Linked lists were used instead of arrays because of the variable length of the payment streams that were abstracted using this class. Variants of these lists are the most widely used data structures. They are used to represent yield curves and all the subclasses of the CPaymentStream class. This includes interest payments, fee payments, principal payments and coupon payments. There was an option to use dynamically allocated arrays instead of lists since they are more efficient for lookups. However this was not necessary as the lists were not a bottleneck in the pricing process.
2. *Tables*: The program used tables to map strings to objects and objects to objects. This was basically used as an optimization technique to cache intermediate values which are frequently used. The usage of these tables is elaborated upon in the optimizations section.
3. *Trees*: Trees are used to represent the binomial tree classes. Binomial trees, as described in the design section, are different from binary trees in that a node can have two parents. From the perspective of computer science, binomial trees are not trees since there exists more than one path between two nodes. An example of a tree structure is shown in figure 8. Every tree node has pointers to its children and to its parents. The initialization of the tree is done recursively. The tree node structure is initialized by passing to it the addresses of both its parents. One of these pointers can initially be null as the parent pointers can be set at a later time too. Whenever we are at a node and we want to initialize its child we first test to see if it has already been initialized by the other parent. If it has, we set up the links appropriately. This is facilitated by methods which try to find the child of node A by traversing the alternate

path to the child. This is possible since a binomial tree has more than one path from a node to its child.

The trees are also priced in a recursive manner. The 'visited' flag is used to determine whether the node has been already visited and priced. This ensures that the price calculation for every node is done at most once. Here we are using recursion to price the trees, abandoning the totally object oriented approach for this part of the program. This is done because it is a much more efficient way to price the tree rather than sending messages between tree nodes till convergence is achieved.

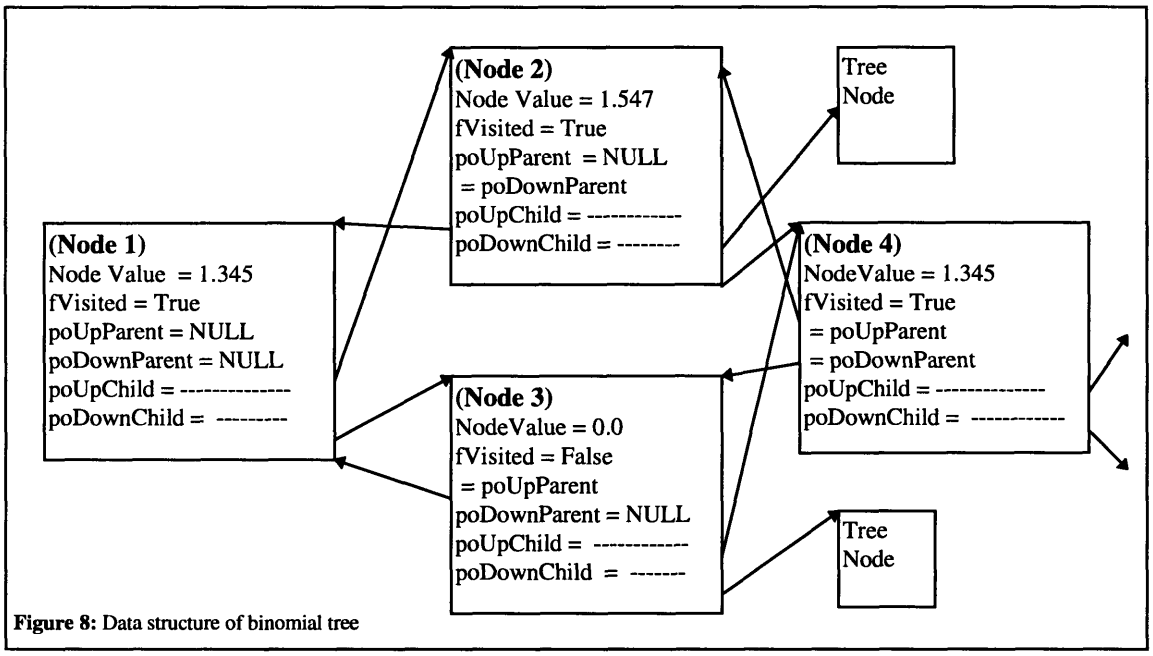


Figure 8: Data structure of binomial tree

6.2. Caching

Caching is used in three different ways in the system. They are as follows

- *Caching intermediate results:* Intermediate values are often cached for methods which are called many times. These decisions are based on statistics obtained by profiling the program. An example of this is the caching of the DV01 values (dollar value of 1 basis point change in the interest rate). A DV01 calculation is very time consuming as it depends on the many other values, many of which are calculated using numerical approximation algorithms. Moreover, these DV01s are requested many times and change only when the risk adjusted discount factors

