ABSTRACT
When developing large software systems, it is often difficult to foresee exactly which trade-offs are important, and which quality parameters will be of importance down the road. This paper reports experiences from a project in which a large application framework for B2B integration has been continuously developed and used over a five-year period. The framework has been the foundation for a variety of different concrete applications. Here we will report on our experiences from this endeavor.

Categories and Subject Descriptors
D.2.13 [Reusable Software]: Reuse Models, Reusable libraries
D.2.11 [Software Architectures]: Domain-specific architectures

General Terms
Design.

Keywords
Application Frameworks, Object-Oriented Programming, C++, Architecture, Enterprise Application Integration.

1. INTRODUCTION
On long-term, software development projects, it is not possible to anticipate all future requirements. Therefore, it is necessary to design for change. One approach is to define a generic solution to a set of related problems, and then iteratively extend that solution to solve the next problem. As the problems change, the solution can be refined and adapted.

An application framework provides such a solution by defining a template, which can be reused to build many related applications. As new requirements emerge, new applications can be constructed that build upon the efforts and understanding of prior deliverables.

This paper examines an object-oriented, application framework called EXPRESS. In particular, it explores the experiences of the project team – presented as 13 lessons learned – in the context of software engineering. Software engineering involves analyzing trade-offs and making decisions to optimize value given scarce resources. It also involves measuring the quality parameters that describe the product. One of the challenges of long-term software projects is that these trade-offs and quality parameters are not always clear at project inception.

1.1 Problem Domain
EXPRESS was originally developed within the financial services industry to integrate middle and back office applications at banks. While the architecture and project experiences can be applied to other domains and industries, this paper focuses on banking applications that can be created with EXPRESS.

When investment banks trade securities (e.g., stocks, bonds), messages that contain the trade details must pass through many applications at multiple institutions in the Bank-To-Bank (B2B) supply chain. Examples of institutions include other banks, depositaries, settlement networks, and government agencies. Processing and delivering these messages involves integrating across a wide range of communications protocols, data formats, and application-level processing.

Figure 1 illustrates the flow of messages among applications. The front office is where traders enter trade details. The middle office is where both banks – one for each counter-party to the trade – agree electronically to the details of the trade. The back office is where securities and funds are actually exchanged between banks. In between the trading banks are many applications at intermediary institutions (e.g., settlement networks).

Application integration is complicated by a dynamic industry and requirements that change frequently. For example, prior to the introduction of the Euro currency, many banks in Europe routed messages by currency code; once the European Union moved to a single currency, the currency code attribute was no longer useful for message routing. Other examples include the re-regulation of the banking industry in America [1] and on-going bank mergers. The EXPRESS project grew from the need to insulate middle and back office applications from these changes.

Traditionally, the acronym B2B implies Business–To–Business e-commerce, in which two or more businesses trade in a digital community. The problem domain for this paper, Bank–To–Bank, can be described as systems integration for supply chains. Both interpretations of B2B involve similar technical challenges.
2. APPROACH

EXPRESS is an object-oriented, application framework written in C++[2]. An application framework provides most of the structure for a generic application, which can be reused to build a family of related applications. This framework portion is developed once and contains elements common to all applications, typically in the form of collaborating, abstract base classes.

Implementing derived classes then creates specific applications. In EXPRESS, derived classes are grouped into components called Plug-ins. Plug-ins can also be reused from one application to the next.

One of the benefits of application frameworks is a reduction in the development time required to build each new application due to the fact that much of the code in the common framework is reused. Another benefit is that maintenance effort is reduced since code common to all applications is in one place.

The EXPRESS framework builds on the Broker Pattern[3], in which applications at remote banks are inter-connected with EXPRESS in the middle providing integration services. The services are broken down into the following subsystems:

- **Communications Layer.** Provides adapters to send and receive messages to and from other applications. This layer provides adapters to all of the middleware and session layer protocols used by the various institutions. Messages are exchanged through protocol-specific channels called message queues, which are implemented as Plug-ins.

- **Format Layer.** Interprets and translates incoming message streams from one format to another. Formats can be proprietary messaging standards or de-facto industry standards such as SWIFT[4]. Translators for different formats are implemented as Plug-ins.

- **Application Layer.** Routes messages among a pool of proxies, which represent remote applications at other banks. Proxies provide hooks for application-level (bank-specific) processing. Proxies are implemented as Plug-ins.

