ET++SwapsManager: Using Object Technology in the Financial Engineering Domain

Thomas Eggenschwiler
Erich Gamma

UBILAB
Union Bank of Switzerland
Bahnhofstr. 45
CH-8021 Zurich, Switzerland
Tel.: +41-1-236 79 47
E-mail: {eggen,gamma}@ifi.unizh.ch

Paper category: Experience

Abstract

Providing the financial engineering community with adequate software tools presents several challenges to application developers. Experience shows that reusability-oriented software development as supported by object technology has the potential to meet these challenges. To back the argument, a project of building a pilot implementation of a swap valuation system using a comprehensive class library including an application framework is reported. As a novelty the project emphasized the use of so-called design patterns. The project experience suggests that the use of design patterns significantly eases the application of a large class library and facilitates the reuse of design.

1 Background

UBILAB's (Union Bank of Switzerland's Informatics Laboratory) software engineering group is currently involved in adopting object technology. During the last two years this group's core activity was the development and refinement of ET++ [Wei88, Wei89, Gam90], a comprehensive class library for C++. The work on ET++ started in 1987 at the University of Zurich. In 1990 the developers of ET++ joined UBILAB.

The value of ET++ for the construction of applications with an advanced user interface was demonstrated in various projects inside and outside UBILAB. However, an application that illustrated the value of objects in the context of a banking application was still missing. To overcome this deficiency, we chose the problem domain of financial engineering applications.

2 Challenges of Financial Engineering Applications

Financial markets continue to force suppliers of financial services and products to design and implement more effective solutions to ever complex financial problems. Such innovation increasingly depends on the availability of adequate software tools. In an attempt to provide these tools the software engineer is confronted with the following challenges:

- The pace of the markets requires a flexible development process. Once an opportunity for a new financial instrument is perceived software development must be able to react quickly with adequate support. Therefore *short life cycles* and tight communication between developers and traders are essential.
- Traders want to assess the consequences of a deal more accurately through more complete, and consequently larger, models of computation. To help risk management to capture the big picture flexible "what-if" analyses must be provided.
- Furthermore, the interconnection of financial markets brings together a myriad of market conventions, for example day-count conventions. This requires a base of reusable domain specific building blocks which can not be engineered separately for each tool. As a result, the development focus can no longer be one single project but spans entire *project families*.
- Intuitive human/computer interfaces are necessary to reduce a trader's mental load and optimize trading activities.

In view of these challenges, a reuse-oriented approach to software development appears attractive.

3 The ET++SwapsManager

The ET++SwapsManager is a tool for the valuation of swaps. It was implemented in collaboration with trading specialists from the swaps department at UBS.

3.1 ET++SwapsManager Objectives

The ET++SwapsManager project focused on the following goals:

- *Promotion of modern user interface technology for banking applications*
  We emphasised a highly graphical user interface based on the direct manipulation principle beyond push buttons and menus.

- *Object technology as a tool for developing banking applications*
  The project should help to evaluate the suitability of object technology in general and ET++ in particular for the development of banking applications.
- *Domain specific frameworks*
  The construction of frameworks for a specific problem domain is a key issue of using object technology on a large scale. As a first step in this direction the project should be a mean to acquire domain knowledge.
- *Design Patterns*
  Design patterns are a way to abstract and to reuse design experience. In the pilot application we wanted to try and evaluate this approach.

3.2 Swap Valuation in a Nutshell

A financial swap is an exchange of two streams of (future) cash flows between two parties [Kap90]. A swap has two sides which differ at least as to the currency of the cash flows or how the cash flows are determined, i.e., fixed or floating (bound to some market index which is set periodically).

As the term “swap” suggests, at the time of the deal, both streams or sets of cash flows need to have equal value. To compare the two streams the associated cash flows are discounted (“present valued”) based on interest rates taken from reference markets.

Since the two sides of the swap are not identical it follows that as interest rates move, the values of the two sides of a concluded swap will eventually be misbalanced. Managing such misbalances on a portfolio (collection of swaps) level is of utmost importance to the swap trader.

In what follows we will sometimes use the convention of referring to an interest rate as a *yield*. A yield is a normalized interest rate. Yields can be gathered from market data for various investments with different terms to maturity. The term *yield curve* refers to a set of yields for different terms to maturity.

3.3 ET++SwapsManager Concepts

The ET++SwapsManager has a highly interactive "object-oriented" user interface. Abstractions from the trading domain are presented to the user as iconic objects. These objects are:

– *Swap*

– *Portfolio* - a collection of swaps

– *Yield curve* - yields for a set of maturities, yields are major input data for swaps pricing

– *Scenario* - interest rate differences applicable to a yield curve

– *Counterparty*

Data pertaining to these objects can be edited through object specific inspectors/editors.

Much emphasis was put on *direct manipulation* and on *adaptation of established user interface metaphors* to the trading activities.