change. Therefore these values are cached and the cache is flushed when the risk adjusted discount factors change. When the DV01 is requested, the cached value is returned if it exists, otherwise the calculations are performed.

- *Lookup Tables using hash functions*: This was incorporated into the design of a class where we felt the necessity to cache the values beforehand. An example of a class that uses tables is the CCachedValueList class. This class caches all requests for interpolated values between lists. This is useful for structures like yield curves, where discount factors are needed for dates at specific intervals from the business run date. These values are calculated by interpolating over the yield curve. The list caches the values for these particular times and subsequent requests do not need to re-interpolate over the curve. Classes for mapping strings to objects and objects to objects are part of the standard class library and were used extensively for this purpose.
- *Memory reuse:* This feature is useful when many facilities are priced at a time. The users typically price about 200 facilities at a time, using the same business run date. The yield curve is an example of a structure which is reused. After the first facility calculates the yield curve, the curve and its associated discount factors are reused for all the subsequent pricing runs. This point is further elaborated upon in the product readiness section.

6.3. C++ Language Constraint Artifacts

Several issues arose due to the fact that we were implementing the system in C++. They are as follows:

- *Hierarchy of objects*: The fact that every object is created and destroyed by some other object tends to make the hierarchy more tree like. Circular references and complex graph references are discouraged by this C++ constraint as every object is "owned" by some other object.
- *Construction and destruction code*: Since C++ does not implement any garbage collection, the destruction of objects had to be done in a systematic way. This was to ensure that the all the memory was always deallocated upon termination. The issue for the construction procedures was that a constructor cannot call a virtual function. This often results in the duplication of some initialization code.
- *Constraints imposed by strict typing:* The fact that C++ is strictly typed affected the implementation of certain classes. An example of this is the existence of a CDatedValue and a

CTimedValue class. Both these classes associate certain values with certain times, except that the types of the times are different. In some other object oriented languages like Smalltalk, implementing two classes would not be necessary. There would be one class which had temporal values (temporal values associate a value with any point in time).

7. Production readiness

7.1. Memory management

The PMA attempts to reuse memory when ever it can. From observing the usage of the PMA we found that most of the time multiple facilities were priced in a single pricing run. The PMA allows the user to price another facility after it returns the results of the previous facility. If another pricing run is requested then the PMA tries to reuse as much of the memory and calculations that were made for the previous pricing run. As mentioned before, if the business run date is the same as that of the previous facility pricing run then the PMA just uses the yield curves and the yield curve calculators from the previous runs. It also avoids operations like validating the yield curve inputs, reading the model environment variables, etc. There are cases where the reallocation of memory is avoided. An example of this is the memory needed to store the binomial trees. The PMA avoids deleting the memory and reallocating it every time a pricing run is requested. This process of memory management is tricky since C++ does not have garbage collection and all objects created have to be explicitly destroyed. This means that in all possible cases, one has to make sure that all memory is freed upon termination of the program.

7.2. Exception handling

A framework for exception handling is provided for in the DLL. The exception handling mechanism allows the program to throw an exception at any point during its execution. The critical exceptions are of two types - SQL exceptions and application exceptions. SQL exceptions are raised due to some problem related to the remote database server e.g. the inability to connect to the database due to network problems. Application exceptions are raised due to some computation failure like no convergence of a numerical algorithm. Application exceptions can also be used to validate the inputs although the validation is performed by stored procedures on the database to make it independent of the pricing process. If an exception is thrown at any point, it is caught at the closest point on the run time call stack of the program. At the catch point, the program may execute some code and then try to proceed along a different path or throw the exception to the next level. Very often the code executed before throwing the exception to the next level deletes the objects that have been initialized at that level. This is necessary to avoid memory leaks.