- **Foundation Libraries.** Contains classes and functions for message routing, control, scheduling, exception handling, and various shared technical services such as file processing utilities (banks often exchange batch files as well as on-line message traffic).

To create an EXPRESS application, one must write Plug-ins (if they do not already exist) for each of the three layers and, optionally, a derived controller that inherits from the default Controller in the Foundation Libraries. Additionally, one must add reference data in the EXPRESS control database. The generic controller provides for subsystem initialization when EXPRESS starts up, event dispatching during run-time[5], and subsystem termination when EXPRESS shuts down. It is possible to write application-specific controller instead of using the generic controller.

- **Examples of Plug-ins for the Communications Layer include sockets, flat files, in-memory tables, PrintQs, MQ Series, and proprietary bank protocols.**

- **Examples of Plug-ins at the Format Layer include support for fixed-length fields, tag-value formats, and file formats with different header/body/trailer options.**

- **Examples of Plug-ins at the Application Layer include IIOP-enabled proxies that stand in for applications at other banks.**

Our initial thinking at the beginning of the project was that within a layer, only one Plug-in would handle a given message stream. After experimentation, we discovered that it can be useful to pass a stream of messages through several Plug-ins in the same layer. The inspiration for this idea came from the Pipes and Filters architectural pattern[3]. For example, the Format Layer needs to parse incoming messages into an intermediate token list and then apply translation. As another example, the Application Layer might need to implement duplicate message checking. This could be done by inserting a filter Plug-in into the stream to catch and remove duplicate messages (based on some bank-specific information).

Figure 2 shows how EXPRESS can scale to larger deployments.

![Figure 2 EXPRESS Switching Network](image-url)
As the project evolved, the scope of deployments expanded to include different business units and locations. We realized that the output of one EXPRESS application could feed the input of another to form a switching fabric across the enterprise. In this way, EXPRESS supports a wide-range of deployment scenarios from large data centers supporting global clearance and settlement traffic to one-process adapters for translating a message stream to a different format.

**Lesson #1:** When building application frameworks, always maintain a clear separation between application independent elements, which belong in the framework and application dependent elements, which belong in derived classes (in components called Plug-ins).

### 3. DECISIONS

An important aspect of software engineering involves selecting the best solution for a given problem. Project experience comes from living through the consequences of long-term architecture decisions. It is only after the consequences have been realized that a team can go back and assess the original decisions. Over time, these post-mortem reviews add to the collective wisdom of the team. It is important for a team to clearly document important decisions since, years later, it can be difficult to remember exactly how the team got into a particular situation. Several project decisions are presented below.

#### 3.1 Determining the Right Level of Abstraction

Software engineers are notorious for creating unnecessary layers of abstraction in the name of software reuse. Wrappers for Application Programming Interfaces (APIs) are a common example. When we designed the Communications Layer, we considered writing an abstraction layer to hide middleware APIs such as MQ Series. The argument against this approach was that it would add complexity and limit access to MQ Series’ rich features. The argument in favor of this approach was that we needed a consistent abstraction layer across a wide variety of communications protocols including proprietary message distribution systems that used databases. Ultimately, we decided to introduce our own abstraction layer, which is illustrated in figure 3.

The framework portion consists of four abstract base classes: MsgBase specifies methods common to all MsgQs such as connect() and disconnect(); MsgInQ is for receiving messages; MsgOutQ is for sending messages; and MsgQ combines all methods into one class. MsgInQ and MsgOutQ use virtual inheritance to specialize MsgBase. There are more methods than illustrated such as the ability to use non-blocked I/O. Additionally, the MsgBase class uses a finite state machine to enforce proper execution of the MsgQ class hierarchy methods. For example, it is not possible to call send() without first calling open().

The application-specific portion of the class hierarchy contains the derived classes of strongly-typed MsgQs. So far, we have implemented MsgQs using MQ Series, SWIFT, Sockets, files, in-memory queues, and proprietary middleware (which uses a relational database). And, we have experimented with FAXQs. We discovered many uses for MsgQs:

- Point-to-point messaging between banks
- Network access points for settlement and clearance networks
- Routing duplicate message streams to contingency sites
- Load balancing across different EXPRESS nodes
- Message capture and replay for regression testing and bank emulation
- Creating audit trails and message logs

![Figure 3. MsgQ Class Hierarchy](image-url)
MQ Series traffic between both banks. Fortunately, we were able use MsgQs to switch production traffic to an alternate communications platform that still worked. Had we not decided to wrap the MQ Series API, we would not have had this recovery option.