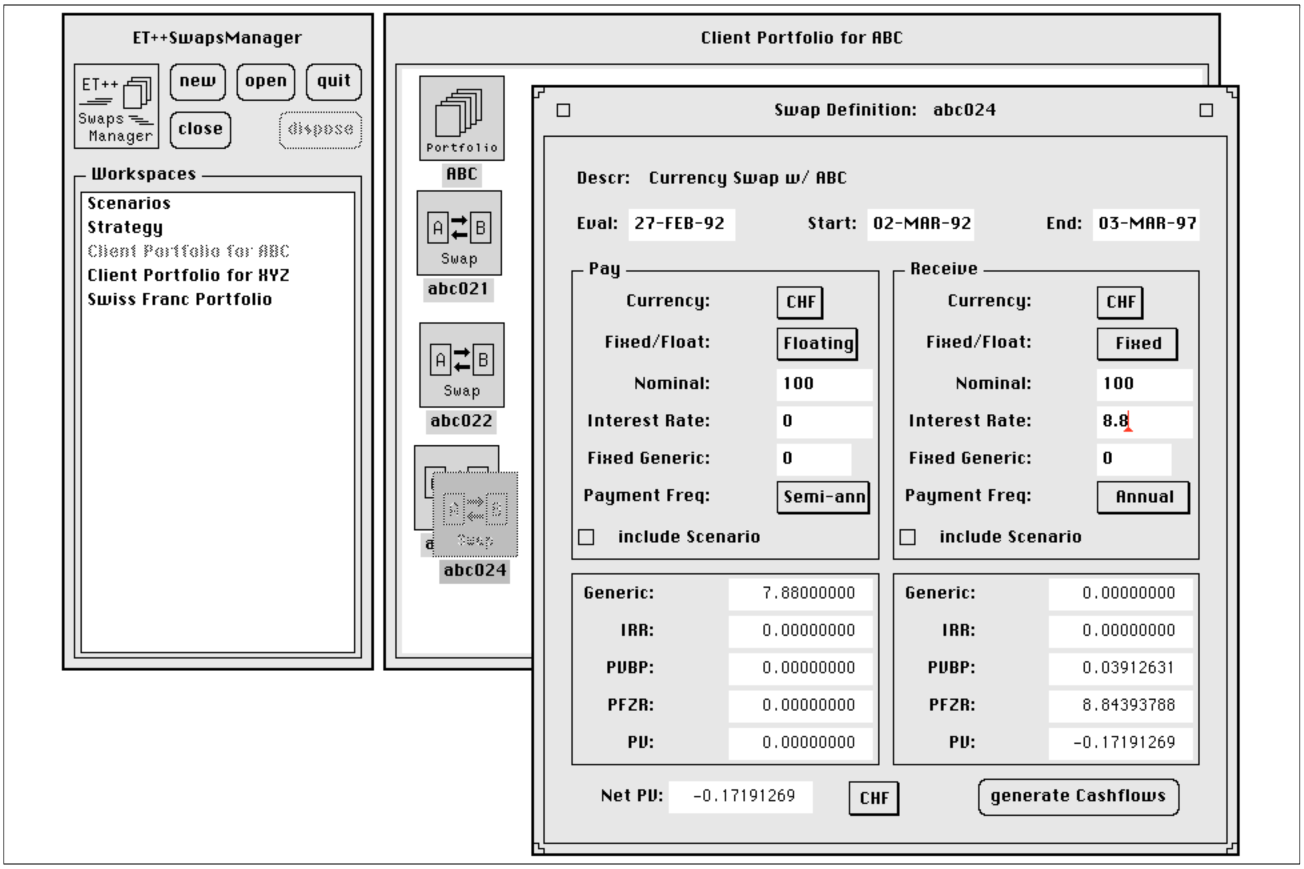

Figure 1: Definition of a new swap

One such metaphor is used in the domain specific desktop on which domain objects are represented as icons and relationships between the objects (swap, portfolio, etc.) can be established through drag-and-drop manipulations. Dragging an object over an other object automatically provides semantic feedback.

An other example is the yield curve editor which is a specialized drawing editor. It contains different tools for directly manipulating yields used in the cash flow valuation process.

The simultaneous and modeless presentation of data belonging to different objects in different windows avoids deep menu hierarchies. However, to let the user open windows freely has its own drawbacks. The screen can quickly become cluttered with windows and considerable time must be spent on rearranging windows.