The system flags warnings for non critical errors. In this case the current branch is terminated and execution proceeds to the next branch. This feature can be used when the program fails at a point that affects some but not all of the results.

7.3. Testing

The time taken to test the PMA was a significant percentage of the total development time. Testing was done at a unit level and at system level. Each method of every class was tested to ensure that the outputs were as in its specification. The goal of the PMA was to duplicate the results generated by the spreadsheet that it was replacing, and explain any discrepancies that arose. Integration testing was done by comparing the values generated by the PMA to the values generated by the spreadsheet. This was done for all the facilities that were being priced every month ($\approx$ 190 facilities). This method does not ensure that all the paths through the code have been executed and tested. However, given the time constraints to put the model into production, this testing procedure was good enough. The model was debugged till the difference of the results of the model and the spreadsheet were within a specified tolerance.

8. Conclusion and further extensions

The loan pricing model is currently in production and is being used to price loans. The time it takes to price a loan is approximately 10 seconds on a 486 processor as compared to about 5 minutes on the spreadsheet. In addition to being more efficient the users do not have to manually gather all the data as the inputs are automatically downloaded from LS2. The interface is more user friendly and is decoupled from the pricing. This allows more people to use the model without having to experience a significant learning curve. Although the classes from the pricing model have not been reused yet, I envision that there will be significant reuse when another similar application is developed. Examples of other applications that might use some of these classes include other pricing models (e.g. derivative pricing models) and various risk analysis models. Any finance model could use the framework of payment streams, yieldcurves and asset classes.

As we move away from the problem this model may not be extensible in some ways. The objects in the model are tightly coupled to the database table layouts. If the model were to be extended into a generic analytical engine then a mapping mechanism would have to be added to map the inputs to the objects. This could involve a significant amount of work. Another weakness of the design is that although attempts were made to separate out the specifics of the pricing model, in some cases the member functions of the generic classes are algorithm specific. Hence, some of the classes may need some re design if the second project is completely different from this one. A third problem may arise due to the fact that the model has not been designed for parallelism, i.e. if we want to price several loans together in a multi processing environment, significant re design will be required.

However, there are ways in which the system can be extended. The system is extensible both in the business and technical domains. Some possible functional extensions involve interest rate simulation, what-if analysis, and portfolio level risk analysis. Interest rate simulation involves running the model using different yield curves. What-if analysis simulates various other market scenarios. Portfolio analysis attempts to analyze a set of loans and the risk associated with them as a whole.

As technical enhancements, parts of the system can be used to make a finance toolkit or class libraries. While designing the system we tried to separate the application specific algorithms into different classes. As a result, most of the classes can be reused by any system attempting to solve a problem in a similar business area. A second extension could involve redesigning the

system using a more sophisticated design methodology. As discussed in the design methodology section, we used RDD (Responsibility Driven Design) which is one of the simplest methodologies to use. Using a methodology which required more analysis may improve the current object model. On the C++ side, certain advanced techniques could be added to make coding easier. An example of this is the usage of reference counting. This process keeps track of the active references at any given time and destroys an object automatically when it is not being referenced.

Given the high rate of change in the field of finance, the ability of a system to evolve and the length of the development cycle are two very important factors. We believe that the approach used while designing and developing this pricing model helps optimize both these factors.

9. References

[1] Bankers Trust Architecture 2.0

[2] James Martin, James J. Odell, *Object Oriented Analysis and Design*, Prentice Hall 1992

[3] Rebecca Wirfs-Brock, Brian Wilkerson, Lauren Wiener, Designing *Object-Oriented Software*, Prentice Hall 1990

[4] Grady Booch, *Object Oriented Design with Applications*, Benjamin/Cummins Publishing Company 1991

[5] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, Gunnar Overgaard, *Object-Oriented Software Engineering: A Use Case Driven Approach*, ACM Press 1992

[6] Derek Coleman, Patrick Arnold, Stephanie Bodoff, Chris Dollin, Helena Gilchrist, Fiona Hayes, Paul Jeremaes, *Object Oriented Development The Fusion Method*, Prentice Hall 1994

[7] John C. Hull, *Options Futures and other Derivative Securities*, Second Edition, Prentice Hall 1993