Lesson #2: When designing abstract base classes, distinguish between specifying behavior (i.e., class protocol) vs. factoring out common implementation shared by derived classes. It is generally better to do one or the other, not both.

Lesson #3: Resist the temptation to create unnecessary abstraction layers (e.g., API wrappers) unless absolutely necessary.

Lesson #4: Systems built with redundant components of different types are generally more reliable than systems built with redundant components of the same type. The space shuttle, for example, has different types of computers whose outputs are compared with voting logic. Such an approach increases reliability and availability.

3.2 Compromise in Layer Positioning

One aspect of software engineering is compromise. We experienced this when deciding whether to put the EXPRESS MsgFactory in the Communications Layer or the Format Layer. To appreciate this dilemma, it is first necessary to understand how EXPRESS uses the MsgFactory to create messages.

In EXPRESS, the MsgFactory interprets data on an input buffer and instantiates the correct, strongly typed ExMsg object. Type safety is necessary to ensure proper translation of messages from one message to the next. Note that languages such as Java provide class loaders: C++ has no built-in function to do this.

The two aspects of EXPRESS that make class loading challenging are that (a) no assumptions can be made about the format or protocol of an input stream since its specification is generally defined by another bank; and (b) extensibility is essential: new types have to be introduced without touching existing code. As an example, introducing new code for mortgage message processing could not put bond message processing at risk.

One good approach to this problem is the Exemplar idiom[6]. An Exemplar is an object that represents a class, which can be used for class loading in C++. At system startup, one exemplar for each type of message is instantiated and added to a linked list of static objects. Then, at run-time, the MsgFactory iterates over the exemplar pool passing each one a reference to the input buffer. Exemplars examine the input buffer to determine if they are the same type of message as the data in the buffer. Most exemplars return null and the iteration continues until the pool of exemplars is exhausted or one of them finds a match (i.e., it recognizes its own type in the input stream). If there is a match, the exemplar clones a new message object of the same type, un-flattens itself for EXPRESS is that existing code, including the factory code, does not depend upon newly introduced classes. Figure 4 illustrates an example.

The Exemplar idiom generally worked well, although we experienced some unexpected results during unit and system testing which led to minor design changes. Each exemplar was declared with file scope and defined in its own *.o module. Since they were defined outside of main and linked into the final executable, the order in which exemplars were added to the message Factory’s linked list was unpredictable (at least across compilers). Thus, we had to be careful to ensure that class loading did not depend on the order of exemplars. In other words, one input byte stream could never match two or more message types. This turned out to be non-trivial since EXPRESS had to support a wide variety of message formats defined by different institutions.

```
template <class B> class Exemplar {
   public:
      Exemplar( Base& );
      ~Exemplar() {
         static Base* MakeObj( istream& );
      }
   private:
      Exemplar();
      Exemplar( Exemplar& );
      Exemplar& operator=( Exemplar& );
      static Exemplar* _list;
      Exemplar* _next;
      Base& _obj;
   }

   #define ADD_EXEMPLAR( Base, D )
   /
   static Derived Derived ## _dummy;
   /
   static Exemplar<Base>Dervied ##
   /
   _Exemplar_ ## Base( Derived ## _dummy );

Figure 4. MsgFactory using the Exemplar Idiom
```

Now, let us return to the issue of whether the MsgFactory belongs in the Communications Layer or the Format Layer. Regardless of the layer chosen, the MsgFactory was part of the EXPRESS framework, while the exemplars were part of Plug-ins depending upon the application to be built.

One option was to put the MsgFactory in the Communications Layer. This had appeal since EXPRESS applications often associated a particular MsgQ with a particular bank (hence format). This information helps the MsgFactory find the right reference message set from which new messages can be created.

The other option was to put the MsgFactory in the Format Layer. From a modeling perspective, this is where the MsgFactory – in particular, the exemplars – belonged. But the problem with this approach is that the Communications Layer would then depend upon the Format Layer, which would introduce a circular dependency between subsystems. We ultimately decided to put this into a common library, which was not a great solution, but solved the dependency problem.

