Batch Control Application Frameworks and Reuse

Allen D. Benton  
Consulting Application Engineer  
ABB Automation Inc.  
650 Ackerman Road  
Columbus, OH 43202  
Tel. (614)261-2506/Fax (614)261-2621  
al.benton@us.abb.com

KEY WORDS
Application Reuse, Object-oriented design, Frameworks, Domains, Patterns, Components

ABSTRACT
ABB Automation has adopted elements of object-oriented analysis and design in their batch control projects in order to take advantage of the reuse potential inherent in the ISA S88.01 model. Here are some architectural and process tradeoffs that anyone should consider before doing their first project:

1) Define modules at a level of granularity that optimizes reuse without obscuring the process.
2) Be prepared to manage significantly smaller and more numerous modules as compared with traditional procedural decomposition.
3) Give careful attention to module interfaces and communications between modules.
4) Begin with solution patterns that have proven to work in the real world.
5) Think beyond code reuse to design and test documentation reuse for quicker payback.

The goals of this paper are: 1) provide a common perspective and terminology for discussions on application reuse, 2) discuss examples of object-oriented analysis and design in batch control and 3) show how most batch control products can be made to support this model, even if they do not utilize object-oriented application languages.
INTRODUCTION

Object-Oriented (OO) Design and Application Reuse are two topics that arguably should be addressed separately. Certainly, many books have been written on either discipline without even a mention of the other. One purpose for combining them is that in the absence of an object-oriented programming language for process control, many developers have been overlooking potential rewards of being early adopters of these technologies. Another reason is to demonstrate the synergy inherent between them and to erase some of the hype that still exists. As Coggin’s Law of Software Engineering states: “pragmatics must take precedence over elegance, for Nature cannot be impressed”.

Architecture Design

The role of the system analyst is to break complex problems into smaller problems and discover useful abstractions that model the problem solution. There are three general approaches to doing this work. They include:

- Procedural decomposition
- State Machine decomposition
- Object-Oriented decomposition

Procedural decomposition looks at a system as a hierarchy of algorithms, represented visually as blocks with data inputs, outputs and transformations. For sequential control systems, it is assumed that external events trigger block execution. The result can be visualized as a pebble being dropped into a still pond and the resulting ripples representing the system response. For regulatory control systems, block execution is continuous and expected to occur at whatever rate produces a “real-time” environment.

State Machine decomposition views the system as a set of finite states and transitions between states. A system will remain in a given state indefinitely unless a qualified event transitions it to another state. Few developers would apply this approach to the entire analysis task because of limited number of meaningful states contained within most systems. Some common application examples include equipment startup and shutdown, operator interaction, etc.

Object-Oriented decomposition starts with the concept that a system can be described in terms of the physical and logical entities (objects) that make up the system environment. Similar objects are grouped into Classes and the behavior of each class is then analyzed. Here, the emphasis is placed on modeling data and functional relationships as opposed to describing algorithms.

Having used the first two methodologies extensively as part of a “Structured Design” approach to software development, our applications group encountered the following difficulties:

- Procedural decomposition methods lead to solutions that are efficient, but weak architecturally. They often lack abstractions that are sufficiently granular to promote application reuse.
- Using a procedural approach too early often results in significant rework or even re-design. Since our markets are dictating shorter development cycles, accompanied with earlier involvement, this means the risk is increasing.
• Systems created using procedural decomposition are often difficult to extend and maintain because the functionality is so tightly interconnected. Scalability is an important aspect of achieving reuse from project to project.

For these reasons, and more, we have been begun adopting the object-oriented decomposition approach to our batch applications work.

**Architectural Tradeoffs**

The first thing an engineer soon discovers using an OO approach is that the level of module granularity increases. End-users (and project management) should understand that this increase in module count and time to design can be good if the time spent during the implementation and test portion will be less. There are several design guidelines that can help drive solutions in this direction.

• Restrict the scope of any software module to its unit boundaries. This accomplishes two things: 1) it encapsulates the specifics of how each unit performs its functions, and 2) it addresses concurrency issues associated with shared resources.

• Limit the granularity of phase definitions to a level where the process being implemented is not obscured. Again, the ISA S88 definitions for phases help guide this.

• Avoid trying to force-fit all functionality within the Batch Control model. Certain functions, such as unit monitoring and supervision, belong outside the sequential environment. Using the OO approach will reveal which system behaviors to include.

**OBJECT MODEL TERMINOLOGY**

**Software Components**

As software designs become more modular, the need for communications between modules increases. Often a solution will consist of several modules or components that must all be present in order for the solution to be complete. The term “component” is used in its object-oriented sense to describe a piece of software that performs a well-defined function using an interface. The interface establishes a contract between software components. It consists of a data structure and a protocol for using the data structure (methods in OO terminology). The logic used to satisfy the interface is encapsulated in the software. The concept of an interface is the key to creating a library of components that are modular, connectable, and extendible as new requirements evolve. In batch applications, two types of communications are common. They are defined generally as:

1. Peer-to-Peer
2. Client-Server

The differences between these two types are best defined by example. The figure below illustrates how a peer-to-peer interface might be used to implement a tank-to-tank transfer:
In this example, each program has its own interface as shown by the Transfer Status and Receive Status data stores. For the sake of discussion, assume that these stores are implemented using shared memory arrays. Each array contains two elements: 1) Unit State and 2) Unit Data. The sequence of steps, or protocol, to perform a transfer might be as follows:

