Frameworks in the Financial Engineering Domain An Experience Report

Andreas Birrer
Thomas Eggenschwiler

UBILAB
Union Bank of Switzerland
Bahnhofstrasse 45, CH-8021 Zurich, Switzerland
E-mail: {birrer, eggen}@ubilab.ubs.ch

Abstract: To supply the financial engineering community with adequate and timely software support we advocate a reusability oriented approach to software development. The approach focuses on frameworks and reusable building blocks. This paper presents a domain specific framework for a calculation engine to be used in financial trading software. It is as such an example of using frameworks outside their typical domain of graphical user interfaces.

1 Introduction

At UBILAB, the Information Technology Lab of Union Bank of Switzerland, the ET++Swapsmanager project [Egg92] is an effort to find a satisfying way to provide traders with sophisticated software support by employing a reusability-oriented approach based on object technology.

Financial engineering is concerned with the creation and valuation of financial instruments. To help understand the following architecture, we must look at some of the basics that drive financial markets.

The *primary financial markets* are the places (not necessarily physical) where supply of and demand for funds meet. For instance, if a corporation needs money to finance its operating expenses or some other investment – such as buying a subsidiary – there are roughly three sources of funds. The company could go to a bank and apply for a loan. The second possibility is to issue stock (equity securities) and thereby have investors participate in the investment's returns (and risks). The third approach is to issue some sort of bond (debt securities). In all of these cases the investors (suppliers of funds) expect a return to compensate for forgone opportunities. This return is usually in the form of interest payments. From the company's point of view it is the price it has to pay to temporarily dispose of the funds.

Besides the mere price there are other factors such as when the interest payments are due, how these are determined, the currency, the amount of security (collateralisation) provided, etc. With all these variable factors considerable effort, and therefore cost, must be expended to match investors and borrowers and negotiate on the various characteristics of a potential deal. *Financial instruments* help reduce these costs by standardising the conditions under which financial transactions are carried out. Investors and borrowers can then choose the most attractive construction (instrument) available to them; typically with the aid of some intermediary. This is also eased by the fact that most financial instruments are highly fungible and traded anonymously.

Financial instruments have evolved from mere contracts for lending and borrowing to specialised means for transforming various characteristics of funds as corporations have learned how to use such instruments to manage all kinds of risks. Examples of the characteristics are determination of interest amount, interest payment schedules, exposure to market movements and currency conversion. It is therefore possible to customise a transaction by pipe-lining several financial instruments. The result, also called a *synthetic instrument*, is then sold as a package.

Each instrument and its associated practices of trade can be considered as defining one *financial market*. However, for a market to work efficiently, some system for *price indication* is needed. Active participants (especially traders) are the main source for this information. In the markets with high turnover volumes the prices at which deals are concluded are continuously published on electronic networks. We will refer to instruments for which the current price level is publicly known as *standard instruments*.

The price of a standard instrument is almost exclusively determined by supply and demand. But if a new instrument is engineered, how does one find its proper price? A widely accepted approach is to use the prices quoted in high volume financial markets for the derivation of a discount function. The higher the volume traded the more reliable is the price indication.

A *discount function* is used to assign to any cash flow in the future an equivalent value today. This is based on the observable fact that a Dollar received today is worth more than a Dollar received some time in the future; this is also known as the *time value of money*. Deriving a discount function effectively means expressing the time value of money in a market independent form. The present value, and hence the price, of an instrument is found by discounting all its associated (future) cash flows.

Valuation not only makes sense in the context of a single instrument. In effect, small and large collections of deals stemming from various instruments[1] are managed and evaluated on a portfolio basis. This is especially true for investment banks and broker houses that actively trade in these instruments. Exposure to market movements (price

[1] In object parlance: an instrument is a class of deals, a deal would be an instance of an instrument.

risk) and performance is measured with respect to an entire portfolio rather than on individual deals.

The rest of the paper is organised as follows: In section 2 we give an introduction to the swaps business and shed some light on the current situation concerning software support. In section 3 we go into the details of a calculation framework for financial instruments. In section 4 we summarise our experiences of the development and usage of the framework.

2 Swaps

Of the kaleidoscopic palette of today's financial instruments we will for the purpose of this paper only consider one genre of financial instruments, so-called *swaps*.