Lesson #5: Partitioning functionality across different components often involves compromise. Constraints such as eliminating circular dependencies across libraries or components take precedence over other partitioning issues.

4. TRADE-OFFS

Another aspect of software engineering is optimizing trade-offs, particularly with respect to limited resources and competing goals.
4.1 Entropy vs. Change

Sometimes software teams do the wrong things for the right reasons. Two and a half years into the EXPRESS project, our bank initiated a global IT reengineering program. An external consulting firm was hired to audit all IT projects, including EXPRESS, to ensure alignment with the corporate strategy. Production releases were suspended while employees and consultants worked together for nine months on a major redesign of the EXPRESS framework.

Conflicting views soon emerged on the combined team: the employees represented entropy; the consultants represented change. Over time, we learned that each side held an essential perspective the other lacked. In the end, many things were changed in EXPRESS: some things were improved, some were worse, and some were just different. This section gives an example of how competing ideas can lead to unforeseen trade-offs in design.

In the first generation of EXPRESS, prior to the redesign, the process of translating messages and their fields from one format to another was done in code. The thinking behind the original EXPRESS design, and the rationale for code-based translation, was that the strong type-safety of C++ would prevent software defects. Type safety is a feature of a programming language in which the compiler ensures that the actual arguments passed to a function match the formal arguments declared in the function’s signature. If they do not match, the code does not compile, thereby preventing this class of software defects. Our faith in strong type-safety was confirmed by an early bug.

Some of the early message translation routines were written in C using functions like sprintf(). Unfortunately, sprintf() does not enforce type safety. So, if a pointer to an integer is passed to sprintf() instead of a pointer to a string, a general protection fault will likely occur. While this is a well-known problem in C, programmers compensate by being careful with types. The problem is that people make mistakes. We discovered this bug during user acceptance testing for the first release of EXPRESS. This confirmed our selection of C++, and justified our subsequent expanded use of type-safety.

As illustrated in Figure 5, an ExMsg is a strongly-typed container for all data related to a message. The clone(), Unflatten(), and Flatten() methods are used by the MsgFactory described above. The srcMsg is a Unicode String buffer that contains the original message as received by an EXPRESS application. The tokenList contains the internal representation of the message. The destMsg is a Unicode String buffer that contains the destination message. (Note that EXPRESS does support the ability to translate one input message to multiple output messages.)

Message format translation within the Format Layer of EXPRESS occurs in two stages: first, the input source message is parsed into an in-memory list of tokens; second, that list of tokens is translated into the destination message format. By doing message translation in two steps, we reduced the number of possible translations from n(n-1)/2 to n+1. The trade-off for this reduction in complexity is that each message takes twice as long to process. An example of a source message, in-memory token list, and destination message are presented in figures 6, 7, and 8, respectively.

![Figure 5 ExMsg Class Hierarchy](image)

![Figure 6 A Source Message using a Fixed-Structure](image)
base class. The idea was that any one type of ExMsg could be challenge in that one of the formats (i.e., message sets) was an internal API, which we exported to client applications. The linked in at build time by make files and packaging scripts. Since we needed to load different destination formats (depending upon the destination bank), we had difficulties with separating member declarations from member definitions across two (or more libraries). In hindsight, this was a learning experience as there are we used dynamically linked libraries and overloaded, static functions with two strongly typed formal arguments, one for the srcMsg buffer and one for the destMsg buffer. This worked well to prevent illegal mappings between message types, but didn’t provide recursive mapping into composite messages[7]. Once again, we discovered what we already knew: the software development process involves compromise.

The strength of the code-based approach used in the first generation was that both messages and their translation methods were strongly typed. In fact, the compiler and link editor ensured that illegal mappings (i.e., the translation from an input message to an output message in a different format) were not possible for incompatible message types. This significantly reduced opportunities for software defects allowing us to focus on other issues during testing. We combined this type safety with rigorous regression testing. The result was that the first generation of EXPRESS never had a production software defect with business impact that was due to an invalid mapping. The trade-off for this quality was time: new requirements had to be implemented as code changes, which meant releasing EXPRESS through our Software Development Life Cycle (SDLC). As a result, we did many software releases into production, but each release took considerable effort.

In the second generation of EXPRESS, after the redesign with the consultants, the process of translating messages and their fields from one format to another was implemented using reference tables stored in a database. The rationale for this change was that the corporate strategy favored rapid application development, and databases could be changed rapidly.