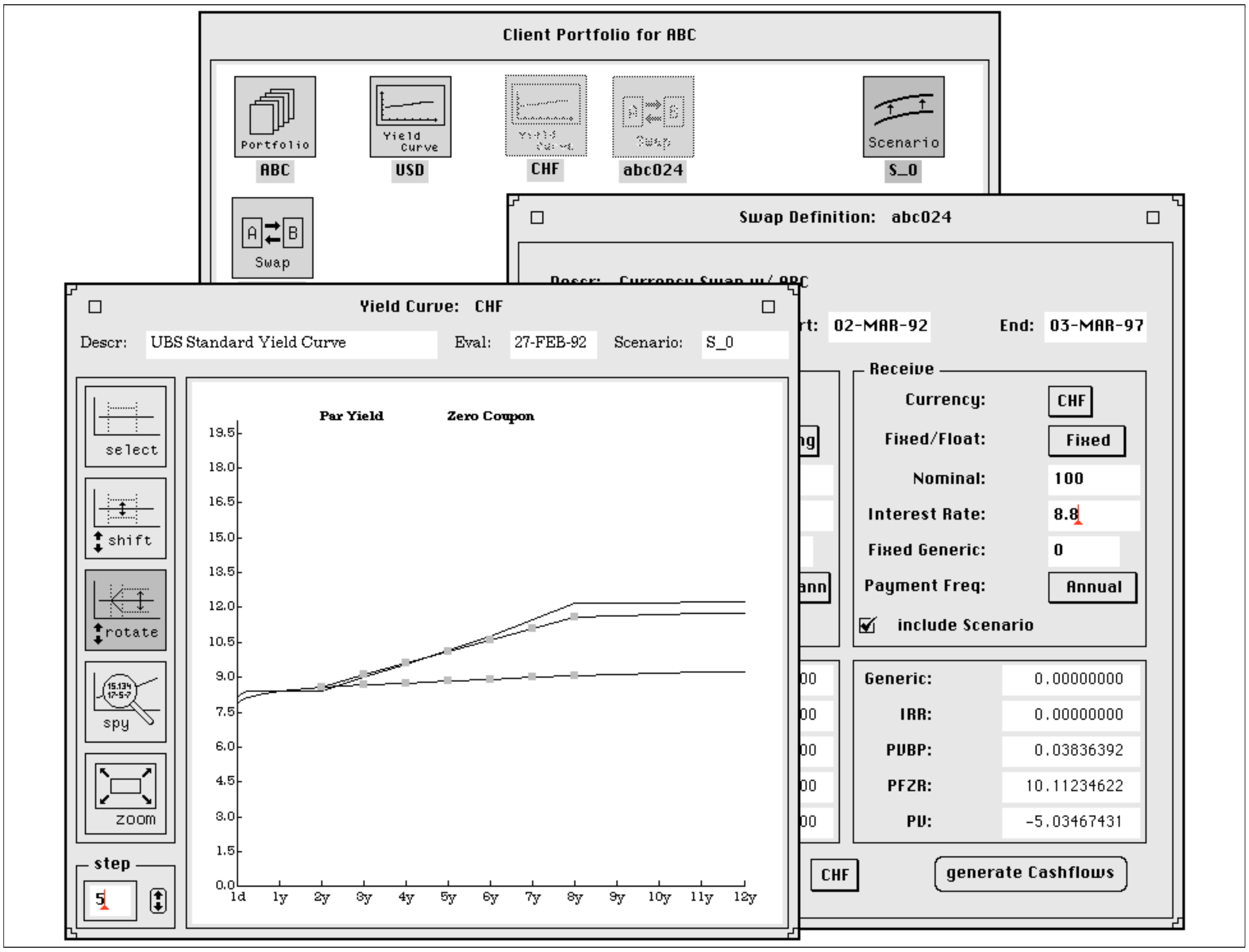

Figure 2: Creating a scenario through direct manipulation

To avoid this so-called window thrashing [Hen86] the ET++SwapsManager offers the idea of a workspace. A workspace organizes problem domain objects. The objects can be grouped according to individual work foci or tasks, and provides for fast switching between one focus of attention and an other. An example of such a focus is the set of swaps with the same client (see 3.4).

When the user switches workspaces all windows pertaining to the current workspace are closed. The new workspace is automatically installed just as it had been left the last time it was in use [Hen86].

Since workspaces only hold grouping and placement information they can be created and destroyed by the user at will. Domain objects are not affected since they exist separately from workspaces in a database. Domain objects may belong to multiple workspaces.

3.4 Using the ET++SwapsManager

To illustrate the ET++SwapsManager we present a sequence of interactions that comprises a complete example of a user task. In this example the user a) wants to create a new swap and b) add it to a portfolio holding all prior swaps with the same client. In addition, c) the trader wants to perform "what-if" analysis on the swap d) as well as on the entire portfolio. Traders have to be able to estimate the quantitative risk associated in the case of interest rate changes.