2.1 The Swap Product

Swaps are a fairly recent development. The most common of these structures is the *interest rate swap* where two parties agree to exchange, or swap, two sets of interest payments which are determined by two different interest rates [Kap90]. Hence there are two sides, or legs, to a swap. Typically, one of the interest rates is fixed at a certain percentage of some principal amount and the other is floating, i.e., pegged to some changing market index. Fig. 1 shows a situation where borrowers B and D engage in an interest rate swap. The rationale for doing this is that B can raise funds on floating rate terms comparatively cheaper than D can. Although B supposedly prefers to engage in a fixed rate liability it is still profitable to take on a floating rate debt and then swap it with D who has opposite preferences.

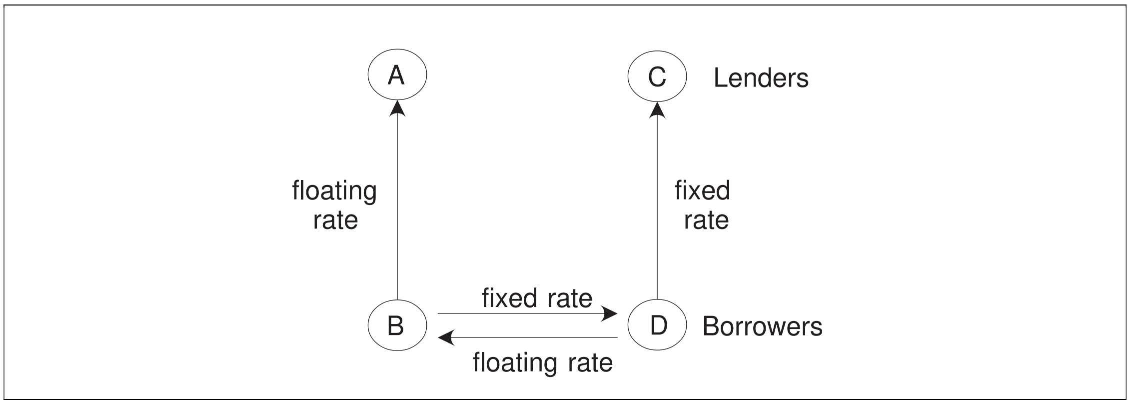

Fig. 1: Flow of interest payments when two borrowers swap their obligations through an interest rate swap

The other frequently met construction is the *currency swap* in which the flow of interest payments is identical to the interest rate swap but with the difference that the two sets of cash flows are denominated in different currencies.

At conclusion of a swap, the two sides must have equal value under the then prevailing market conditions. This possibly calls for a compensation payment by one of the counterparties. The value of one swap side is determined by present valuing each cash flow according to an appropriate discount function.

In trading situations it is not the present value of the two sides that is of most relevance to the trader or the customer. It is either the interest rate of the fixed side or the spread in excess to the floating side that are then referred to as the *price* of the swap (We will use price and rate interchangeably).

As market conditions (interest rates, exchange rates) change, the two sides of a swap perform differently and will eventually be misbalanced. Managing these misbalances on a portfolio basis is of paramount importance to the swaps trader.

2.2 Current Situation

The correct valuation of financial instruments in a trading situation depends on the availability of adequate software tools. Adequacy is needed with respect to both the user interface and the calculation support.

Innovation driven by competition among suppliers of financial instruments requires that software tools must constantly be extended to include calculation support for new financial products.

However, current approaches to the development of software tools in financial engineering are not satisfactory. Software tools are either built on top of a spreadsheet program or as a separate application. Both approaches have a number of deficiencies.

While spreadsheets allow for fast development of new tools they are often too slow for extensive calculations and almost impossible to maintain once the computational models become more complex. The user interface typically is very rudimentary and performs poorly under trading conditions.

Development of standalone applications is very costly and time consuming. Therefore developers often resort to “quick and dirty” implementation techniques in order to meet time constraints dictated by the fast moving markets. These time constraints have also led to minimalistic, menu driven, user interfaces.

A very crucial shortcoming of both approaches is that similar functionality is re-engineered for every new tool.

2.3 Towards better Solutions

Almost each instrument has its own trading practice and jargon. However, when looking at the instruments and the valuation procedures on a more abstract level one can find many commonalties and interrelationships among them. Most notably:

- Each deal consists of a set of actual or potential cash flows.
- A cash flow has an associated business day on which it is realised.
- The value of any instrument is sensitive to changes in interest rates. Therefore the quantification and monitoring of the interest rate sensitivity is applicable to any instrument.
- Arbitrage between different markets tends towards consistent “time value of money”-profiles across markets which allows the valuation of one instrument in terms of the price structure of other instruments.
- Instruments can easily be expressed in terms of each other when adopting a LEGO® (or building blocks) approach in trying to understand financial instruments [Smi87].

These observations led us to the conviction that the domain of financial instruments is ideally suited for the employment of a framework approach.