Note, that this approach does not take advantage of strong type-safety. Message translation in the second generation of EXPRESS uses data-driven translation by storing a set of translation rules in reference data in static tables in a database. This includes rules that specify data formats, state machine reference tables, which can be changed on the fly. This has the advantage of more flexibility, and the disadvantage that more defects occur in quality assurance testing. With sufficient testing, the second generation has yielded flexible results and is currently used by EXPRESS.

Lesson #6: The risk of failure during production releases is inversely proportional to the frequency of releases.

Lesson #7: A big advantage of strongly-typed languages such as C++ is type safety. The compiler and link-editor prevent incompatible types from being used in unintended ways. Leverage this feature to reduce opportunities for software defects.

Lesson #8: After two or three years into a project, it can be valuable to introduce new team members with different perspectives provided they do not introduce change for the sake of change. Properly introduced, such change ultimately encourages developers to put aside the not-invented-here prejudice and to be open to new ideas.

Lesson #9: Software development teams sometimes respond to pressure for aggressive development schedules by putting programming logic and static, reference tables into databases. Unless it is controlled carefully, this approach is bad practice for two reasons: it bypasses strict, corporate deployment lifecycles by encouraging developers to maintain production control logic; and it introduces the possibility that human error could lead to integrity problems, which could lead to unpredictable program execution.

4.2 Accuracy vs. Time

This is an example of a run-time trade-off. EXPRESS supports validation methods in object constructors that can be turned on or off recursively at run-time to balance current traffic volume and throughput. (The latter is adversely affected by validation processing.) In the Securities Industry, straight through processing (STP) – the automation of back-office functions with minimal human intervention – can take considerable processing resources. This is particularly true for exception transactions which require off-line research to resolve. We initially planned to support a variety of STP scenarios. After a while, we realized that the 80:20 rule applied since we were spending too much effort on features that were ultimately did not use. This is because we found other approaches to address performance issues through a combination
of process configuration (described below) and load balancing across processes. Thus, controlling accuracy as a quality parameter became less important as we had other ways to address performance.

Lesson #10: Always seek to solve the most important problem first, followed by the hardest problem second. (Generally, the hardest problem is also the most important problem.) Drop everything else and repeat this cycle. Tackling the hardest problems early helps define the scope of the project.

4.3 Flexibility vs. Complexity
This is an example of a build-time trade-off. At start-up, express can be configured for either horizontal or vertical scaling. With horizontal scaling, each protocol layer within EXPRESS (e.g., the Communications Layer) is configured as a cluster of processes load-balanced across loosely coupled server hardware on the same LAN segment. This model is appropriate for deployments in data centers that need to support high On-Line Transaction Processing (OLTP) volumes. Conversely, with vertical scaling, a single process boundary encapsulates the entire protocol stack. Multiple processes are configured on shared hardware with each process handling a limited number of information channels. These channels, in turn, map to specific business functions (e.g., stocks vs. mortgage-backed securities).

Lesson #11: Enforce architectural constraints early; defer configuration changes as long as possible. This maximizes opportunities for discovering unforeseen applications while maintaining the quality controls and structure of a well-defined architecture.

Figure 9 Horizontal Scaling

Our initial thinking was that most deployments would use horizontal scaling as depicted in figure 9. It turns out that we made more use of vertical scaling, shown in figure 10, because it addressed an important customer requirement: each new product line had to be released independently of the rest. For example, high volume from mortgage-related products (e.g., pool allocation day) could under no circumstances adversely affect the processing of government securities (e.g., treasury bonds) which had tight processing deadlines. We found that this requirement outweighed the performance penalties of vertical scaling. Thus, we bundled product-related feature sets into vertical process groups. The key point is that this engineering trade-off was implemented as a configuration option, not a design limitation.

Lesson #12: Reuse for the sake of reuse can be counterproductive. Instead, focus on good architecture, understanding the cost of reuse, and applying it appropriately.

4.4 Reuse Considered Harmful
Two aspects of designing for reuse are particularly challenging: not knowing how software being developed today could potentially be reused tomorrow; and determining the right balance of granularity among development units (i.e., lightweight objects, big components, entire systems). While reuse is generally desirable, there is a point beyond which reuse yields diminishing returns.