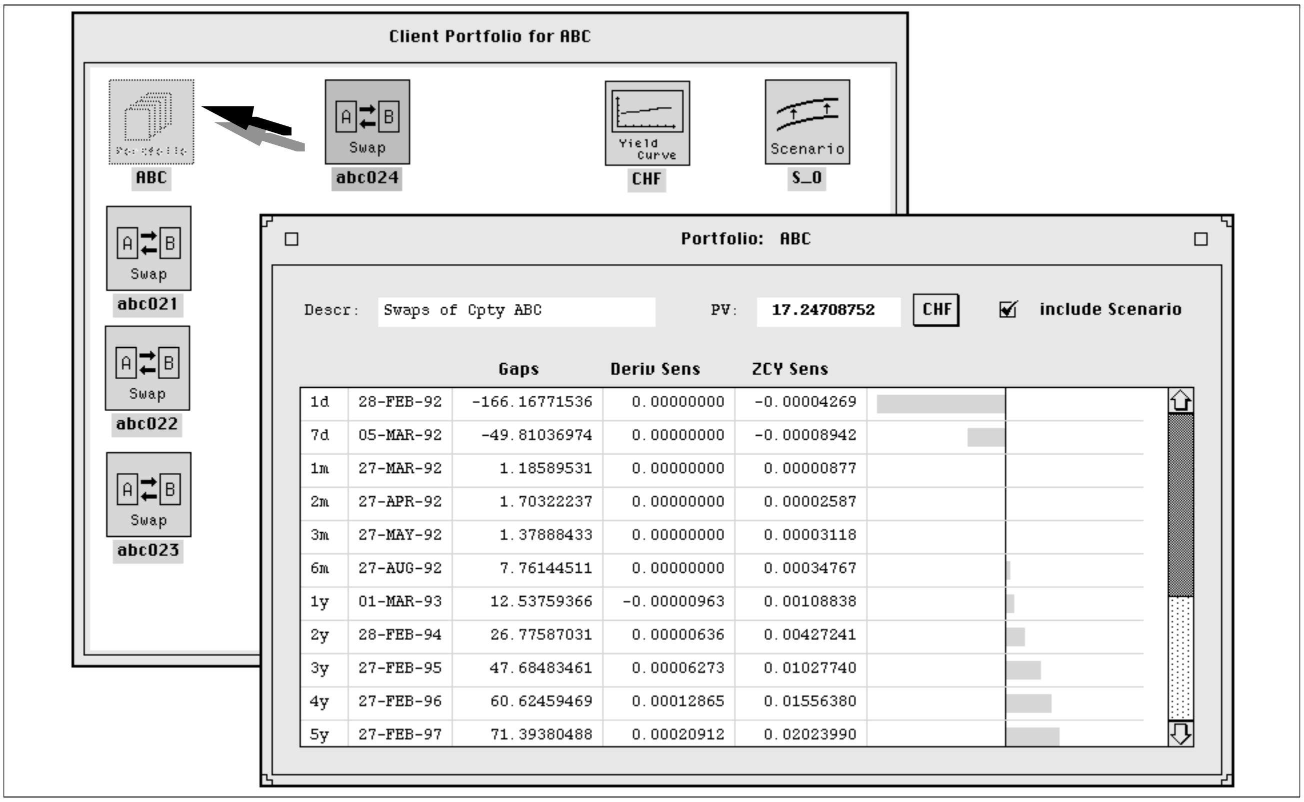

Figure 3: Adding the new swap to the client's portfolio

Figure 1 shows a snapshot of the system where the trader has opened a workspace which holds client ABC's swaps and a portfolio. A new swap (abc024) has already been created and an editor for defining the details of the swap has been opened. The trader can now fill in the details and then have the system generate the swap's cash flows.

In figure 2 the trader has opened the graphical editor for the Swiss franc yield curve (CHF) used to value the new swap. An interest rate (or yield) scenario can be created by directly manipulating the curve. A new scenario (S_0) is automatically added to the current workspace. This will invoke a recalculation of dependent data. Direct manipulation on the curve thus allows to immediately trace the effect of interest rate changes on the values of the two sides of the swap. A palette offers different tools for manipulation. Tools to select, rotate or shift a range of the curve. In addition there are tools to zoom into the yield curve or investigate points on the curve (spy).

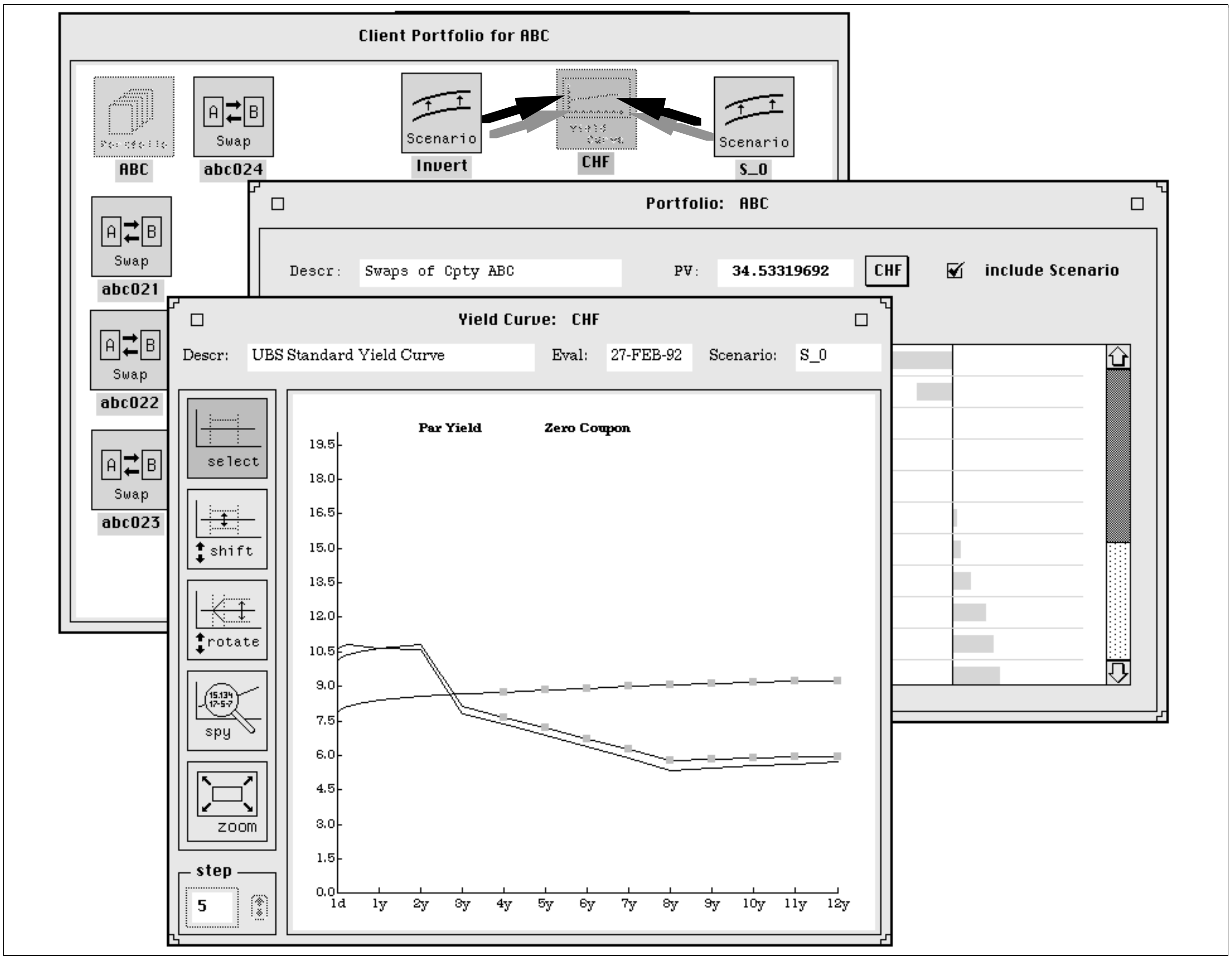

Figure 4: Valuation of the portfolio using scenarios

After the new swap has been analyzed by itself, then the collective behavior of all swaps with client ABC is investigated. To do this the swap is added to the portfolio of client ABC by dragging the swap's icon onto the portfolio's icon (ABC) and dropping it, as shown in figure 3. This operation demonstrates the "desktop"-like functionality offered in the workspace window. Figure 3 also shows a window of the portfolio's contents where its cash flows are aggregated to a manageable number of "buckets". Here, the user can immediately see the effect of adding the new swap to the portfolio.

