The Bank of Montreal and SAS Institute (Canada) Inc. have collaborated on the development of an IS system which supports the sales and executive arm of the Corporate and Institutional Financial Services Group. Key considerations of the project, from both the business user and development/support perspectives, included:

- Intuitive Graphical Interface
- Cost Effective
- Faster Development Cycles
- Modular Design/Extensibility
- Solid Data Warehouse Foundation
- Platform Independence/Portability
- Better Use of Existing Programming Resources
- Data Driven/Reduced Maintenance
- Solid Performance
- Stability

Discussion covers the conversion of an existing critical application, the expansion of functionality, and optimal use of emerging client/server architectures.

Introduction

Bank of Montreal's Relationship Profitability and Measurement (RPM) System was conceived in 1991 by Relationship Management Executives, who required an information system which tied together all the products and services offered to our client base. A primary objective was to identify the Lines of Business and profitability of our relationships and provide a vehicle through which Relationship Management could best serve these relationships.

RPM strengthens communication among Bank of Montreal's Product Managers and Relationship Managers across North America, as well as the Bank's subsidiaries, Harris Bank Corp. in Chicago and Nesbitt Thompson in Toronto.

The main business goal of RPM 2.0 was to deliver a more intuitive system to the emerging client/server platform strategy undertaken by the development group.

System Conversion

Version 1 of RPM was developed as a DOS based application developed in Magic-PC and stored in Btrieve files. The daily and monthly production cycles required a physical delivery of all data files from the mainframe to the local file server, followed by physical distribution to eight local area networks across North America.

The business development team required a software development tool which could easily meet the challenges of the client/server architecture being implemented by the Bank, while at the same time provide immediate additional value to the Business users of the RPM application.

The Bank could have simply undertaken a systems conversion exercise to create a 'Windows' look and feel, while addressing the Business Users' requirements. But this clearly would not have taken advantage of the improved design and development opportunities which presented themselves in the forms of Object Orientation, Data Warehousing, Prototyping, and Portability.

Object Oriented Design

Much has been made of Object Oriented (OO) Analysis, Design, and Programming. Many debates have occurred among computer professionals as to what comprises true OO methodologies, varying from as few as three to as many as seven principles (Object Management Group). Some of the more important tenets include:

- Abstraction
- Encapsulation (modularity)
- Inheritance (enact global changes without massive redevelopment)
- Reuse (environmentally friendly programming)

All of this promises to yield amazing benefits such as:

- Reduced maintenance
- Adaptability and extensibility through an object library and object classes
- Rapid application development by assembling pre-built parts

Although we did not set out to follow strict OO methodologies in developing the new RPM application, the
limitations of previous design approaches did impact the
decisions taken in both the design and development.

The application was divided into major components or
modules (objects) which would be connected only in their
similar traits and their ability to function within the RPM
framework (class structure).

There are five sub-classes of RPM screens:

- Push-button menus;
- Profile Request;
- Financial data display, typically through scrolling
  extended tables;
- Message or information screens requiring little or no
  user input (e.g. click OK push button); and
- Dialogue screens displaying selection options via pop-up
  menus, list boxes, radio buttons, and check boxes.

Each of these screen components would be responsible for
ensuring and obtaining required parameters, data extraction
from the server, preparation of that data for display, data
validation, display of information on-screen, and messaging
(to other components of the application and to the user).

Messages or data elements were communicated among the
objects using a combination of global macro variables (useful
for single operations and profile key values) and SCL lists (local
Environment List). Parameters between frames were limited
to some calls to the Profile Request screen. This was
required because of the decision to use a CALL GOTO over
a CALL DISPLAY method for invoking a screen.

The success or failure of this approach will, of course, be
realised and measured over time.

Intuitive Graphical Interface

Today's desktop applications and software tools are moving
increasingly toward the Multiple Document Interface (MDI).
One of the chief features of this design is the "flattening" of
the user interface so that much of the work is done in one
central screen, with dialogue boxes, tool bars, and other
operations being performed on the "objects" or "documents"
within that main screen. Think of your favourite MS-
Windows word-processor or spreadsheet. This is seen as
being more intuitive for users.