However, a framework can not be designed from scratch on paper. Rather, experience suggests that it must grow bottom-up and successively mature by applying it to different problem (sub-)domains. Therefore, with the notion of a general framework for financial engineering in mind we set out with our first development in the domain of swaps trading.

Fig. 2 shows a pilot application that was the result of a series of prototypes implemented in collaboration with trading specialists. The purpose of this endeavour was to acquire initial domain know-how and experiment with various user interfaces.

The upper right window shows a domain specific desktop on which domain items are depicted as icons. Relationships among the items may be edited through drag-and-drop type manipulations on the icons. The bottom left window shows an editor in which data can be modified directly via its graphical depiction by directly dragging a part of the curve.

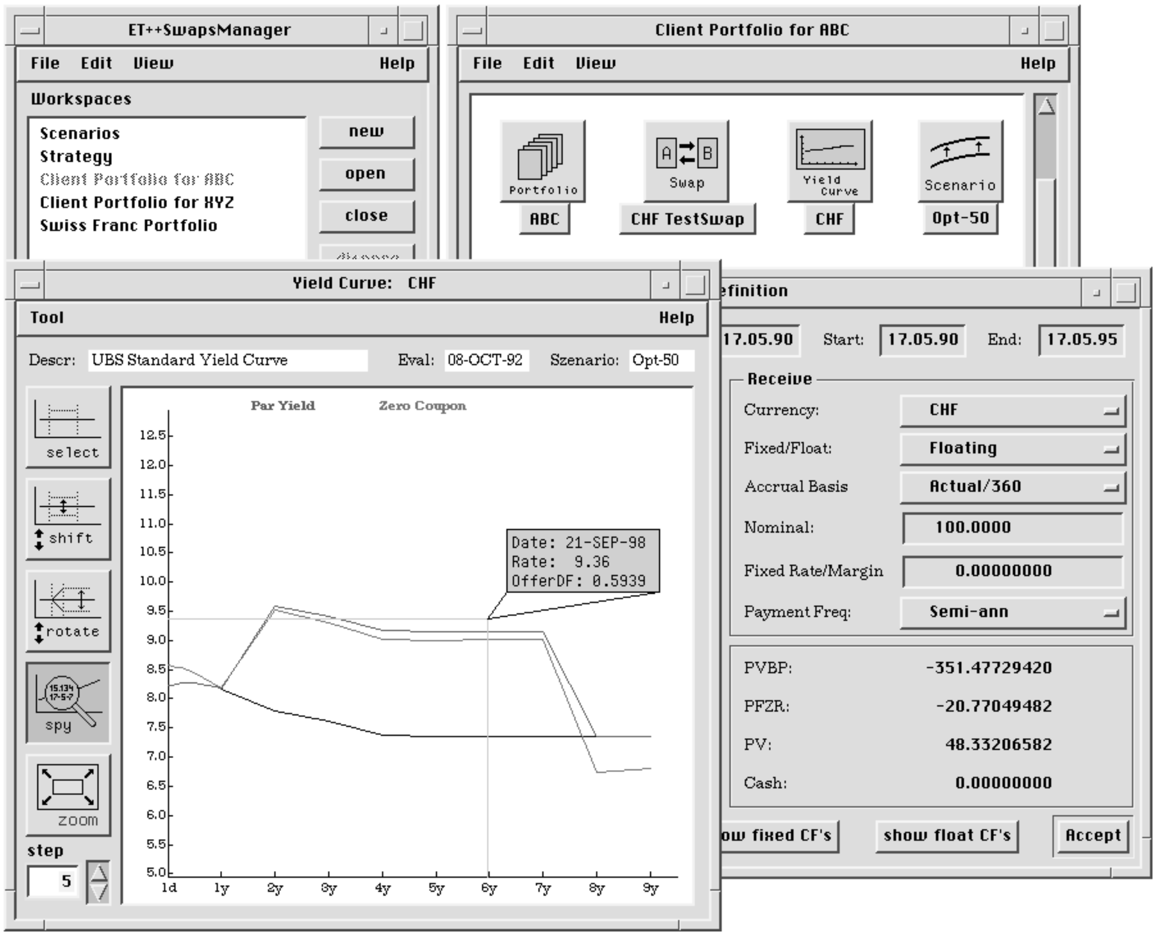

Fig. 2: Screen sample of Swaps Manager I

The following section describes the framework as it is being used in the context of a swaps trading software.

3 The Architecture of the Calculation Engine Framework

Fig. 3 shows the structure of the framework and the most important objects. We will show the key abstractions in more detail, and then we will describe two scenarios which demonstrate their use.

