DEVELOPING A CUSTOM TRADING APPLICATION IN NEXTSTEP:

A CASE STUDY

Written by Gregory H. Anderson
President
Anderson Financial Systems Inc.
909 Sumneytown Pike, Suite 207
Springhouse, PA 19477
(215) 653 0911 voice
(215) 653 0711 fax
Email: greg@afs.com

CONTENTS

ABSTRACT

The afs:TRADE Trading Management System is a custom application designed for the First National Bank of Chicago to manage its commercial paper trading and underwriting operations. The system provides on-line, real time access to all of the information needed by traders and salespeople as they make decisions throughout the trading day. As the sole data entry point for all securities transactions, it streamlines the flow of information between the front and back offices by printing a copy of each ticket and transmitting trade information to the mainframe accounting system.

The afs:TRADE system runs in a client/server environment, with a Sun Sparc 2 database server and NeXT workstation clients. The front end application was designed and implemented with native NEXTSTEP tools in Objective C. Printing and communication daemons were written in ANSI C, because they run on both Sun and NeXT hardware. The application has been running in production since April 1992, and was recently upgraded to handle additional investment types. In addition, AFS has implemented distinct versions of the system for three new customers over the past year. High code reuse through subclassing and minimal maintenance requirements in those projects have proved the value of an object-oriented approach in custom applications.

STATEMENT OF THE PROBLEM

First Chicago underwrites commercial paper (a form of short-term financing similar to a credit line) for its best corporate clients. Trading proceeds at a brisk pace, because all activity must be completed by noon. Inventory changes constantly as paper gets bought and sold, making it difficult for the sales staff to stay informed of what is currently available. Due to the fast pace, mistakes may occur as paper trade tickets pass through multiple areas before being recorded electronically.

First Chicago evaluated Anderson Financial Systems' existing DOS system (written in compiled Microsoft BASIC) and decided that it was functionally complete for the task, but not in accordance with the bank's long term goals for operating environment, ease of use, ease of learning, and program maintenance. As a result, First Chicago contracted AFS to rewrite the system in Objective-C on NeXT workstations.

What benefits were expected?

- *More activity without increased staff* - This met an existing business goal to increase the transaction volume of the department.

- *More selling opportunities* - Salespeople would have better customer information at their fingertips, leading to more intelligent selling approaches and better customer relationships. Also, since salespeople would spend less time on paperwork, they would have more time for prospecting.

- *Fewer data entry errors* - Traders and salespeople previously hand-wrote buy and sell orders, which were later keyed into the mainframe by data entry clerks. The new system would provide a simple, interactive electronic form to enter orders, which would then be transmitted directly to the mainframe.

- *Fast development* - AFS had never programmed a graphical user interface before. It was expected that object-oriented encapsulation of the difficult details would lead to more rapid prototyping and development.

What aspects demanded an object-oriented approach?

- AFS has written almost 50 custom trading systems over the past 10 years. All obey the 80/20 rule: 80% of the code is shared among customers, 20% is different at each site. Unsurprisingly, the last 20% requires 80% of the maintenance effort. Typical customizing involves unique screen layouts and report formats, along with switches to drive customer-specific behaviors.

- For this project, AFS created a set of classes which encapsulate standard financial trading functions. These base classes are then subclassed for each customer's individual needs. This technique makes maintenance of both standard and customized code far easier.

Why wouldn't conventional approaches have solved the problem?

- Conventional approaches have been tried and found cumbersome over 10 years of experience. They solve the essential application requirements, but have been difficult and expensive to maintain. Bug fixes and upgrades, in particular, are difficult to monitor and distribute in conventional software libraries.

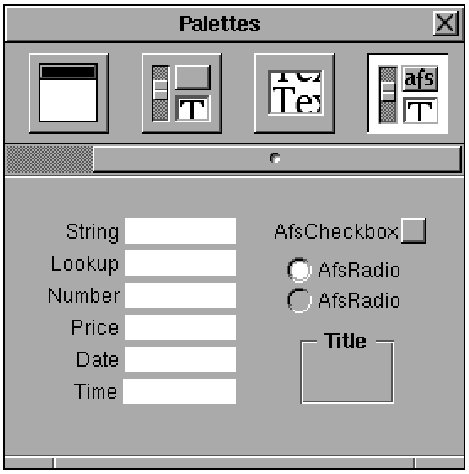
- User involvement is another key requirement for successful implementation, especially in the design phase. Otherwise, the design may undergo numerous revisions before it is approved for implementation. The NeXT InterfaceBuilder tool not only lets programmers sit down with users to create prototypes, it also allows them to test the prototype without compiling and get immediate feedback under "real life" circumstances. The benefit is that users gain a sense of ownership of the system, because it is not simply imposed on them. This involvement would not have been possible with a conventional technology.

THE SOLUTION

DEVELOPMENT ENVIRONMENT

The standard NEXTSTEP development tools were used for compiling and debugging Objective-C and ANSI C source code. Most of these tools are based on the gnu 'gcc' compiler and the gnu 'gdb' debugger. The 3.0 release of NEXTSTEP provides a ProjectBuilder tool and integrated Edit (a NeXT-supplied structured text editor) facilities to manage the interactions between these processes.

All user interface prototyping, design, development, and maintenance was performed with the NeXT InterfaceBuilder application, which allows the programmer to set the launch-time attributes of objects and connections between them. This information is archived into "nib" files, which are loaded when the application launches. (Object archive files are called "nibs" because that is their file extension.)



InterfaceBuilder is a highly extensible tool; it is not restricted to managing only the objects in the NeXT-supplied AppKit. During the project life cycle, AFS created numerous custom objects and built Inspectors to set their unique attributes inside InterfaceBuilder.

Since only two programmers are responsible for class maintenance, Unix access security is sufficient for storage management and version control. Changes to class interfaces require the approval of a specific manager. Once code has been certified and tested, changes must be marked and signed clearly in the source code, and nothing may be physically removed. Most user-specific changes are performed in subclasses for which only one programmer has responsibility.

Synopsis of tools and hardware:

Interface hardware - NeXT workstation
Interface software - NEXTSTEP

Core code hardware - NeXT workstation
Core code software - Objective-C and ANSI C

Database hardware - Sun Sparc 2
Database software - Faircom c-tree client/server

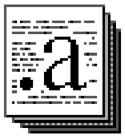
Development hardware - NeXT workstation
Development software - NEXTSTEP InterfaceBuilder application, AppKit classes

CLASS OVERVIEW

AFS expected to reuse this code for other customers. Therefore, the system was designed in four tiers to differentiate between core functionality and customization. Experience with new customers has shown that the lines between these tiers may slide, but that fact highlights a benefit of the object-oriented approach: With good encapsulation practices, it does not take much effort or retesting to move functionality between classes.

Here is a breakdown of the four functional tiers:

1) afsclib - The lowest tier is a library of ANSI C functions for database access, financial calculations, and event handling. These are written in straight ANSI C because they are needed in non-OO programs, such as daemons which run on the file server and transmit information up the communications link. Most low-level functions are wrapped inside afskit classes (see next paragraph) for application development.



2) afskit - The second tier is a set of general-purpose extensions to the standard NEXTSTEP AppKit classes. This includes a complete forms-based application development framework, an access control mechanism for securing specific functions and views (because different types of users have different access privileges), and user-requested enhancements (such as keyboard-enabled checkboxes). A hierarchical chart of the afskit classes and a list with brief descriptions is attached at the end of this document. These classes are universally reusable. AFS resells them to other NeXT developers as "ObjectWare" and uses them in-house to build applications in other functional domains.