The flattened structure also helps to avoid that question that
users often pose in a hierarchical navigational structure, "Go
back where?" While programmers are either trained to or
naturally think in terms of upside-down trees, where else,
other than the corporate structure, do we see this manner of
operating?

In RPM, we flattened the hierarchy by constructing a GoTo
pull-down menu system. When users want to move from
Task A to Task C, they simply click GoTo > Task C from
the pull-down menu of Task A. There is no need to Return
to the Main Menu or back their way up some branch!

This required building a dynamic application stack (for those
few who wanted hierarchical navigation) with its own SCL
Methods to manipulate and manage it. It also required
reducing or eliminating the need for parameter passing. This
actually forced better encapsulation of the modules, in
support of an OO design.

The benefits were immediate. Users define their own
application flow at run-time. The application reacts to and
reflects the workflow, rather than dictating it. Users also
appreciate the minimisation of mouse clicks or menu
selectors to get where they're going.

Technically, another benefit was that we could limit the
number of active and inactive FRAME entries that were
consuming resources, particularly memory, at run-time. A
CALL GOTO would invoke the next FRAME, clearing the
SAS/AF® application stack and closing the preceding entry.
Entries remain open, but inactive, if a CALL DISPLAY
routine is used. With the added complexity of these
encapsulated modules, each consumed a fairly high level of
resources, thus if the GOTO feature had not been used, the
performance and stability would have been severely degraded.

Data Warehouse

A key objective in creating the data warehouse for RPM was
to allow other applications and departments to share the
common data tables. SYBASE, a relational database, on a
UNIX platform was selected as the standard, providing user
access via several front-end and data retrieval tools.

One of the advantages of using a relational database instead
of Btrieve was to enable the introduction of new fields
without impacting existing extraction programs. Btrieve
identified fields by relative position, rather than name, thus
continued stability relied heavily on minimal user requirement
changes.

Another benefit was the introduction of a Structured Query
Language (SQL) to facilitate flexible data retrieval. Changes
in extraction logic, as well as security processing, could be
made at the database level, without requiring extensive or
immediate front-end code changes.
There are three main types of data in the RPM database:

- Customer and organizational tables;
- Daily and monthly historical financial data; and
- Application control data.

Of special interest are the control tables. These are specific to the RPM V2 application and were created to be maintained by the Finance arm of the Bank, without need of programmer staff to enact changes. These maps provide dynamic data-driven support for various aspects such as rate changes, accounting general ledger terminology, formula calculation, and screen layouts.

Portability

Each line of code was evaluated to ensure that it was written to be portable. Other than the dependence on certain SAS® software products on the client platform, the client workstations can be a mixture of MS-Windows (16-bit or Win-32), OS/2, or UNIX, without any code changes or even recompiling. This allows the RPM application to take advantage of the portability of the SAS System. In fact, the only real dependence is on the availability of the SAS/ACCESS® Interface to SYBASE and SQL Server on the client platform.

For example, a typical SCL code fragment to get around the specification of the SYBASE SQL Server engine depending on the platform might be:

```sas
IF SYMGETC(SYSSCP) IN (VAN,VAN-32,OS2) THEN
  SQLSERVER = 'sqlservr';
ELSE
  SQLSERVER = 'sybase';
```

or using the SAS Macro Language in the database load and extraction programs:

```sas
%IF &SYSSCP EQ VAN OR &SYSSCP EQ VAN-32 OR &SYSSCP EQ OS2 %THEN
  %LET &SQLSERVER = sqlservr;
%ELSE
  %LET &SQLSERVER = sybase;
```

Using either the SCL variable or the macro variable reference, after it is determined, allows the code to be evaluated at run-time, without having to recompile and maintain separate versions for different operating systems. Furthermore, by placing these code blocks into SCL Methods or loadable Macro Routines, additional operating systems and other modifications can be made as they evolve. In fact, the WIN-32 specification was added when the documentation of Release 6.10 was made available, in anticipation of the upgrade.