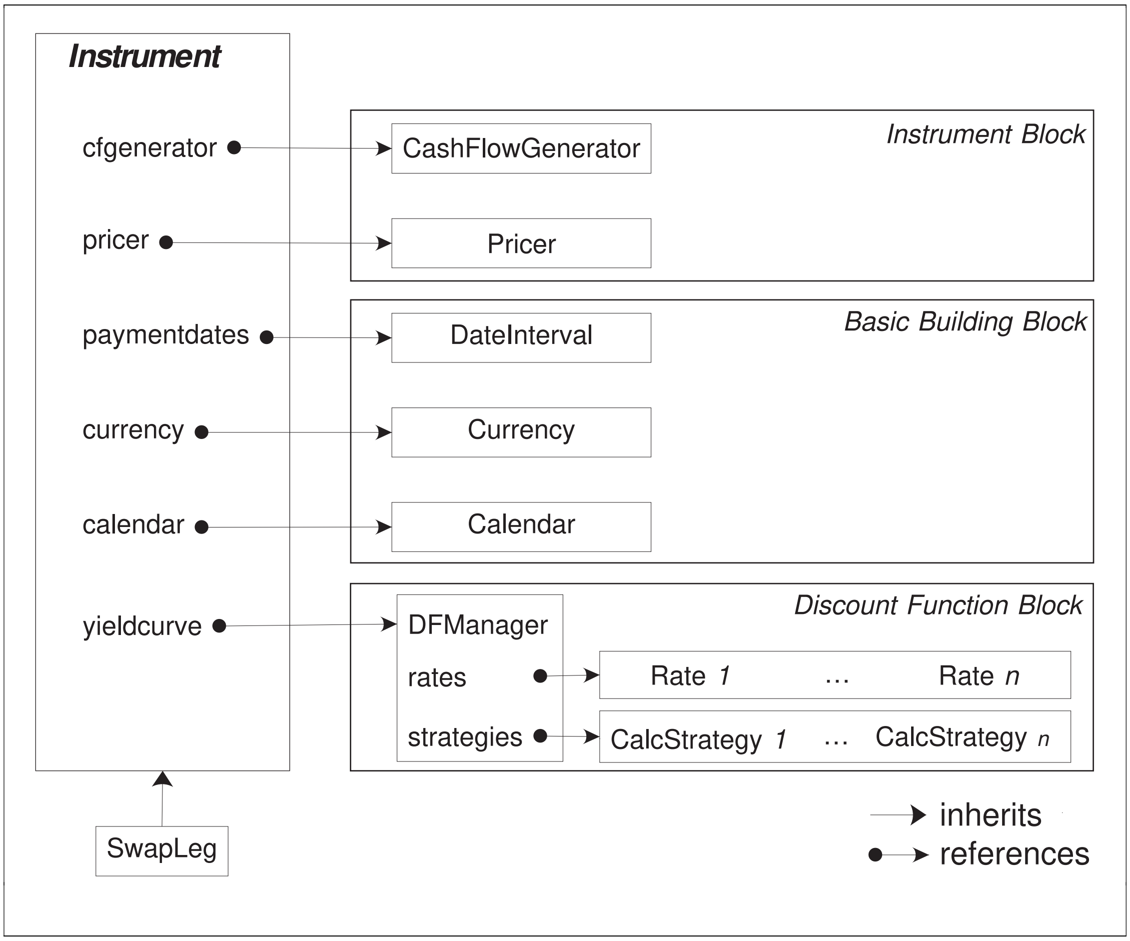

Fig. 3: The calculation engine objects

3.1 The Key Abstractions

As mentioned in section 1 standard instruments and the conventions to guide their trading define financial markets. The most important guidelines deal with the pricing of an instrument. The main problem here is achieving comparability of cash flows occurring at different dates. This is solved through the definition of a discount function which allows to calculate the present value of any cash flow due in the future [Mir91]. This leads to two key abstractions, the *instrument* and the *discount function.*

3.1.1 Instrument

An instrument represents any traded facility for generating cash flows in today's financial markets.

Instrument	
• generate cash flows • calculate the present value • calculate the price • construct the payment dates • construct generator	- CFGenerator - Pricer - DFManager - Calendar - DateInterval

Fig. 4: CRC-Card [Beck89] for an instrument

Every instrument has the ability to generate the cash flows from a description and a valid market environment by a discount function manager. It has a connection to objects encapsulating relevant market information (e.g., `Currency`, `Calendar` and `DFManager`).