3) tradekit - The third tier is the core set of trading classes. These classes define the 80% of functionality that is shared by all customer sites, plus the most common implementation of the remaining 20%. The implication of this design is that by compiling and linking tradekit, a minimal "shrink-wrapped" trading system can be produced. Only methods that differ from the most typical implementation must be subclassed for individual customers. Although AFS usually performs the customizing, any skilled programmer could use them to build customized trading applications after a short training period. A hierarchical chart of the tradekit classes and a list with brief descriptions is attached at the end of this document.

4) afstrade - The final tier is the finished work product: A custom application with a unique user interface and subclasses of the tradekit classes as required. At this writing, six variations exist at four customer sites. They are described in the "Benefits" section, with details about how much subclassing has been performed at each location.

OBJECT-ORIENTED FEATURES USED

Except at the very lowest levels, all functionality is delivered through objects. User interface objects are typically created in InterfaceBuilder and archived into "nib" files. At launch time, the nib objects are reloaded, reinstantiated, and reconnected. Non-UI objects, such as database entities, are instantiated at runtime and connected to the appropriate UI.

The system makes extensive use of inheritance, polymorphism, and messaging to perform its work. Virtual methods provide the basis for customizing. If a customer needs to replace or enhance the system's default behavior, a specific method is subclassed. In some cases, the override calls back to its super class's implementation of the method; in other cases, it completely replaces the existing implementation.

The most interesting way to describe the object-oriented features of the system is to take a walk down the main menu. All of the important methods (save:, edit:, find:, etc.) are polymorphic. The main menu is wired to send generic messages to the first object that can respond to them. Members of the NeXT Responder class keep a chain of objects that can respond to events, so events show up automatically at the active window for processing. This design leads to a shallow menu structure, because tasks need not be repeated in different contexts. As a result, the main menu is easy to navigate and learn. Most menu operations have an associated command key, which allows experienced users to trigger specific actions with a single keystroke.

Info Submenu

Help

To get on-line help for a specific object, the user simply clicks the mouse on it while holding down the Control key. This action sends out a generic "Help!" message. The Help Panel responds and asks the sending object some questions about itself (class, title, owner, message sent when activated) to figure out the best kind of context-sensitive help to provide. The programmer does not need to create specific connections between individual objects and the help files, which reduces maintenance requirements. Abstract implementation also allows users to extend the help system at their convenience and guarantee the information will be found when it is needed.

Preferences

Users can set operational preferences, several of which are mapped dynamically at runtime. For example, rather than providing its own report rendering facility, the application creates Rich Text Format (RTF) files and passes them to a user-specified rendering program. All NEXTSTEP applications built with the AppKit classes are guaranteed to respond to the message "openFile:". In just three statements, the custom application opens a port, asks the global WorkspaceManager to attach it to the specified rendering application (which will launch itself, if necessary), then asks the rendering application to open the file that was just created. Inter-application dispatch is so easy that it is never worth reinventing the wheel inside a custom application, if the same task can be performed by an existing application.

Forms Submenu

The Forms submenu provides access to the list of available forms, plus a set of options for managing the information on those forms. All forms derive from the same base class; therefore, they respond to the same messaging protocol. For example, here is the class hierarchy for the trade-processing form:

Object - A NeXT-supplied root class, implements global object behaviors.

Responder - A NeXT-supplied abstract base class for objects that respond to events (keyboard, mouse, timer, etc.).

Window - A NeXT-supplied class with basic Window methods.

AfsWindow - An afskit class with first-level window extensions, such as initialization, defaults management (preserve the window's size and position between runs), edit status, programmatic resizing and placement, command key processing, and connecting the user interface to database fields.

AfsForm - An afskit class which defines a protocol and standard implementation for database entity management: new, delete, edit, save, revert, duplicate, fetch, find, find previous, find next, set/clear locks. Also defines a delegation protocol (discussed below) for important application states, such as formWillSave:, formWillDelete:, formDidChangeEditMode:, etc.

TicketForm - A tradekit class which defines a protocol and standard implementation for all methods required to manage the ticketing process. The TicketForm class also redefines some of the AfsForm methods where special processing is required.

FNBCTicketForm - A custom subclass that implements many of the delegate methods and overrides several TicketForm methods where site-specific processing is required.

In most cases, site-specific override methods supplement the standard methods, rather than completely replacing them. In other cases, delegation is sufficient. Delegation allows the custom parts of the system to participate in critical actions of an object without writing a complete subclass. Delegates are especially useful for methods that are almost always overridden, such as validating fields prior to saving. It is still necessary to write a custom class, but often that custom class can implement delegate methods for all forms in the system. At runtime, the system uses the method dispatcher to look up whether a delegate exists, and if so, whether it can respond to a particular message, such as "formWillSave." This design has the benefit of consolidating all validations into one object, which simplifies code maintenance.

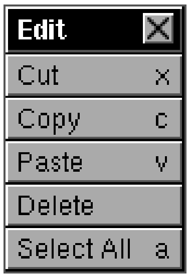
Persistent objects are implemented through a custom set of object wrappers. The selected client/server database engine—Faircom's c-tree product—is an ISAM server, not an object-oriented database. To compensate for that limitation, at runtime the system matches the names of the database fields (columns) and the user interface for the form that will be presenting the data. These object pairs need not be connected programmatically, which saves program code and makes it trivial to add fields as the system grows. Quite literally, the programmer adds a field to the database, adds a correctly-named instance variable to the form, and restarts the application. By the time data makes it to the form, it has been recomposed into objects and attached to the correct fields.

Find submenu

The Find submenu implements the section of the AfsForm protocol that relates to retrieving specific records. As on the Forms submenu, these operations use polymorphism to send messages to the first object that can respond, usually the active form's Find Panel. Each form has a custom Find Panel that allows the user to specify a query for its class of managed objects. When the user asks to "search:" using a specific query set, the Find Panel calls a custom Scan Panel to locate any matching records and presents them in a browser for selection.

Scan Panels are flexible enough to look for outside data sources. In a system created for another customer, the Scan Panel that looks for securities was extended to query a third-party dial-up database. A special client/server process was created around the NeXT Speaker/Listener classes (wrappers for 'rpc' remote procedure calls) in less than a day. As a result, users can query the external database just as easily, and in the same manner, as the native database.

Edit submenu



The Edit submenu supports NeXT-supplied polymorphic behavior for cutting, pasting, copying, and deleting text in the user interface controls. These messages are sent to the "first responder," in this case the object being edited.

Reports submenu

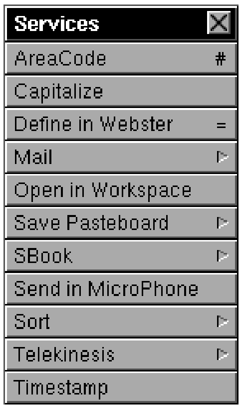
The reporting system provides an excellent example of dynamic class/method generation through run-time loading and binding. Reports are among the most frequently customized features, and new reports are often added to a certified system. The goal was to integrate reporting into the application, but make it easy to modify without re-certifying the mainline code. It was also expected that the customer's programming staff might want to add reports without AFS assistance, so the mechanism had to be easy to learn and use.

Each report is a custom subclass of a base class that knows how to open a report file, generate an RTF document, and access the user-specified rendering program. All report objects are stored in a special subdirectory whose contents can be altered freely. At runtime, the application looks in the report directory and loads whatever it finds.

Each report consists of dynamically linkable object code with the specific output formats, plus a "nib" file to capture any user-supplied data constraints (date range, specific customer, etc.). A class method supplies the menu path to access the report. The application uses this information to build the Report submenu hierarchy dynamically at runtime.