Coding for Stability

Another important aspect to the application development was the vigorous enforcement of robust coding. It is important to try to inform the user (and developer) of an unexpected event, then exit elegantly. With application modules, database Stored Procedures, database access rights, communications, workstation software, and server software all being maintained and upgraded by different IS professionals in different departments and, even, different cities, one change could cause something unexpected to occur. Under no circumstance should the user get bad data! While still frustrating, at least a message informing the user that "the database server could not be found" or that a "variable could not be found" with a telephone number for the Support Desk, would help to diagnose the problem so that a cure can be investigated and made as quickly as possible.

No assumptions in the code can be made. All return codes from SCL functions, SQL queries, and DATA and PROC steps must be checked. All required parameters must be checked before a function is used, a routine is called, or a block of code submitted. This may lead to what seems like ridiculously redundant processing overhead, but the benefit is well worth the additional split-second it takes to perform the several additional SCL statements!

In a complex application module consisting of 1500 or more lines of SCL code, plus related SCL Methods, a different programmer may modify a section of code that will adversely and inadvertently impact another section. If a DBA alters the structure of a database table or the results produced from a Stored Procedure, problems may arise. Of course, to mitigate this, programmers were required to thoroughly and accurately comment their code as they wrote it! Additionally, complete written system documentation of all the related parts was produced for all levels of IS managers, analysts, and developers, who might want to reuse or modify a portion of the RPM environment.

Finally, an internal messaging system was built to provide detailed and friendly custom error messages to the user. The custom text can consist of several lines (up to 20 in our case), which are read from a global SAS data file. This can be easily customised to support multiple languages at once. The error codes are themselves understandable names, e.g. DATA-SET-NOT-FOUND, making them easy for the programmers.
to use. Finally, the programmer can call the error message routine with optional run-time parameters, such as the data set and module names, which help to diagnose the specific occurrence of the problem.

Hopefully, most of the error messages will never be seen by actual Business users!

**Optimisation**

It was important to develop under actual circumstances to better grasp the true performance that a user would face when running the system. Too often, systems are developed by modelling or prototyping on a stand-alone system then scaling up and attempting to optimise later. Prototyping is valuable, but like a good author will often throw away the abstract and start over, a prototype may have to be radically rewritten to take advantage of the true environment.

Systems developers usually have access to the best hardware. Contractors often prefer to work off-site rather than having to work at the client-server location. The server components may not be or are lagging behind the front-end development, and sample data sets are, oh, so much easier to work with. All of these are valid reasons for wanting to develop in an atypical environment, but the application performance suffers, resulting in user dissatisfaction.

By leaving the communications issue to the last, the fastest or optimal method may not be possible to simply retrofit. Nobody wants to redesign at that point. With the SAS System, a number of options were available to provide the data extraction from the server SYBASE database to the MS-Windows SAS/AF client front-ends:

- SAS/CONNECT® software on both platforms with a PROC DOWNLOAD step in a submit block;
- SAS/CONNECT and SAS/SHARE® software taking advantage of the Remote Library Services capabilities;
- SAS/ACCESS® to ODBC with a SAS/ACCESS to SYBASE and SQL Server extraction on the Server; or
- SAS/ACCESS to SYBASE and SQL Server on the client workstation via SQL pass-through.

The SAS/CONNECT software options are probably the easiest to retrofit from a stand-alone prototype, after all, we're still dealing with simple SAS data files on both the server and the client. If we had required complex report processing or analysis, that would have been slower to run on the MS-Windows PC. SAS/CONNECT software with replay of the output on the attached client would have been the way to go. As it turned out, the option that delivered the greatest performance for the simple data extraction (typical query result is fewer than 100 rows) was to use the SQL pass-through and let SYBASE download the data. This also allowed us to use the pre-compiled SYBASE Stored Procedures that contained the pre-optimized database extraction plans (joins, index usage, and subsets). The average response time from the centralized server in Toronto to any city was about two seconds!

As the front-end development progressed, it was necessary to do the parallel database optimisation to ensure that the application specific Stored Procedures delivered the best overall time. For example, should the data manipulation take place on the server or be done on the client using SCL or SUBMIT blocks? The parallel development allowed us to evaluate this on a module by module basis. The Transact SQL of SYBASE allowed us to explore parameter passing, complex logic, and multi-step processing possible in Stored Procedures, as well as the optimisation provided by various clustered index arrangements.