Because we can't describe the algorithm for cash flow generation at an abstract level, we introduce objects which encapsulate this algorithm. This is a well known design pattern, called *strategy pattern* [Gam92], which we use in several situations.

CFGenerator	
• generate cash flows	- Instrument - DFManager - Currency - Calendar - Accrualbasis - DateInterval

Fig. 5: CRC-Card for a generator

The CRC-Card of Fig. 5 shows only one responsibility. This narrow set of tasks is typical for a strategy object. A specific implementation of a generator knows more about its instrument and is able to combine its parameters to achieve the results wanted, i.e., a list of cash flows. A similar technique is used for pricing an instrument. Fig. 6 shows the CRC-Card for a *pricer*.

Pricer	
• calculate the price • calculate the present value	- Instrument - DFManager - Currency - Calendar - Accrualbasis

Fig. 6: CRC-Card for a pricer

Since `Instrument` is an abstract class, it can't decide which strategies are useful for a given implementation. It delegates their allocation to its subclasses by means of factory methods. A factory method is a method that is called by a base class to create a subclass dependent object. These are called *`DoMakeGenerator`* and *`DoMakePricer`*.

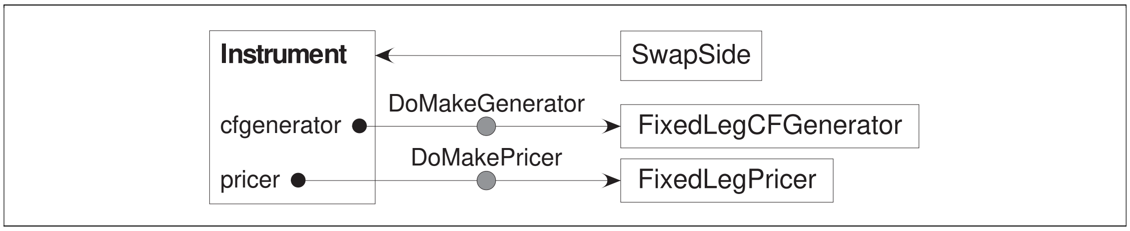

Fig. 7: Factory methods to generate customised strategies

An `Instrument`'s interface is defined at the level of the abstract class, which allows to treat a collection of instruments. This collection generates cash flows as if it were a single instrument. Therefore a portfolio of instruments can itself be modeled as an instrument. Fig. 8 shows the class hierarchy for instruments.

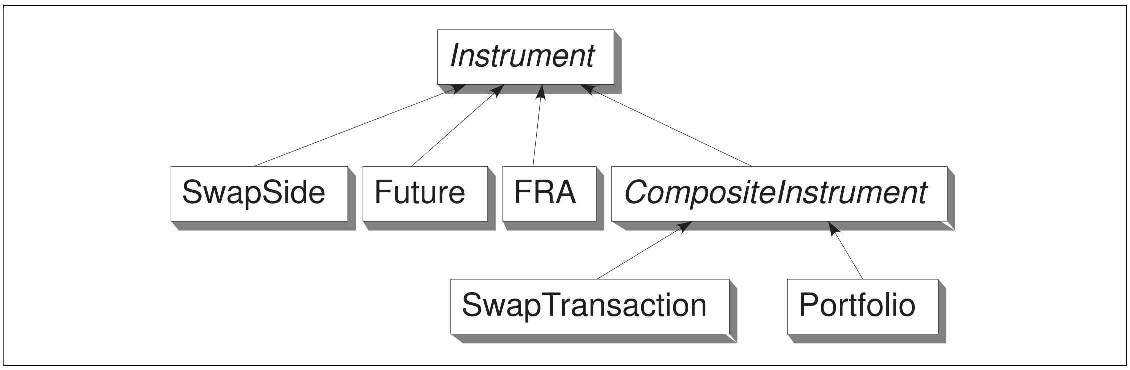

Fig. 8: The class hierarchy for instruments

3.1.2 Discount function

The discount function for a given market mix is implemented by a `DFManager`.

DFManager	
• incorporate pricing information. • imply prices at dates where no price is quoted by the market. • calculates the discount function	- Rate - CalcStrategy - DFCalculator - InstrumentType

Fig. 9: The CRC-Card for the DFManager

We calculate a discount factor for a given standard instrument by the method of zero-coupon pricing. For different kinds of standard instruments we use different formulas.