When the user walks down the menu path and picks a report, its data capture UI is swapped into a resizable Report Definition Panel. This panel runs a modal dialogue to get the required information, then generates the report. After report generation, the resulting file is handed off to the user-specified rendering program (such as WordPerfect or Frame), which displays the output and allows further manipulation of the contents. From the programmer's standpoint, a maximum of three methods needs to be overridden, and usually only one.

Services submenu



The Services submenu is maintained by the NeXT WorkspaceManager application. In the "nib" file, this submenu is empty. At runtime, the WorkspaceManager builds a list of all applications that provide services to other applications. For example, in any TextField, if the user highlights a word and presses 'Command' plus the '=' key, the Webster Dictionary application will provide a definition of the highlighted word. The custom application doesn't need to do anything to benefit from such inter-application services, except provide an empty submenu to hold them.

Print...

Printing, faxing, and page layout panels are built right into the NeXT AppKit classes. As with Services, custom applications simply need to provide menu options to access these features. Total time spent providing sophisticated printing and faxing capabilities for the custom application: 1 minute.

Utilities submenu

Access Control

Access control is a critical part of the application, because trading activity is sent directly to the mainframe accounting system. Since users come and go, and application functionality may change unexpectedly, access control is built into the application in a way that never requires programmer maintenance.

Forms, Boxes, and Buttons can declare in InterfaceBuilder whether they want access control, and if so, the default read/write/execute state to use. For example, if a user does not have execute access to the Edit button on the Customer Form, he will not be allowed to edit customer records. Rather than securing individual UI objects, Boxes are used as securable containers, because experience has shown that usually groups of things are protected. Many access control methods are implemented as "categories," an object-oriented feature unique to Objective-C. Categories replace methods in or add methods to a base class, without adding instance variables. This eliminates the need to subclass every object that receives access control messages.

At runtime, the system queries all objects about their desired access control state and takes the appropriate steps to protect them—in some cases, removing them from the system. As new objects are created, they automatically get added to the list of securable things, and they run in their default state until the security administrator overrides them for specific users. To set access control states, the administrator simply launches the application. It recognizes his or her special role and provides a list of users and access privileges for editing. That's all there is to it!

Lookup Tables

Lookup tables used to validate fields with restricted contents. At runtime, when an AfsLookupField user interface object needs validation, the AfsLookupTable class (subclassed from the NeXT Storage class) loads a user-maintained flat file and performs the validation. The AfsLookupTable class also owns a simple window on which the contents of a table can be edited. All of these features are added at runtime if the application provides a menu option to support them. The custom application does not have to perform any additional work to get the feature.

PROJECT LIFE CYCLE

ANALYSIS AND DESIGN

Anderson Financial Systems has 10 years of experience building trading systems, and a similar system had already been implemented in Microsoft QuickBASIC under MS-DOS, so the task analysis was almost complete at the start of the project. For similar reasons, the database schema needed only a few modifications.

Program design, on the other hand, presented major challenges. This was AFS's first project using object-oriented techniques, and AFS had never used the NEXTSTEP operating system before. Fortunately, one of the two programmers assigned to the project had prior experience in C++ on Sun workstations, and the other programmer had designed the original BASIC system. The combination of these two skill sets led to an initial plan in about a month.

The most difficult task at this stage was grasping all of the resources delivered with NEXTSTEP. The main set of AppKit classes is not so much *hard* as it is *big*, and it took a while to figure out the best approaches. The most difficult initial decision was whether to use separate windows and controllers. Since the application is forms-based, it was decided to extend the Window and Panel classes with enough intelligence to manage the information contained on them, then subclass the base classes for specific entities. This design breaks the popular model-view-controller paradigm, but it cuts in half the number of classes that need to be maintained. The concept of delegation allows other objects to intercede with specialized behaviors as necessary, so this is a fair compromise.

After initial versions of the data entry forms had been created in InterfaceBuilder with standard NeXT user interface objects, the visual prototype was taken to the users for evaluation. They agreed with the general model: A set of forms attached to a central application spine, each managing its own entity (tickets, customers, etc.). Each form would own a Find Panel and a Scan Panel specific to its entity, shareable with other forms to support database joins.

Total work time in design and analysis period: 2 man months. Elapsed time: 1 month.

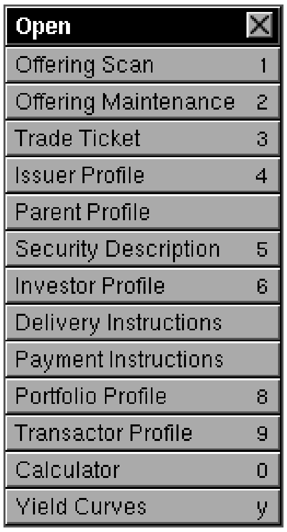
DEVELOPMENT

With the user-approved interface in hand, development began. In order to minimize wasted development effort, it was decided to implement one complete form and get user feedback. One programmer wrote the core code for database access and the financial calculation library, because these functions were well known and not subject to user modification. The other programmer created an initial set of foundation subclasses on top of the standard NeXT AppKit objects, and then the complete set of classes required for issuer management: IssuerForm, IssuerFindPanel, and IssuerScanPanel classes. This sub-stage was completed in approximately four weeks.

It will not surprise any experienced reader that numerous problems were identified when this prototype was shown to the users. Many of the objections concerned native NeXT object behavior. For example, standard NeXT checkboxes and radio buttons can only be manipulated by mouse clicks. Due to limited desk space, users stated that all mouse operations, including task invocation from the main menu, had to provide keyboard alternatives. Also, to minimize accidental data entry changes, users requested that editing be a modal process, with an intentional action to put a form into an editable state. Back to the drawing board.

For the first time, the benefits of object-oriented programming became obvious. Even though NeXT does not supply source code for its objects, AFS was able to override only three methods in each of the checkbox and radio button classes to supply the desired keyboard behavior! Modal editing was implemented in the AfsWindow subclass so that all forms would inherit the capability.

With user approval of the new implementation, several additional forms were coded to demonstrate object interaction. Of particular interest was the way forms would share their Find and Scan Panels. After several iterations, a general rule was adopted: If the user typed a few characters in a text field whose value came from a joined entity, that entity's form would pass the field value to its Scan Panel, which would "stand in" over the active form and provide validation services. If the user clicked the text field's label, the joined entity's form would loan its Find Panel to the active form. This architecture eliminates the need for multiple instances of Find Panels.



Two months after the IssuerForm had been critiqued for the first time, all major forms were complete. A four-workstation test environment was established in the MIS department. Two traders and two salespeople spent a day exercising the system in its entirety. New problems were uncovered, but users expressed generally favorable impressions about the quality of the application and the progress being made. Another month of development addressed these problems and put the system in a state where it could be deployed to the entire trading desk for functional testing. While one programmer finished this work and created an initial set of reports, the other wrote a daemon process to watch the real-time event stream and print purchase and sale tickets as they were confirmed.

Total work time in development period: 8 man months. Elapsed time: 4 months.

DEPLOYMENT

While development was underway, the trading desks were physically reconstructed to hold the large color workstation monitors, and users were given basic NEXTSTEP training. (AFS did not perform these tasks.) Since the users know their business and had been involved in the design process, they needed minimal training to start testing. Also, the custom application closely follows NEXTSTEP interface guidelines. Having spent time learning other NeXT applications while the custom application was in development, users had little trouble operating similar features.