Finally, in figure 4 the trader has opened another interest rate scenario (Invert) and, to study the portfolio's behavior, applies both scenarios to the relevant yield curve. Application of the scenarios is done through a drag-and-drop operation where a scenario icon is dragged over the yield curve icon. The trader is also free to directly manipulate the yield curve and thereby study the effects of interest rate changes on the portfolio in an ad hoc way.

3.5 Architecture

The architecture of the ET++SwapsManager is based on the application framework ET++. ET++ is a class library designed to provide a foundation for interactive graphic applications with consistent user interfaces following the direct manipulation principle. ET++ is implemented in C++ and runs under several operating and window systems. The application specific extensions to the ET++ class library are depicted in figure 5. Bold class names are abstract classes of ET++. The new, application specific, classes can be grouped according to their responsibilities. The main categories are:

- *Problem Domain Component*
  Several abstractions of the problem domain are modelled as separate classes. These classes encapsulate all domain specific data and functionality, e.g., all calculations take place in these classes.
- *Editor/Inspector Classes*
  These are object specific views onto data pertaining to domain objects. They provide the user interface functionality to browse and edit domain object.
- *Desktop Classes*
  These classes render domain objects as icons on the desktop. They provide the functionality to invoke inspectors and to relate objects through a drag-and-drop mechanism.
- *Manager Classes*
  Coordination of the interaction between domain objects and their respective representation is factored into these classes. In a sense, they provide the glue between domain classes and the user interface classes.
- *Activity Classes*
  These classes represent undo-able direct manipulation commands.