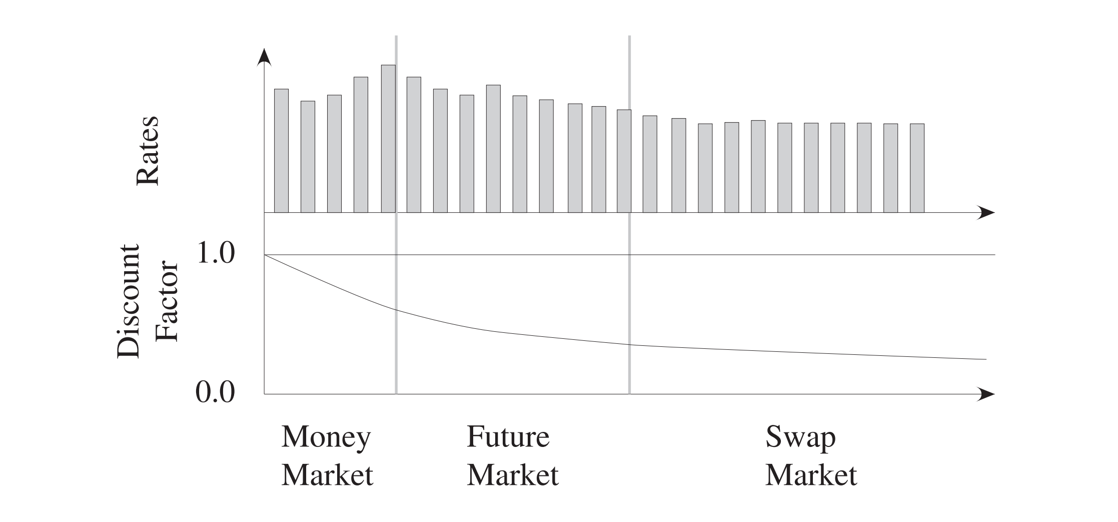

Fig. 10: Discount Function constructed from different kinds of rates

The `DFManager` forwards the calculation of discount factors (out of different kinds of rates as shown in Fig. 10) to specialised objects. Again we apply the same design pattern to this problem as for the generators and pricers and called them calculation strategies. They encapsulate the algorithm which is needed to compute the value of the discount function over a range of homogenous rates. These strategies are selected at calculation time by the type of the rate we calculate the discount factor for. A calculation strategy may depend on the discount factors at previous points to calculate a correct value. For every type of rate there must exist a calculation strategy.

Fig. 11 shows the structure of a DFManager object. The `rates` attribute consists of rate objects with each representing a standard instrument.

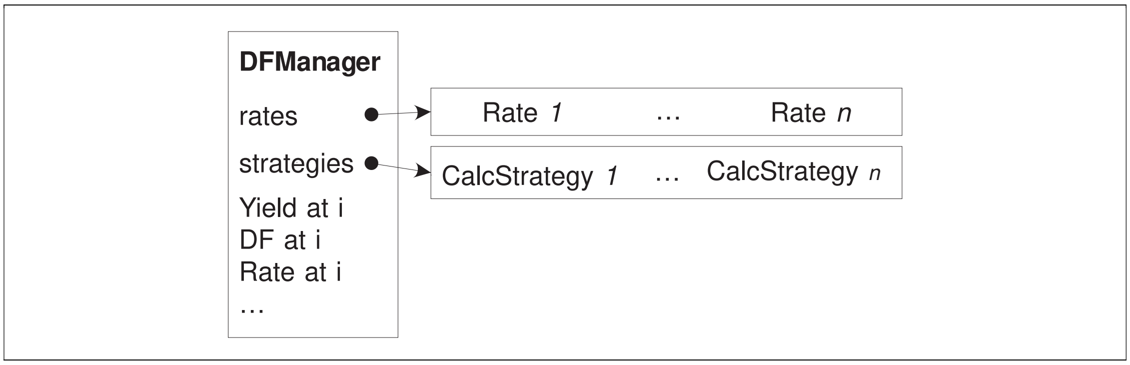

Fig. 11: The structure of a DFManager object

Currently, we support three kinds of standard instruments (Fig. 12).