The first parallel test was a disaster. A reasonable stress test had been performed with a few workstations the previous week, but it had not uncovered several serious flaws in the low-level interface to the database server. For the first time, users could review customer data that had been downloaded from the mainframe accounting system, and it was universally agreed that the data was in bad shape. (Since users had never seen or used this data directly, they had no reason or ability to keep it up to date.) For example, many customers had the wrong salesperson assigned. This caused insurmountable validation problems when salespeople tried to find and confirm their own trades. Parallel testing was stopped for over a month while these issues were resolved.

A new round of parallel testing with complicated trades uncovered some processing requirements that had gone unstated and unrecognized. Several of these new requirements led to database schema changes, and again the benefits of object-oriented programming were apparent. Never before had AFS modified core designs so extensively at such a late stage without re-certifying the entire system. In this case, the benefits of object encapsulation allowed testing to consider the only the affected classes and assume (correctly, as it turned out) that nothing else stopped working. In some cases, changes were turned around overnight.

Parallel testing continued on and off for several months, because the acceptance plan required a full week of clean processing, and the clock restarted after any problem was discovered. (In retrospect, this requirement could have been relaxed, but this was the first in-house system to completely replace the manual ticket-writing process.) Even though the system was not officially "live" during this period, it was used on a daily basis and resynchronized weekly.

Most of the problems during the end of the deployment period were caused by the one functional area that could not be completely encapsulated: database access. The low-level client/server routines supplied by the vendor did not provide multiple contexts for each open file, so the contents of each record buffer was public. Unanticipated interactions between objects sometimes left the record buffers in a dirty state when the system was expecting them to stay clean. In a way, even this deficiency shows the value of object-oriented programming, because the one area that was not fully encapsulated proved the hardest to debug and caused the most lingering problems.

Total work time in deployment period: 4 man months. Elapsed time: 5 months.

MAINTENANCE

At this writing, the system has been in production for over a year. It has been patched three times to correct undiscovered bugs, and modified twice to support new specifications. For a mission-critical system of its complexity, maintenance requirements have been minimal. User satisfaction is high.

Maintenance has always been the most difficult part of the custom programming process, because there was poor support for segregating core functionality from customized areas. Object orientation solved this problem in an elegant way that is readily apparent in the subclasses, because only custom behaviors appear there. AFS has also found it much easier to train new programmers. With only 22,000 executable statements in the entire system, there's not that much to learn. By comparison, the original system contained almost 100,000 lines of code, much of which was concerned with text-based user interface details.

All enhancement requests were held for the second version, which entered parallel testing in late April 1993. This version adds support for several new instrument types and implements many improved object classes.

BENEFITS

Many benefits have been stated throughout this discussion. Here is a recap of the highlights.

USER BENEFITS

- *Easy to learn and use* - Since users participated in the design, they already knew how to operate the application by the time it was deployed. The NEXTSTEP operating system provides an elegant and intuitive GUI, and AFS-supplied enhancements for specific objects made it even friendlier.

- *Increased business volume* - On a recent day, total activity exceeded $1 billion, yet all processing was finished by 11:00. The traders estimated that completing so many orders under the old manual system would have taken at least two hours longer. Also, the traders estimate that they can each manage twice as many issuers as before, which provides plenty of room for growth without increased staff.

- *Higher-quality information* - The customer file on the mainframe accounting system was cleaned up as a result of this project. In addition, salespeople can enter local information of interest, such as a customer's buying preferences. The system automatically takes these preferences into account when it shows the salesperson items which may be interesting to a specific customer.

- *Fewer data entry errors* - Traders and salespeople immediately notice the few errors they introduce accidentally, because the incorrect information shows up on their screens right away.

PROGRAMMING BENEFITS

- *Rational, obvious organization of code functionality* - It is much easier to train new programmers to work on the object-oriented version of the afs:TRADE system. For three programmers who joined AFS during the last year, the typical training period was less than a month. In the past, the minimum training period was three months.

- *Rapid prototyping, development, and deployment* - In this project, total work time was 13 man months, with an elapsed time of 10 months. Keep in mind this was AFS's first experience with NEXTSTEP, Objective-C, and the Faircom database engine. With that experience in hand, it is estimated that a forms-based system in a different functional domain (using afskit, but constructing the equivalent of tradekit in the new area) could be built in less than four months.

- *Minimal maintenance* - fewer than 10 incidents in the first year.

- *Dramatically fewer lines of code* - 22,000 vs. almost 100,000, little of which is concerned with user interface details.

- *Painless refinement* - With proper encapsulation, the line between standard and custom functionality can shift frequently, with little or no code recertification.

- *High code reuse* - Since delivering the First Chicago system, AFS has implemented five other systems for three new customers. By conventional standards, the results are astonishing. Here are the total classes and maintainable statements in each of these systems:

	Mainline Code		*Reports*	
	Classes	*Statements*	*Classes*	*Statements*
Shared Classes and Libraries				
afsclib - base functions	8	2,378	none	
afskit - foundation classes	51	6,857	3	880
tradekit - trading classes	77	14,464	none	
repotradekit - repo extensions	13	3,533	none	
Average statements/shared class		182		293
Customized Systems				
First Chicago money market	5	343	19	2,507
Citicorp money market	9	528	6	1,133
First Chicago repo trading	5	184	6	675
Citicorp repo trading	8	936	6	605
HSE municipal bond trading	12	1,656	under development	
Soros portfolio management	10	1,405	under development	
Average custom classes/system	8	842	9	1,230
Average statements/subclass		105		137

Now that's reduced program maintenance! (By the way, "maintainable statements" are defined as all lines in header files and code modules that begin with '#' or contain a ';'.) When aggregating statement counts, it should be noted that each customized system uses a subset (approximately 75%) of the afskit and tradekit classes, which are grouped on this chart for convenience. Estimated total statements for the First Chicago money market system:

Subdivision	*Statements*	*% of total*
afsclib (100% usage)	2,378	11%
afskit - (75%)	5,143	23%
afskit reporting (100%)	880	4%
tradekit - (75%)	10,848	49%
Custom subclasses, mainline	343	2% (!!!)
Custom subclasses, reports	2,507	11%
TOTAL	22,099	

An impressive 87% of the total statements are shared with other customers' implementations, and only 2% of the mainline code is unique. This is a significant improvement over the conventional approach. It is hoped that the next version of the system will include better reporting tools to reduce that section to less than 5%.

Of course, the user interface is different for each system, but these differences are captured in the object archive files. Since UI attributes are set through direct manipulation in the InterfaceBuilder application, and custom objects are just as easy to manipulate as standard ones, creation and maintenance times are minimal.

The HSE and Soros systems are larger than the others because they are still under development. For safety reasons, some tradekit methods are copied and fully overridden in custom classes until the changes are tested and accepted. Based on past experience, the size of these modules will decrease by 50% after such changes are re-integrated into tradekit.

CONCLUSIONS

This project has convinced Anderson Financial Systems that object-oriented programming represents the optimal solution for custom programming of mission critical applications. It provides fast prototyping, rapid deployment, and stable finished products that are easy to use, learn, and maintain. All new development work at AFS, even on platforms other than NEXTSTEP, is being performed with object-oriented techniques. (A Windows/C++ port is underway.)

The programming truism "Be prepared to throw one away, because you will" proved accurate, but the rebuilding of version two was far less painful and far more iterative than conventional programming. For the most part, new classes were written to correct deficiencies—for example, an entire set of self-verifying text fields was created for different data types—and slipstreamed into the application. A better method for linking the database fields to the UI was discovered, and it "just worked" across all deployed systems. Duplicate (copied) program code in multiple classes has been pushed relentlessly up the inheritance tree, and such changes require only minimal testing. These are tangible benefits that were not possible with conventional programming.