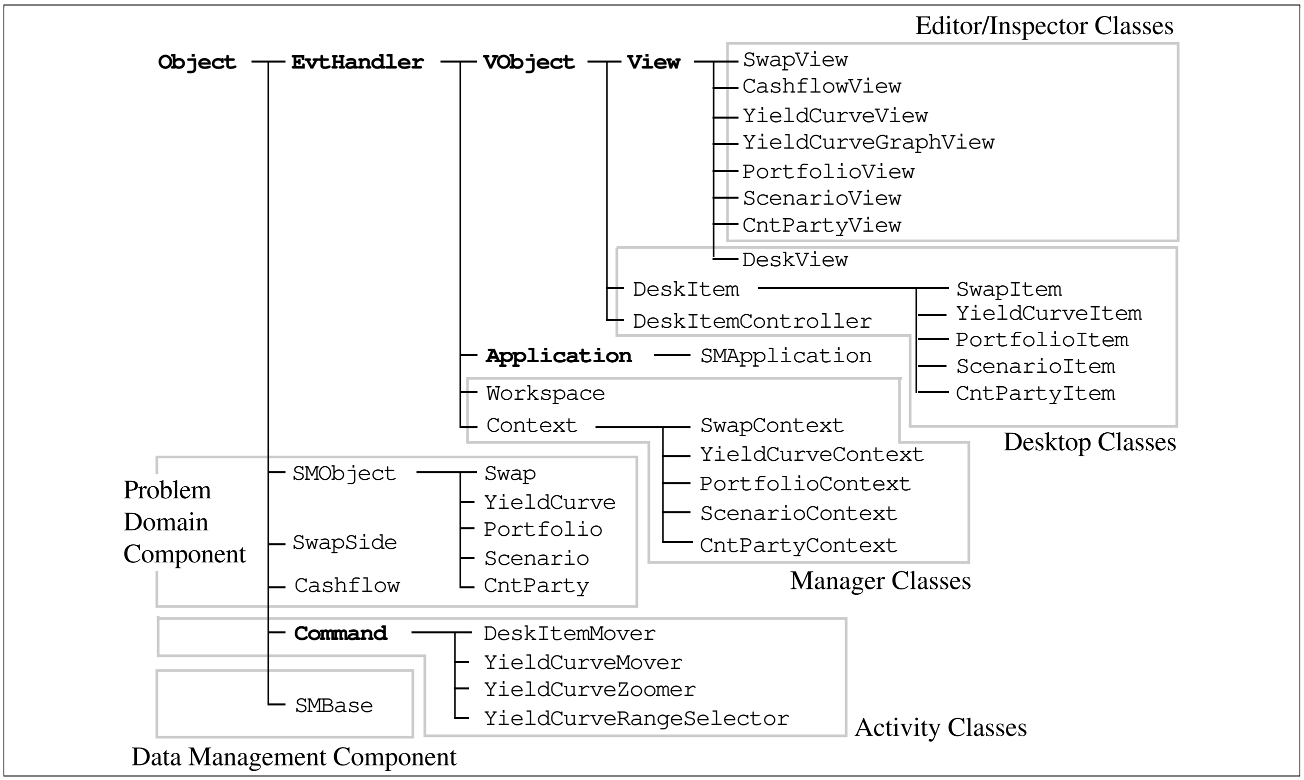

Figure 5: The ET++SwapsManager class hierarchy

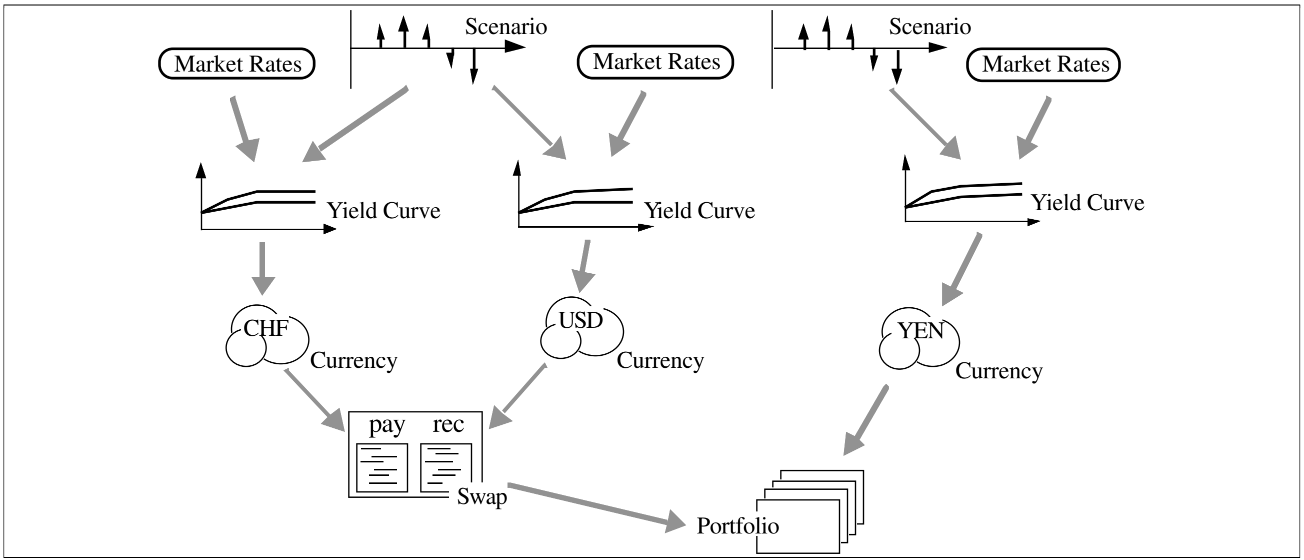

Figure 6: Object dependencies during valuation

A different view of the system architecture is given in figure 6 which shows the possible arrangement of domain objects and the dependencies between them. Since domain functionality is distributed among domain objects change propagation is used to ensure consistency. As an example, if a scenario is modified then any yield curve dependent on this scenario must recalculate its discount factors which in turn invalidates the present values of dependent swaps and portfolios. These will then also recalculate their value.

4 Creating the ET++SwapsManager

The overall design approach taken in the ET++SwapsManager project was guided by a strong conviction that new systems are grown, and not built [Bro87]. We set out with a half-page specification for the project with the understanding that the bulk of know how necessary to build the system would be acquired during multiple analysis/design iterations ("round-trip gestalt design" [Boo91]).

The lack of domain knowledge in financial engineering on the side of the developers and the diffuse notion on the side of the trading specialists how state-of-the-art user interface technology could support their work lead us to employ exploratory prototyping [Flo84]. The aim was threefold:

- Learn about the domain through building prototypes on the basis of informal communication and intuition, and then expose these prototypes to the critique of domain specialists
- Explore implementations of interaction techniques in the context (guidance and constraints) of the employed application framework
- Motivate domain specialists to take off precious time from trading.

The design iterations witnessed in the various prototypes incrementally caused a better understanding of the key domain abstractions, what they are and how they relate to each other. We started with a simple abstraction of a yield curve which was recognized early in the project and step by step evolved the system from there on. The first prototypes mainly concentrated on the representation and the direct manipulation of the yield curve. Intermediate prototypes then successively included definition and valuation of swaps and portfolios. Finally, the workspace and domain desktop metaphors integrated the domain objects.

5 Design Patterns

A powerful form of reuse is design reuse. A promising approach to support design reuse is the idea of design patterns. It was a goal of the ET++SwapsManager project to acquire experience with this idea.

5.1 What are Design Patterns

An important lesson we learned during the development of ET++ is that there are some design structures that emerge repeatedly [Gam91]. These structures typically improve a design in terms of flexibility, reusability and elegance.

Design patterns explicitly name and capture such structures. They describe the intent and organization of design structures at an abstract level. By doing so, a design pattern can be reused when designing new or improving existing systems. For this reason, we consider design patterns as a valuable addition to existing object-oriented design methods [Wir90, Boo91, Rum91]. The use of patterns to simplify and guide the design process has also found applicability in other domains, e.g., Alexander's work for architecture design [Ale77]. Design patterns are related to idioms as introduced by Coplien [Cop92]. A difference is that design patterns try to describe abstract design structures, idioms are more concrete solutions in the context of a specific programming language.

Design patterns occur at different levels of abstraction. At a lower level there are patterns to design individual classes, i.e. their interfaces and internals. Examples for such patterns are: template methods [Wir90], double dispatching [Heb90], or iterator classes [Str91]. Higher level design patterns are concerned with the design of collaborating objects or object teams[1]. These patterns focus on:

- Interaction among objects
- Distribution of responsibilities [Wir89, Bec89]
- Role of an object in a team
- Composition of objects.