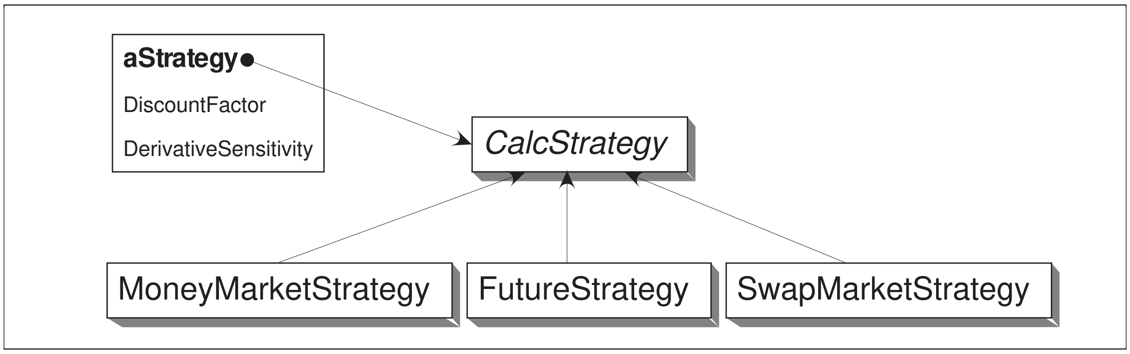

Fig. 12: The interface of the class `CalcStrategy` and it's sublcasses

Based on the above factorisations, the structure of the calculation context may be set-up at will. The `DFManager` inspects its structure when it has to calculate something for the first time and installs the needed strategies (through factory methods).

Consider a trader doing business in Swiss Franc Interest Rate Swaps. In these markets there exist several kinds of instruments which influence trading decisions. To achieve a better overview of the current market situation, several `DFManagers` can be in use at the same time. They differ in the mix of included prices. In this way we support the trader in integrating simultaneously accessible market information and arriving at a more accurate decision.

3.2 Scenarios

To illustrate the use of the classes, we present two scenarios which show the interaction among the actual objects involved.

3.2.1 Discount Factor Calculation

In the first scenario a `pricer` object starts a request for a discount factor at a date where a rate is defined, to calculate the present value of a cash flow (cf. Fig. 3).

This is processed in the following way:

```
double DFManager::DiscountFactor(Date theDate)
{
    Rate *theRate= rateslist->GetRate(theDate);
    CalcStrategy *cs= this->GetStrategy(theRate);
    return cs->DF(theDate);
}
```

What is the advantage of doing it this way? A `DFManager` relies only on the interface of the collaborator objects. It just uses the responsibilities defined in their abstract base class and manages them. We achieved a strong separation between structure information (`Rate`), selector knowledge (`DFManager`) and procedural knowledge (`Strategy`). To identify such a collaboration scheme we again use a design pattern, the *manager pattern.*

To support a new kind of rate, we implement the corresponding formula in a strategy subclass and define the new rate. So we end up implementing one new subclass with essentially one formula and possibly a caching algorithm.

3.2.2 Cash Flow Generation

The second scenario shows how cash flow generation works in an instrument. We describe a fixed swap leg which is fairly easy to understand without further knowledge in financial engineering.

A fixed swap leg is defined by several parameters, the most important being *principal*, *fixed rat*e, and *accrual basis*:

- Principal denotes an amount of money which is the base for cash flow generation.
- Fixed rate is the interest rate used on the principal to calculate the amount paid in regular intervals.
- Accrual basis is the definition of the amount of days per year for which interest accrues.

These definitions are specific to one instrument, a fixed swap leg. To process the information correctly, a generator knows about the parameters supplied by its instrument. So a fixed leg generator accesses the information of a fixed swap leg by the specific swap leg interface.

Information that is common to more than one class is used through the interface of the abstract base class. The aim is to always use the most abstract interface possible.

In the scenario the swap leg collaborates with its generator to produce the cash flows. The generator obtains the necessary parameters from its instrument, in this case the fixed swap leg. The following shows a generic algorithm which implements the formula found in many swaps handbooks.

```
void FixedLegCFGenerator::GetCashFlows(List &theCFList)
{
    Interval *cashdate= swapleg->GetInterval();
    AccrualBasis *acb= swapleg->GetAccrualBasis();
    Date at,prev;
    double amount;

    //--- generate interest payments
    while (cashdate->Next(at)) {
        cashdate->Previous(prev);
        double accrual= acb->Alpha(prev, at);
        amount= Principal(prev) * Rate(prev) * accrual;
        theCFList.Add(new CashFlow(at, amount, …));
    }
}
```

In the above code sample we use several parameter objects whose purpose we list briefly:

- `Interval` describes the structure of the payments over time.
- `AccrualBasis` is an object encapsulating the convention on how to calculate interest accrual periods.

To iterate over the payment periods, the generator uses `Interval`. This object generates in the loop all the dates necessary for the calculation. In each iteration one cash flow object is generated to store the information. The generator adds these to the list and returns the list to the instrument.

Evaluation

This example shows us not only the de-coupling of *what* and *how*, but also the co-operation of smaller abstractions. The `Interval` is defined by one object and used by another. The objects are solely connected by the interval object and its respective protocol. This is used in the swaps leg, for example, where one object knows how to specify the temporal structure, and the other needs to know the actual dates for the calculation.