Anderson Financial Systems would advise any organization chartered with the development of custom applications to take a hard look at object-oriented programming for future projects. Especially with an experienced object-oriented programmer to serve as a mentor and guide the design, the learning curve is not difficult to surmount with an open-minded staff—in AFS's case, less than six months. The benefits are dramatic, the pitfalls are minor, and the definition of what is possible will never be the same.

APPENDIX A

A TECHNICAL OVERVIEW OF THE afskit FOUNDATION CLASSES FOR NeXTSTEP APPLICATION DEVELOPMENT

afskit™ is a collection of foundation classes for forms-based NeXTSTEP application development. The objects in afskit provide an application framework with standard forms, find panels, and scan panels for managing the entities in an application. These objects can create and respond to real time events and keep users at all active workstations informed of changes and activities which affect them. The afskit objects also validate user security, to ensure that the application prevents unauthorized use of critical functions and presents itself in an appropriate manner to each class of users.

APPLICATION FRAMEWORK

The following classes provide the application framework for forms-based applications. (The parent class appears in parentheses after the subclass name.)

AfsApplication (Application) - Provides numerous default behaviors for program startup and termination (database connection, printer/color panels, menu contents and alignment, window defaults, etc.); user security (validate user login, validate user security profile and remove menu items as needed, automatic logout times); user preferences management; real-time event management; and real-time menu option management.

AfsEventManager (Object) - Acquires and dispatches real time events to all objects that have registered for such services. Also transmits events to other machines when the local workstation is the source of the activity.

AfsWindow (Window) - Provides a complete set of screen-to-database binding methods; user preferences management; user security management; window/browser resizing and movement; multiple-editing controls; and scrollable text field management.

AfsPanel (Panel) - Identical to AfsWindow, as a subclass of the NeXT Panel class.

AfsForm (AfsWindow) - Extended behaviors for AfsWindows which manage entities from persistent databases. Implements a complete protocol for instance (record) management: new, delete, edit, save, revert, set/fetch active, and find/scan. Provides "find:" and "validate:" services to other AfsForms which use the entity it manages. With its inherited AfsWindow methods, the AfsForm class is the heart of the system.

AfsFindPanel (AfsPanel) - Provides standard behaviors for Find Panels. Always subclassed for specific entities, to provide the required query criteria. AfsForms (and the Main Menu) have a Find button which attaches to these. Find Panels always dispatch selection criteria to a related Scan Panel, which fetches the desired record(s) and displays a selection browser if necessary.

AfsScanPanel (AfsPanel) - Provides standard behaviors for Scan Panels, which fetch specific entities and appear when needed to display and request a selection from among multiple matching records. The standard "scan:" method takes the query criteria from the FindPanel and performs a database scan. If there is no match, an AlertPanel is posted. If there is one match, it is returned immediately. If there are multiple matches, they are displayed in a browser and the user is prompted to select an item from the list.

AfsSwapPanel (Panel) - Provides a facility for NeXT-style inspectors, where the underlying Form is constant and a PopUpList triggers a series of subviews for an **AfsSwapView** on the Form.

AfsReportPanel (AfsPanel) - Provides standard report option management and a complete set of .rtf wrapping functions, standard header and footer methods, and dispatch to external text processing programs (WordPerfect, WriteNow, etc.). For easy extensibility and user security validation, reports are selected through a cascade of main menu options which trigger this panel. Individual reports (subclasses of **AfsReport**) may provide an accessory view to acquire specific data which is needed to create the report, such as date ranges.

SecurityManager (HashTable) - Security is an inherent part of many afskit objects, so security management is provided as a direct function of the application. As a result, there is no need for a separate program to manage the user and functional security profiles. This approach provides two key benefits: reduced maintenance (because there is one less program and no distinct table to keep up to date), and automatic inclusion of new functions in the security matrix when the application is enhanced. As new objects are added to the application, all users inherit their default security profiles until overridden by the security administrator.

The SecurityManager object provides two panels: an authorized user list (driven by Unix login), and a protected object list. The protected object list works as follows: AfsWindows and their descendents, AfsButtons, and AfsBoxes may indicate a desire for security and a default read/write profile. (These attributes can be set in InterfaceBuilder or under program control.) For example, a SalesOrderForm may desire security, and until overridden want to be in a read-only state. Within the SalesOrderForm, the New button may desire security, and until overridden want to be invisible (no read, no write).

When the security administrator runs the application, it enters a special mode where all objects are queried about their desire for security and default security profile. Responsive objects are added to the protected object list, after which specific users can be assigned a custom profile which overrides some or all of the default states. When end users run the application, it determines their effective privileges and removes any options to which they are not entitled.

INTERFACE OBJECTS

The application framework objects depend upon a set of smart user interface objects to get much of their work done. All of these objects descend directly from the standard NeXT appkit objects (in parentheses after the afskit class name) and can be pulled directly from a supplied palette in InterfaceBuilder. At runtime, they call attention to themselves when they become the active responder by drawing a ring around their bounding rectangle.

AfsTextField (TextField) - The standard NeXT text entry field, subclassed to provide better editing support. When AfsTextFields are in a non-editable state, they set themselves to a muted white halfway between pure white and light gray, and they pass through selectText: messages to their nextText or previousText instead of bringing the editing loop to a halt. They also have an attached label, similar to NeXT Form objects. This label is actually a ButtonCell which can send messages to other objects for validation assistance. AfsTextFields can have an owner other than the window on which they reside. The owner acts as a delegate, getting a chance to service help:, validate:, readStorage:, and writeStorage: messages. Even without owner-specific behaviors, AfsTextFields know how to read and write their values directly to database buffers.

The following subclasses of AfsTextField are provided for specific types of data entry fields:

> **AfsStringField** - Performs initial-caps or all-caps transformations; limits data entry to a specified number of characters for fields which are attached to databases with fixed-length fields.
>
> **AfsLookupField** - An alternative to pop-up lists. AfsLookupFields have a table of acceptable values which is maintained through the **AfsLookupTableEditor**. (The Editor can be included directly in the application, or provided through a separate application. The table values are stored in a set of flat files which can be accessed by non-NeXT applications.) When the user enters data in an AfsLookupField, the entry is validated against the table, and incorrect or ambiguous values are flagged. Alternatively, the user can click on the field's label, which brings up an **AfsLookupPanel** with all acceptable values (equivalent to a pop-up list in a ScrollView). From there, a choice can be made and pasted back to the field.
>
> **AfsNumberField** - Performs minimum/maximum value checking. If the user clicks the field's label, a calculator pops up (**AfsMouseCalcPanel**) which can perform intermediate calculations and paste the result back to the field. AfsNumberFields can be told what type of database field they are attached to (char, short, long, float, double) and translate this data automatically. The inspector also provides access to the floating point display format.
>
> **AfsDateField** - Performs date format checking and displays in a variety of formats.

AfsTimeField - Performs time format checking and displays in a variety of formats.

AfsPriceField - A subclass of AfsNumberField that knows how to interpret and display prices in a variety of financial trading formats.