Six months into the beginning of the project, we made a controversial decision which later became important for survival. EXPRESS was originally part of a larger project to build a next-generation clearance and settlement system. There were many arguments made for both projects to share a common code base. The EXPRESS team unilaterally decided to eliminate all dependencies on shared code. The basis for this decision was that B2B application integration was functionally distinct from a clearance and settlement system. This decision was not without criticism. But, the intent was clear: EXPRESS would evolve independently of other projects.

During the two years that followed, several EXPRESS applications were put into production with a new release approximately each quarter. The clearance and settlement system, however, had not been rolled out into production. Although its scope was larger and its design was brilliant, the clearance and settlement project was ultimately cancelled. While we did not know it at the time, the decision to eliminate dependencies on libraries shared by both projects spared EXPRESS from sharing the same fate. It also positioned EXPRESS to become an enterprise service.

Lesson #12: Reuse for the sake of reuse can be counterproductive. Instead, focus on good architecture, understanding the cost of reuse, and applying it appropriately.

5. MEASURING GROWTH
One of the quality parameters of applications frameworks is the rate at which they grow. This drives the velocity of the project. We defined a growth index as $z = \frac{\text{yield}}{\text{effort}}$, where yield and effort can be measured in several ways such as function points and actual resources expenditures, respectively. This is analogous to return on investment. This metric measures the change in yield over time after considering the up-front investment in developing the application framework. The growth index describes a framework’s ability to leverage previous work (via software reuse) when building new deliverables. After five years of production...
releases, we concluded that application frameworks tend to go through three stages of growth.

1. During the first stage, the growth index is less than one. Most of the time and resources are spent building the framework itself. For EXPRESS, we developed a few Plug-ins with the first release to create a useful, production deployment (allocating collateral for the finance desk). This first product release was important for our project sponsors.

2. During the second stage of the framework growth lifecycle, the growth index is approximately one. Most, if not all of the framework development has been completed, but the original team is still in the critical path of enhancements. Thus, yield and effort are linearly related. Stated in a more meaningful way, yield is constrained by resources, time, and budget.

3. During the third stage, the growth index exceeds one. As an application framework matures, the development team can leverage its customers’ resources by helping them tailor the framework to the their specific needs. At this stage, customers can perform self-provisioning and self-service, which frees up the original development team to pursue other activities such as enhance the framework. Customers generally favor applications that are customizable and have mature administration facilities, especially when it decreases their time to market.

As the EXPRESS project grew, we had some success in realigning our team into an internal consulting role. This did accelerate delivery by leveraging resources in other departments. Unfortunately, we did not consistently track the growth index and other metrics across releases. As a result, we had some difficulty making the case for additional hardware and software.

**Lesson 13:** One of the challenges of building application frameworks that span different departments is that no one group wants to subsidize the development costs of a framework that benefits a competing department. Therefore, it is especially important to track the growth index and tie the results back to the long-term business case for the project. The case for proper use of metrics in software engineering is well-established.

### 6. CONCLUSION

EXPRESS was implemented using object-oriented technology because this was the best approach to satisfy the business requirements. Having an application framework as the basis for our architecture helped us to meet short-term deliverables and adapt to longer term needs as they emerged. Over the course of five years, EXPRESS changed considerably. As EXPRESS matured, we discovered unforeseen applications, both big and small, such as the following:

- Daisy-chaining the output of one EXPRESS application into the input of another helped us scale to larger deployments.
- Batch jobs for file extracts, conversion, and transfer (typically among banks).
- Bank emulators – EXPRESS can be configured to read archived data from files or databases and then send the traffic to another bank. This process can work both ways creating a source or sink for message traffic. Regression tests can be built using production logs of previous runs.

- Miscellaneous utilities used in shell scripts such as filters, file pre- and post-processors, and message routing among internal applications

While this paper has focused on using application frameworks for application integration, there are many other uses for application frameworks. These include both domain-specific and domain-independent frameworks. One theme that emerged from our work is that as integration standards evolve up the protocol stack, services at lower levels become commodities. New opportunities exist for commercial products that can address the challenging problems at the application layer, in particular, business semantics integration. Examples of industry efforts in this area include ebXML[8] and RosettaNet[9]. Finally, the following references provided valuable insight: [10], [11], [12], and [13].

In summary, object-oriented, application frameworks can add great value to long-term projects that involve many releases of a set of related applications. For further information on application frameworks, visit www.metatech.us.

### 7. REFERENCES