In fact, all development of the client/server environment was done in parallel:

- Hardware, operating system, and software of the client workstation;
- Hardware, operating system, and software of the server;
- Server database, including load programs;
- LAN and WAN communications; and
- Client front-end application.

The total system performance is the sum of the performances of all parts. As each of these parts is managed, maintained, and upgraded by different areas, coordination among the parties is integral to a satisfactory conclusion. Nobody said this stuff was easy!

The optimisation was best done as a gradual evolution of all parts throughout the development cycle. Indeed, even as the system has been rolled out, new products and new technologies have been substituted for various parts, yielding better overall results.

**Project Management**

We learned ways to have a successful development project, although we did not necessarily succeed at all aspects ourselves.

The project enlisted the support and involvement of several diverse groups. The Operations Division was charged with building a WAN and LAN environment that supported much more than RPM. Mainframe programmers were required to keep pace with rapidly changing business practices in an intense global economy. User feedback throughout the application development process was solicited to ensure that we were building something that would meet their expectations. And last, but not least, the application
developers themselves had to remain interested throughout an iterative development path. Developers could not get frustrated that what they built yesterday might be completely replaced tomorrow!

Highly skilled and creative programmers were essential to tackle bleeding edge technology. A fair or even average programmer could hamper a project. If you don't have access to these genius programmers, the tried and true tactic of developing thorough program specifications before coding begins might be necessary. In our case, we wanted the advantage of an evolving specification, so this was not an option. Of course, to retain the skilled programmers, it was necessary to build a strong sense of teamwork and ensure that they could see measurable progress, even with an evolving specification.

A ruthless heartless editor was necessary to audit the code produced by every programmer. This provided a second opinion on ways to provide easy to maintain, efficient, robust, and well-documented code. Editors can be code-knowledgable project leaders or other programmers on the team.

A thorough understanding of related source or legacy systems and the strengths and weaknesses (user and technical viewpoints) of the existing system to be replaced was necessary. Additionally, an understanding of the business practices was essential. This may require a liaison between a technically knowledgeable business analyst and a technical systems analyst, who sees beyond the bits and bytes, throughout the development process. There is no way for a technical consultant to instantly pick up 20 years of business knowledge without ongoing discussions with people who have that experience.

Documentation is essential and is best done as soon as the information is freshest—during development:

- Comment blocks embedded in the code (about 25-33% of the total line count is a good rule of thumb);
- An IS manager level overview;
- A user overview of the system and built-in context-sensitive help (preferably written and maintained by a user rather than a programmer);
- Software system descriptions for the technical analyst (heavy on logical flow, light on physical);
- Hardware and communications documentation prepared by and for the Hardware Systems Support (heavy on physical, light on logical);
- Full database documentation (Entity Relationship Diagram and descriptions of the tables, keys, indices, and Stored Procedures);
- A description of the front-end application suitable for an applications developer charged with maintaining or modifying a component;
- All source code; and
- Production processing documentation for support.

To be honest, while we wanted to have this produced as part of the development effort, the pressure to release working code became too great. Of course, as soon as the code was released, people started asking, "So, where's the documentation, already?"

Conclusion

Client/server development is a major undertaking that requires not only expert resourcing, but a strong financial commitment to see it through to delivery. Business users, while very interested in creating an intuitive Windows-based tool, may not consider some of the design and development issues, particularly in the back-end that is hidden from them. Approximately 75% of the total development cost was allocated to this unseen effort.

SAS, SAS/AF, SAS/ACCESS, SAS/CONNECT, and SAS/SHARE are registered trademarks of SAS Institute Inc., Cary, NC, USA.

SYBASE is a registered trademark of Sybase, Inc. SYBASE SQL Server is a trademark of Sybase, Inc.

OS/2 is a registered trademark of International Business Machines Corporation.

Other company and product names may be trademarks or registered trademarks of the respective company with which they are associated.