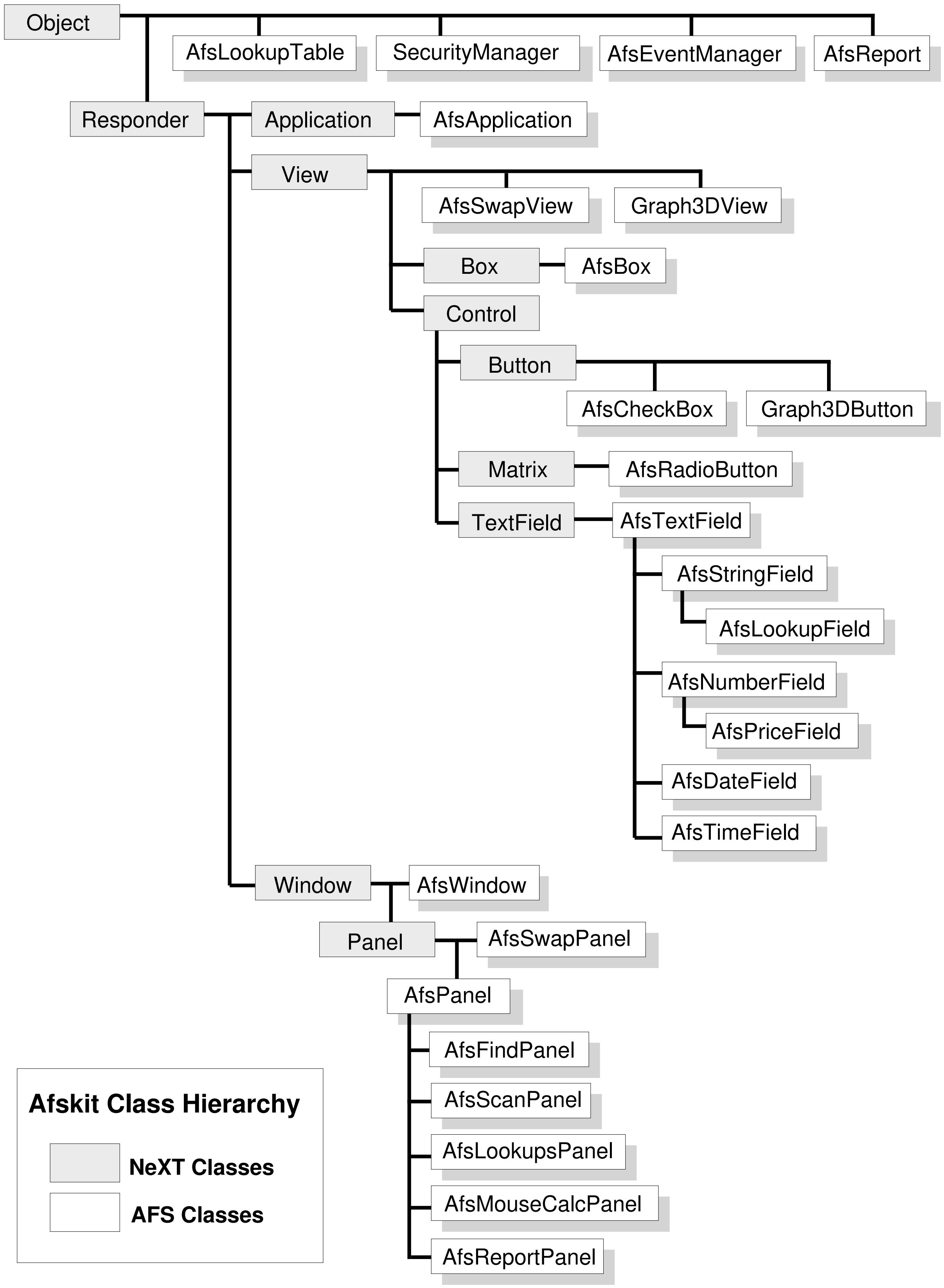
AfsButton (Button) - The standard NeXT push-button object, extended to provide security for the target/action method produced by the Button. See the description of security mechanisms (above) for more details.

AfsCheckbox (Button) - The standard NeXT checkbox object, extended to support keyboard entry and allow inclusion in the edit loop (the sequence when the user presses the Tab key). Addresses complaints about the standard NeXT checkbox being mouse-only, which requires users to take their hands off the keyboard momentarily.

AfsRadioButton (Matrix) - The standard NeXT radio buttons, extended to support keyboard entry and allow inclusion in the edit loop.

AfsBox (Box) - The standard NeXT Box, extended to provide security for the objects it contains. See the description of security mechanisms (above) for more details.

Graph3DButton/Graph3DView - A rubber-sheet 3D graph model that displays a time series of data vectors and allows the user to reorient the viewing angle interactively. Controls for Cartesian and polar viewpoint/objectpoint coordinates, y-axis and z-axis scaling, color vector for point-to-point differences (deltas).


Object
AfsLookupTable
SecurityManager
AfsEventManager
AfsReport
Responder
Application
AfsApplication
View
AfsSwapView
Graph3DView
Box
AfsBox
Control
Button
AfsCheckBox
Graph3DButton
Matrix
AfsRadioButton
TextField
AfsTextField
AfsStringField
AfsLookupField
AfsNumberField
AfsPriceField
AfsDateField
AfsTimeField
Window
AfsWindow
Panel
AfsSwapPanel
AfsPanel
AfsFindPanel
AfsScanPanel
AfsLookupsPanel
AfsMouseCalcPanel
AfsReportPanel
Afskit Class Hierarchy
NeXT Classes
AFS Classes

APPENDIX B

A TECHNICAL OVERVIEW OF THE tradekit CLASSES FOR NeXTSTEP CUSTOM TRADING APPLICATION DEVELOPMENT

The tradekit is a set of base classes for front office trading operations. All classes descend from the afskit objects, adding functionality as required. To clarify the inheritance tree, each tradekit class is followed by its parent class in parentheses.

AfsTradeApp (AfsApplication) - Central coordinator for the entire application. Contains instance variables for all forms, to facilitate communication between them.

Portfolios - Information concerning the user's internal accounts.

> **PortfolioForm** (AfsForm) - Detailed information about individual accounts; provides all functions to add, delete, edit, and find them.
>
> **PortfolioScanPanel** (AfsScanPanel) - Displays a list of all portfolios which match a user-specified full or partial portfolio number; reports user's selection back to the calling object.

Salesmen - Information concerning the user's sales and trading personnel.

> **SalesmanForm** (AfsForm) - Detailed information about individual salesmen and their team members; provides all functions to add, delete, edit, and find them.
>
> **SalesmanScanPanel** (AfsScanPanel) - Displays a list of all salesmen which match a user-specified sales number; reports user's selection back to the calling object.

Client Accounts - Information concerning customers and contra-parties.

> **AccountForm** (AfsForm) - Detailed information about individual customers and contra-parties; provides all functions to add, delete, edit, and find them.
>
> **AccountFindPanel** (AfsFindPanel) - Assists in locating a specific account by allowing the user to enter a full or partial account number or short name. Forwards these criteria to the AccountScanPanel, which locates any matching record(s) and displays a list, if necessary.

AccountScanPanel (AfsScanPanel) - Displays a list of all accounts which meet user-specified criteria; reports user's selection back to the calling object.

AccountPositionsPanel (AfsPanel) - Displays a list of active positions owned by the customer; when visible, synchronizes with the active customer on the AccountForm.

Delivery/Payment/Multiple Confirm Instructions - Information concerning both internal and customer delivery/payment/confirm instructions.

DeliveryForm (AfsForm) - Detailed information about individual delivery instructions; provides all functions to add, delete, edit, and find them. Instructions can be global to the system, so the same set of instructions can be applied to more than one account.

PaymentForm (AfsForm) - Detailed information about individual payment instructions; same functions as delivery instructions.

ConfirmsForm (AfsForm) - Detailed information about multiple confirm instructions; same functions as delivery instructions.