The following is a list of some important design patterns that occur at this higher level. This list is intended to give an idea of the spirit of design patterns. For a comprehensive collection of patterns we refer to [Gam91]:

- *Strategy*
  In the strategy pattern an object delegates the implementation of some behaviour to a separate strategy object. In this way the implementation of the behaviour can be replaced by using a different strategy object.
- *Observer*
  Observer objects observe other objects. When they observe a change they react by

1 Such groups of objects are also called *behavioral compositions* [Hel90]

updating their own representation. Observer objects provide for synchronization, coordination or consistency constraints between objects.

- *Composer*
  A composer object provides for treating a composition of objects like a single object. A composer object is a mean to create recursive objects.
- *Forwarder*
  The idea behind the forwarder pattern is to create a hierarchy of responsibilities among objects. This is achieved by providing the infrastructure to link objects and to forward messages along these links. An object interested in a message will intercept it otherwise it forwards the message to the next object. An instance of a forwarder pattern is the event handling architecture in MacApp [App89] or ET++.
- *Bridge*
  A bridge object separates an abstraction from its implementation or representation.[2]
- *Manager*
  Manager objects manage the communication between objects. They mediate between objects and help to reduce the object coupling among objects. For example, a PrintManager object manages the communication between graphical objects and a Printer object.
- *Wrapper*
  Wrapper objects represent a specific property, which can be attached to other objects by composing them with the wrapper object. Various object-oriented user interface toolkits use this pattern to decorate graphical objects with different borders or to attach additional behaviour like scrolling.
- *Activity*
  The activity pattern distinguishes between activity creators, activity consumers, and activities themselves. The activity object knows how to perform some task and maintains the required context information. An activity object can be created by any object in the system and is then consumed or triggered by a specific activity consumer. The implementation of undoable commands in interactive applications typically uses this pattern (see Fig. 5).

After this general introduction to design patterns we will describe in the next section how they were helpful for creating the ET++SwapsManager.

---

2 Coplien [Cop92] uses the term *Envelope/Letter idiom* for these structures

5.2 Using Design Patterns in the ET++SwapsManager

Design patterns were reused in various places in the overall ET++SwapsManager architecture:

- The manager pattern was applied to decouple problem domain objects from the user interface. A special manager object (Context) takes care of organizing the communication between a problem domain object like a Swap and its visual representation (see Fig. 5).
- The observer pattern was used to trigger the recalculation after any changes in a domain object.
- Wrapper objects were created to attach additional application specific properties to existing graphical objects.

In addition, other patterns like forwarders, activities, abstract factories or strategies were incorporated into the application architecture by reusing the ET++ class library. These examples illustrate that design patterns can be considered as micro-architectures that contribute to an overall system architecture.

The abstract nature of design patterns has shown up as a major benefit. For example, the ET++ class library provides a class Document which is responsible for window management, data management, and command processing. This class accomplishes its responsibilities by mediating between data and window objects. Document is therefore an instance of a manager pattern, and mainly mediates between different objects. In the ET++SwapsManager application we needed a similar abstraction to connect the problem domain objects with user interface objects. The Document class itself was not reusable because it is was a very concrete implementation of such a manager object. However, the abstract design pattern underlying the implementation of Document was reused in the ET++SwapsManager.

Design patterns tend to motivate developers to go beyond concrete objects, i.e., they objectify concepts which are not immediately apparent as objects in the problem domain. The modeling of day-count conventions with a *strategy* object is an example for this.

We also noticed that design patterns have a potential to reduce the learning effort for a class library. Each class library has a certain design "culture". This design culture is characterized by the set of design patterns that were applied by the class library developers. A specific design pattern is typically reused by its developers in different places of the library. For this reason it becomes attractive to teach clients these patterns, especially novice clients. Once they are familiar with them they can reuse this understanding. Moreover, we have noticed that some of these design patterns emerge in different class libraries.

Clients should also benefit from thinking in design patterns. We noticed that this has the benevolent side-effect that they try to look at their own designs from a more abstract point of view, i.e. “how do my objects interact”, “what are their roles”, etc.

Finally, design patterns have proven to be very helpful in design reviews. They provided a common vocabulary to discuss a design. In addition, when there were some problem spots in the design, we noticed that design patterns helped to explore other design alternatives. We refer to this activity as *design pattern matching*.

To conclude, we do not think that the use of design patterns makes someone a better designer. The design process still remains iterative. In the ET++SwapsManager project design patterns mainly helped to reduce complexity by providing named abstractions for design structures. In addition, design patterns motivated reorganizations of the ET++SwapsManager's class hierarchy to improve its flexibility. This fact was not only noticed in our project but was also reported from other developers that were exposed to the idea of design patterns.

6 Evaluation

In this section we will discuss some results of the ET++SwapsManager project.

6.1 Development Effort

The pilot application was completed in seven person months. It is important to note that the ET++SwapsManager is not only a user-interface mock-up, but supports the functionality to price real swaps. The ET++SwapsManager was implemented by the first author while he was coached by the second. None of us had prior knowledge of swaps pricing and financial engineering. In comparison with other developments for the swaps trading department our project achieved a noticeably higher productivity.

6.2 Using a User Interface Framework

In the ET++SwapsManager project we experienced the following benefits from using a user interface framework:

- Support for prototyping of direct manipulation user interfaces through reusable abstractions and building blocks for graphical object editing.

- The user interface framework allows the reuse of an application's architecture and therefore provides the developer with architectural guidance.

One major obstacle of this reuse-oriented approach was the learning effort for the user interface framework. This problem was aggravated by the general difficulty to document frameworks in an adequate way. The comprehension problems could be alleviated by some ET++ specific browsing tools and by teaching the design patterns of the class library.