The `AccrualBasis` calculates accrual factors with the accrual basis given by the swap leg.

To transfer cash flows as parameters we introduce a class `CashFlow` which stores all relevant data for later use.

Thus we achieved effective de-coupling by introducing fine grained abstractions that only define responsibilities. There are many different specific implementations which are all used homogeneously.

4 Experience

Our experience from developing the ET++SwapsManager and from earlier projects is that building, or better growing, a framework is hard work. It takes recurrent, conscious efforts to review a working design and improve it.

Typically, in developing a software system, functionality is added and tested incrementally. As a consequence the functionality of the system grows without much concern for the overall design. But, to really advance the design it has to be reworked periodically. Thus it is possible that, while preserving the monotonically increased functionality, the system decreases significantly in size, due to consolidation efforts.

Development of a framework starts with the implementation of a specific solution in a particular domain. Then the solution is successively reworked to cover a family of applications. For this process of generalising a design we found a few rules of thumb which were helpful in our project:

- Try to consolidate similar functionality found in different parts of the system and implement it through a common abstraction.
- Split up a large abstraction into several smaller abstractions such that a team of collaborating objects results. Each of these smaller abstractions then implements only a small set of responsibilities. A good example of this is separating pricing and cash flow generation from the instrument abstraction as described in section 3.1.1.
- Objectify abstractions that differ; implementing each variation of an abstraction as an object increases the flexibility of a design.
- Use composition instead of inheritance where possible. This reduces the number of classes in a system. Additionally, it relieves the (re-)user of the framework from having to deal with the inner workings of participating classes.

Especially the last rule seems crucial in the development of a highly flexible framework. The novice often tries to find class hierarchies in the problem domain and structures the needed responsibilities and code accordingly. The call for multiple inheritance is then merely a consequence when one is confronted with implementing an abstraction that is really a combination of already available sets of responsibilities. Structuring functionality along inheritance hierarchies, however, leads to static configurations. Frameworks draw their superiority from well designed collaborations among teams of objects.

Acknowledgements

We would like to thank Erich Gamma and André Weinand for their constructive comments on drafts of this paper and encouragement for redesigning it. We also thank Federico Degen and Jürg Gasser for their support on the financial engineering side of the project.

References

[Beck89] K. Beck, W. Cunningham, “A Laboratory For Teaching Object-Oriented Thinking,” In OOPSLA’89 *Conference Proceedings* (*October 1-6, New Orleans, Louisiana),* published as, OOPSLA’89, ACM SIGPLAN Notices Notices, Vol. 24, No. 10, November 1989, pp. 1-6.

[Bro87] F. P. Brooks Jr., “*No Silver Bullet - Essence and Accidents of Software Engineering*,” *IEEE Computer*, Vol. 20, No. 4, April 1987.

[Egg92] T. Eggenschwiler and E. Gamma, “*ET++SwapsManager: Using Object Technology in the Financial Engineering Domain*,” In OOPSLA’92 *Conference Proceedings (October 18-22, Vancouver)*, ACM SIGPLAN Notices Vol. 27, No. 10, pp. 166-177.

[Gam92] E. Gamma, *Objektorientierte Software-Entwicklung am Beispiel von ET++ – Design-Muster, Klassenbibliothek, Werkzeuge*, Springer-Verlag, Berlin, 1992.

[Gam93] E. Gamma, R. Helm, and J. M. Vlissides, “*Design Patterns: Abstraction and Reuse of Object-Oriented Designs,”* In ECOOP *Conference Proceedings (July 26-30, Kaiserslautern)*, Springer Verlag, 1993.

[Joh88] R. E. Johnson and B. Foote, “*Designing Reusable Classes*,” *The Journal Of Object-Oriented Programming,* Vol. 1, No. 2, 1988, pp. 22-35.

[Kap90] K.R. Kapner and J.F.Marshall, *The Swaps Handbook - Swaps and Related Risk Management Instruments*, Institute of Finance, New York, 1990.

[Mir91] Miron P, Swannell P, *Pricing and Hedging Swaps*, Euromoney Publications PLC, London, 1991.

[Smi87] C.W. Smithson, “*A LEGO Approach to Financial Engineering: An Introduction to Forwards, Futures, Swaps and Options,*” In Midland Corporate Financial Journa*l*, Winter 1987, pp. 64-86

[Wei89] A. Weinand, E. Gamma, and R. Marty, “*Design and Implementation of ET++, a Seamless Object-Oriented Application Framework,”* Structured Programming, Vol. 10, No. 2, June 1989, pp. 63-87.