InstructionsFindPanels (AfsFindPanel) - Assist in locating a specific set of instructions by allowing the user to enter a full or partial code number, and select standard and/or non-standard instructions. A slightly different version is provided for each of the instructions windows. Forwards criteria to the associated InstructionsScanPanel, which locates any matching record(s) and displays a list, if necessary.

InstructionsScanPanels (AfsScanPanel) - Display a list of all instructions which meet user-specified criteria; reports user's selection back to the calling object.

Security Descriptions - Information concerning specific securities.

CusipForm (AfsForm) - Detailed information about individual securities; provides all functions to add, delete, edit, and find them. To minimize clutter while supporting numerous security types, separate views (accessible through a pop-up button) are provided for different types of securities.

CusipFindPanel (AfsFindPanel) - Assists in locating a specific security by allowing the user to enter a full or partial CUSIP number, ticket symbol, state code, name, coupon range, and/or maturity date range. Forwards these criteria to the CusipScanPanel or the Kenny Information Systems server, which locates any matching record(s) and displays a list, if necessary.

CusipScanPanel (AfsScanPanel) - Displays a list of all securities which meet user-specified criteria; reports user's selection back to the calling object.

KennySpeaker (Speaker) - Provides access to the Kenny Information Systems server, a separate process (written by AFS) which runs at one published location on the network and coordinates all dialups to the KIS database. Places telephone call, interprets returned information, and signals caller upon completion.

CalculatorForm (AfsForm) - A free-form window on which calculations can be performed. To minimize clutter while supporting numerous security types, separate views (selected automatically according to available features) are displayed for different types of securities. To minimize data entry, has direct access to the CusipFindPanel.

Issuer Descriptions - Information concerning issuers for whom securities are underwritten. Intended primarily for money market systems, but also has applications in long-term fixed income and equity underwriting.

IssuerForm (AfsForm) - Detailed information about individual issuers; provides all functions to add, delete, edit, and find them. Includes criteria that validate ticket activity in real time, such as maximum maturities and minimum block sizes.

IssuerFindPanel (AfsFindPanel) - Assists in locating a specific issuer by allowing the user to enter a full or partial issuer number, symbol, or name. Forwards these criteria to the IssuerScanPanel, which locates any matching record(s) and displays a list, if necessary.

IssuerScanPanel (AfsScanPanel) - Displays a list of all issuers which meet user-specified criteria; reports user's selection back to the calling object.

IssuerPositionsPanel (AfsPanel) - Displays a list of active positions outstanding for the issuer, by maturity date; when visible, synchronizes with the active issuer on the IssuerForm.

Position Maintenance - Information concerning the user's active positions.

OfferingForm (AfsForm) - Detailed information about positions owned for a specific portfolio and issuer combination. Provides repricing functions for all positions (primary and secondary); allows hand entry of proposed primary offerings. Automated end-of-day closeout process (buy/sell order matching) for primary offerings. Direct access to TicketScanPanel to locate outstanding trades against selected positions.

OfferingFindPanel (AfsFindPanel) - Assists in locating a specific set of positions by allowing the user to enter a full or partial trading account and issuer short name. Forwards these criteria to the OfferingScanPanel, which locates matching record(s) and displays a list.

OfferingScanPanel (AfsScanPanel) - Displays a list of all offerings which meet user-specified criteria; reports user's selection back to the calling object.

PositionForm (AfsForm) - Provides an editable view of any one position, unlike the OfferingForm, which operates on sets of related primary underwriting positions.

PositionScanPanel (AfsScanPanel) - Allows the user to specify a set of filters to monitor any subset of active positions in real time. Filters include security type, position type (primary/secondary), portfolio, approved issuers, maturity range, coupon range, offered rate range, and quality ratings range. Accepts customer preference templates from the AccountForm to minimize repeated data entry.

PositionScanPanels display a list of all positions that initially match the selection criteria, then continue to monitor purchases, sales, and repricing. The font switches to *italics* while a position is subject to change. PositionScanPanels may be miniaturized; the mini-icon "cracks" if a change comes in while the window is miniaturized. A separate audit trail log is maintained within the window to review events which occurred while the window was shrunk. Traders can trigger buy and sell tickets directly from PositionScanPanels. The total number of active PositionScanPanels is unlimited, so salesmen can keep separate windows for each outstanding customer inquiry throughout the trading day.

CollateralScanPanel (PositionScanPanel) - An expanded version of the PositionScanPanel that monitors repo/resale collateral usage and recommends the order in which positions should be used, based on trader-specified preferences.

Tickets - Information concerning transactions.

TicketForm (AfsForm) - Detailed information about individual tickets; provides all functions to add, delete, edit, and find them. Usually subclassed for different types of securities and to provide user-specific behavior and appearance.

RepoTicketForm (TicketForm) - An expanded TicketForm with a collateral browser which knows how to manage repo/resale/borrow/pledge transactions. Also provides complete maintenance of these deals, such as substitutions, rate changes on open repo, repricing, and interest cleanup.

TicketFindPanel (AfsFindPanel) - Assists in locating a specific ticket by allowing the user to enter a ticket number, trading account, customer, salesman, date range, and status (confirmed/unconfirmed). Forwards these criteria to the AccountScanPanel, which locates any matching record(s) and displays a list, if necessary.

TicketScanPanel (AfsScanPanel) - Displays a list of all tickets which meet user-specified criteria; reports user's selection back to the calling object. For salesmen, there is special instance of this panel which displays a real time list of all of today's confirmed and/or unconfirmed tickets.

ProposedSellPanel (AfsPanel) - For salesmen, allows primary trades to be proposed to the responsible trader without voice contact. If the trader accepts the order, a ticket is drawn up automatically and returned to the salesman's workstation. Otherwise a rejection notice is displayed, and the salesman can follow up personally.