6.3 Reusability

In the ET++SwapsManager project we successfully reused an existing class library. We are well aware that the identification of reusable domain abstractions requires experience from more than one specific project in a problem domain. Therefore it was not surprising that the project was less successful in directly contributing new reusable classes. Nevertheless, through the project we have identified candidate abstractions for reusable components.

Candidate abstractions for reusable components in the user interface component are:

- Workspaces for task-oriented views onto trading data
- Graphical editor for yield curves
- Domain specific desktop which provides functionality to view and modify relationships among domain objects

We see a large potential for reusable components when we look at support for financial instruments in general. Possible reusable components from this wider problem domain are:

- Market conventions as separate strategy objects
- Cash flows and options which encapsulate how they arrive at their nominal values and thus allow application of generic algorithms in valuation.
- Portfolios as collections of abstract cash flows and options
- Yield curves which encapsulate market valuation system details

In our view, these abstractions will be promising components in a domain specific framework which would integrate these components on the level of abstract protocols.

7 Conclusion and Future Work

The ET++SwapsManager project shows potential to serve as a catalyst for object technology in the bank. It is already frequently cited as the standard reference for how user interface technology could be used in banking applications.

In the short term, emphasis will be put on extending the ET++SwapsManager to a fully operational swap valuation and pricing system. As a medium term goal the evolution of a financial engineering platform in the form of a domain specific application framework is considered. Essential steps in this evolution will be the production of a series of operational tools which will foster the necessary domain know how and will surface more powerful abstractions.

References

[Ale77] C. Alexander et al., *A Pattern Language*, Oxford University Press, New York, 1977. (5.1)

[App89] Apple Computer, *MacApp II Programmer's Manual*, Apple Computer, Inc., Cupertino, CA, 1989. (5.1)

[Bec89] K. Beck and W. Cunningham, "A Laboratory For Teaching Object-Oriented Thinking," In *OOPSLA'89 Conference Proceedings October 1-6, New Orleans, Louisiana)*, published as *OOPSLA'89, Special Issue of SIGPLAN Notices*, Vol. 24, No. 10, November 1989, pp. 1-6. (5.1)

[Boo91] G. Booch, *Object-oriented Design with Applications*, The Benjamin/Cummings Publishing Company, Reading, Mass., 1991. (4, 5.1)

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

[Cop92] J. O. Coplien, *Advanced C++ Programming Styles and Idioms*, Addison-Wesley, Reading, Mass., 1992. (5.1)

[Flo84] C. Floyd, "A Systematic Look at Prototyping," In *Approaches to Prototyping*, K. Budde, ed. Springer-Verlag, Berlin, 1984, pp. 1-18. (4)

[Gam90] E. Gamma and A. Weinand, "ET++ - A Portable C++ Class Library for a UNIX Environment," In *OOPSLA'90 Tutorial*, ACM, New York, October 1990,. (1)

[Gam91] E. Gamma, *Objektorientierte Software-Entwicklung am Beispiel von ET++: Klassenbibliothek, Werkzeuge, Design*, Dissertation, Universität Zürich, 1991. (5.1)

[Heb90] K. J. Hebel and R. E. Johnson, "Arithmetic and Double Dispatching in Smalltalk-80," *The Journal of Object-Oriented Programming*, Vol. 2, No. 6, January 1990, pp. 40-44. (5.1)

[Hel90] R. Helm, I. M. Holland, and D. Gangopadhyay, "Contracts: Specifying Behavioral Compositions in Object-Oriented Systems," In *OOPSLA'90 Conference Proceedings (October 21-25, Ottawa, Canada)*, published as *OOPSLA'90, Special Issue of SIGPLAN Notices*, Vol. 25, No. 10, October 1990, pp. 169-180. (5.1)

[Hen86] D. A. Henderson and S. K. Card, "Rooms: The Use of Multiple Virtual Workspaces to Reduce Space Contention in a Window–Based Graphical User Interface," *ACM Transactions on Graphics*, Vol. 5, No. 3, July 1986, pp. 211-243. (3.3)

[Kap90] K. R. Kapner and J. F. Marshall, *The Swaps Handbook - Swaps and Related Risk Management Instruments*, Institute of Finance, New York, 1990. (3.2)

[Rum91] J. Rumbaugh et al., *Object-Oriented Modelling and Design*, Prentice-Hall, Englewood Cliffs, New Jersey, 1991. (5.1)

[Str91] B. Stroustrup, *The C++ Programming Language, Second Edition*, Addison-Wesley, Reading, Mass., 1991. (5.1)

[Wei88] A. Weinand, E. Gamma, and R. Marty, "ET++ – An Object Oriented Application Framework in C++," In *OOPSLA'88 Conference Proceedings (September 25-30, San Diego, CA)*, published as *OOPSLA'88, Special Issue of SIGPLAN Notices*, Vol. 23, No. 11, November 1988, pp. 168-182. (1)

[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. (1)

[Wir89] R. Wirfs-Brock and B. Wilkerson, "Object-Oriented Design: A Responsibility-Driven Approach," In *OOPSLA'89 Conference Proceedings October 1-6, New Orleans, Louisiana)*, published as *OOPSLA'89, Special Issue of SIGPLAN Notices*, Vol. 24, No. 10, November 1989, pp. 71-77. (5.1)

[Wir90] R. Wirfs-Brock, B. Wilkerson, and L. Wiener, *Designing Object-Oriented Software*, Prentice-Hall, Englewood Cliffs, New Jersey, 1990. (5.1)