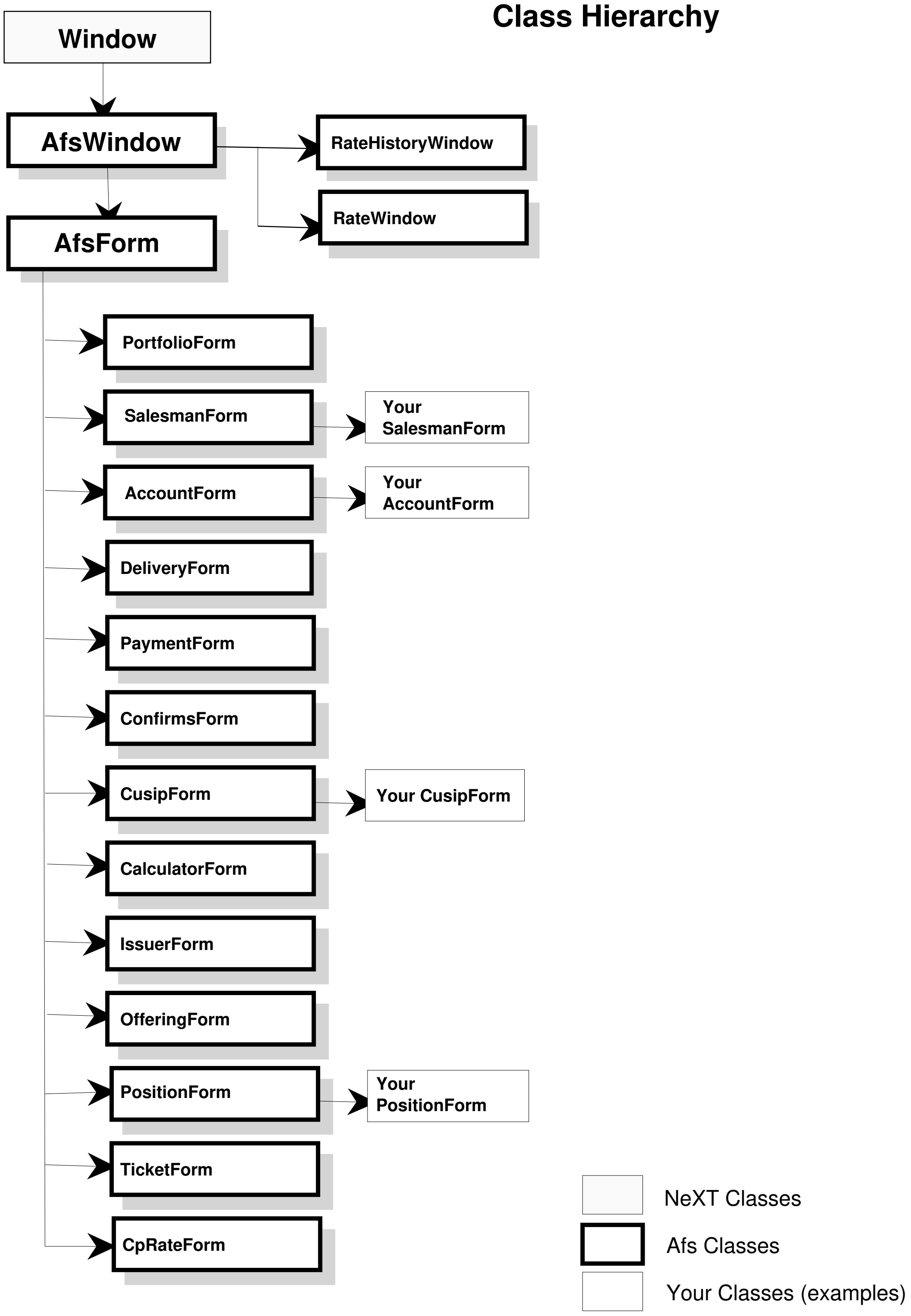
Rate Runs - Information concerning the user's internally generated interest rate runs, current and historical.

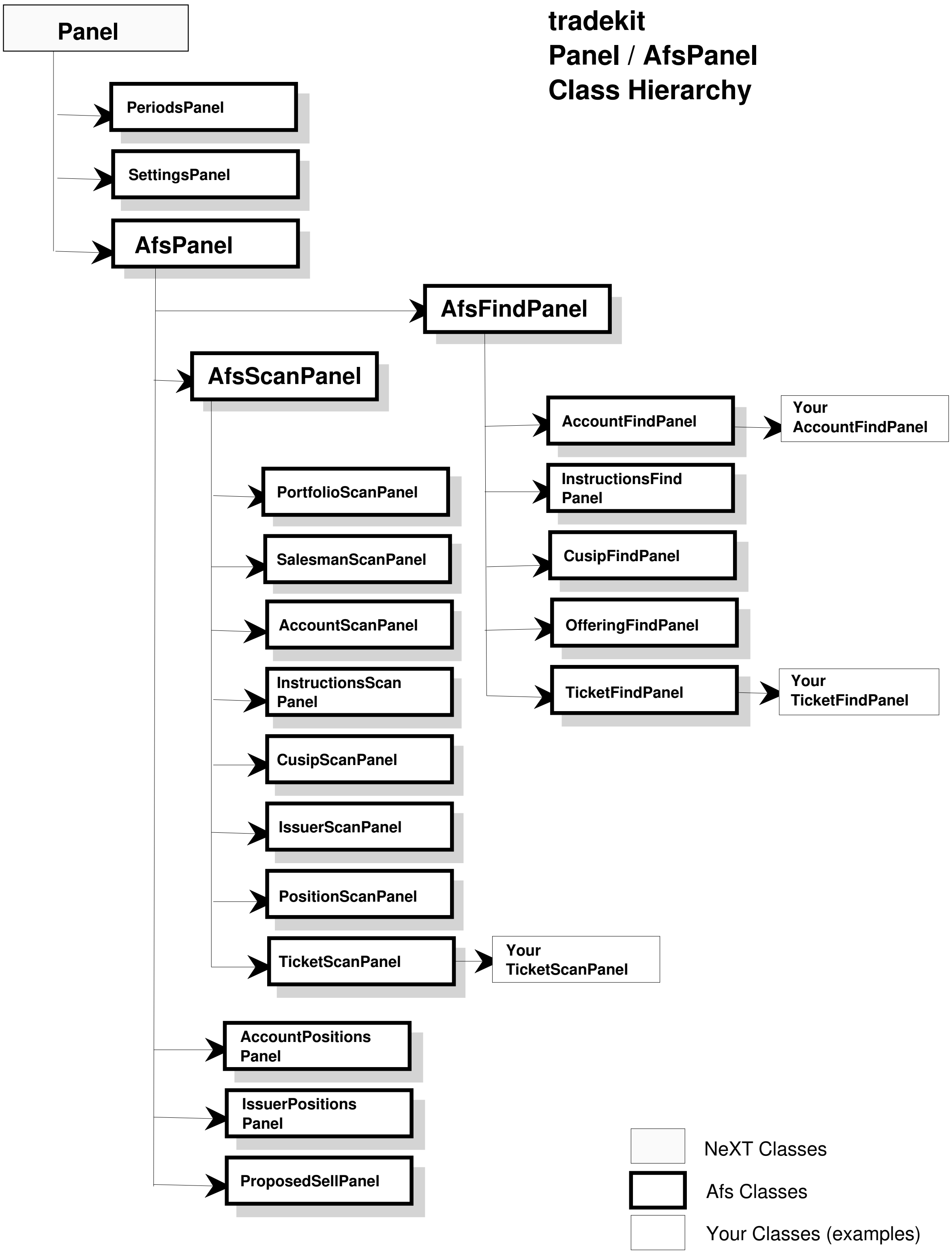
RateWindow (AfsWindow) - For individual security types, a window for traders to enter the firm's internally generated interest rate scales, and for salesman to monitor these tables in real time. Multiple rate windows can be created, one for each security type. Each RateWindow has a **PeriodsPanel** (Panel) which allows the user to determine which maturities will be priced for a specific RateWindow.

CpRateForm (AfsForm) - A special RateWindow tied to a specific issuer and date. Allows issuer-specific historical offering scales to be retained.

RateHistoryWindow (AfsWindow) - A composite monitor of multiple instrument classes, current and historical; provides a real time side-by-side "snapshot" of multiple sectors. The RateHistoryWindow has an attached **SettingsPanel** (Panel) which determines its organization and behavior. The SettingsPanel toggles between current and historical rates, and allows entire curves or just specific maturity terms to be compared over time.

tradekit
AfsWindow / AfsForm
Class Hierarchy




tradekit
Panel / AfsPanel
Class Hierarchy
Panel
PeriodsPanel
SettingsPanel
AfsPanel
AfsFindPanel
AfsScanPanel
AccountFindPanel
Your AccountFindPanel
InstructionsFind Panel
CusipFindPanel
OfferingFindPanel
TicketFindPanel
Your TicketFindPanel
PortfolioScanPanel
SalesmanScanPanel
AccountScanPanel
InstructionsScan Panel
CusipScanPanel
IssuerScanPanel
PositionScanPanel
TicketScanPanel
Your TicketScanPanel
AccountPositions Panel
IssuerPositions Panel
ProposedSellPanel
NeXT Classes
Afs Classes
Your Classes (examples)