<table>
<thead>
<tr>
<th>Steps</th>
<th>Transfer</th>
<th>Receive</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Change state to ‘AVAILABLE’</td>
<td>Change state to ‘AVAILABLE’</td>
</tr>
<tr>
<td>2</td>
<td>Wait for state of Receive unit to change to ‘AVAILABLE’</td>
<td>Wait for state of Transfer unit to change to ‘PREPARING’</td>
</tr>
<tr>
<td>3</td>
<td>Compare Receive capacity and level with actual amount to be Transferred. Alarm and wait if insufficient capacity.</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Change state to ‘PREPARING’</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Wait for state of Receive unit to change to ‘READY’</td>
<td>Change state to ‘PREPARING’</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>Open inlet valve(s) and any other IO operations.</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>Change status to ‘READY’</td>
</tr>
<tr>
<td>8</td>
<td>Open outlet valve(s), turn on Transfer pump and any other IO operations.</td>
<td>Wait for state of Transfer unit to change to ‘TRANSFERING’</td>
</tr>
<tr>
<td>9</td>
<td>Change state to ‘TRANSFERING’</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Wait for operation to complete. Monitor Receive unit for state changes. Alarm if operation takes too long to complete.</td>
<td>Change state to ‘RECEIVING’</td>
</tr>
<tr>
<td>11</td>
<td>Change state to ‘FINISHED’</td>
<td>Wait for state of Transfer unit to change to ‘FINISHED’</td>
</tr>
<tr>
<td>12</td>
<td>Close outlet valve(s), turn off pump</td>
<td>Close inlet valve(s) and other IO operations.</td>
</tr>
</tbody>
</table>
Table 1 - Component Peer-to-Peer Interface Protocol

Note that the interface allows the programs to perform a reasonably I/O-intensive operation (shaded steps) without being aware of, or controlling, each other’s I/O. The Transfer program does not need to know which inlet valves to open on the Receive unit. The Receive program does not need to know if the Transfer program will perform a gravity-based, pump-based or pressure-based transfer just to name some examples. The contract guarantees that if the protocol is followed using the interface then the transfer will be successful.

An example of Client-Server components might be an application where reagents are drawn from a shared header unit. The figure below illustrates how this interface might be used to implement a Header Charge operation:

Figure 2 - Component Client-Server Interface

The command array contains several elements including: 1) Client Unit ID, 2) Client Command, and 3) Command Parameters (charge amount, charge rate, etc.) The status array contains the same elements: 1) Unit State and 2) Unit Data. The sequence of steps, or protocol, to perform a charge might be as follows:

<table>
<thead>
<tr>
<th>Step</th>
<th>Server</th>
<th>Client</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>On startup, set all I/O to a safe state and set state to ‘AVAILABLE’</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Wait for ‘ACQUIRE’ Command</td>
<td>Wait for ‘AVAILABLE’ Server State</td>
</tr>
<tr>
<td>3</td>
<td>Send ‘ACQUIRE’ Command to Server.</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Change state to ‘ACQUIRED’</td>
<td>Wait for ‘ACQUIRED’ or ‘PREPARED’</td>
</tr>
<tr>
<td>Step</td>
<td>Server</td>
<td>Client</td>
</tr>
<tr>
<td>------</td>
<td>--------</td>
<td>--------</td>
</tr>
<tr>
<td>5</td>
<td>Execute unit PREPARE logic. Change state to ‘PREPARED’</td>
<td>Server state</td>
</tr>
<tr>
<td>6</td>
<td>Wait for ‘CHARGE’ Command from Client</td>
<td>Setup unit device(s)</td>
</tr>
<tr>
<td>7</td>
<td>Send ‘CHARGE’ Command to Server, also send Amount Requested and Valve Position</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Check Amount requested against current level</td>
<td>Wait for ‘CHARGING’ Server State</td>
</tr>
<tr>
<td>9</td>
<td>Reset totalizer to 0, Flow control set point to Valve Position and Charge Amount. Change state to ‘CHARGING’</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Monitor unit loops and devices and check for ‘HALT’ Command. Wait for FLW_CTL to cutoff. Change state to ‘FINISHED’</td>
<td>Monitor unit loops and devices, and check for ‘FINISHED’, ‘HALTED’ or ‘OUT OF SERVICE’ Server State</td>
</tr>
<tr>
<td>11</td>
<td>Reset unit loops and devices</td>
<td>Reset unit devices</td>
</tr>
<tr>
<td>12</td>
<td>Wait for ‘RELEASE’ Command from Client</td>
<td>Send ‘RELEASE’ Command to Server</td>
</tr>
<tr>
<td>13</td>
<td>Change STATE to ‘AVAILABLE’</td>
<td></td>
</tr>
</tbody>
</table>

**Table 2 - Component Client-Server Interface Protocol**

Component interface design consists of more than just creating lists of data elements. There are several factors that should be considered and tested extensively. They include:

- **Latency** (real-time performance)
- **Shared Memory** (channel)
- **Indeterminacy** (failure recovery)
- **Concurrency** (multi-client)

Latency deals with the timing between when one component places data on the interface and another component responds to it. Timing issues occur when the receiving component responds either too fast or too slow. The issue of shared memory is usually inherent in the application framework being used to implement the system. Some frameworks provide the concept of a queue that allows multiple clients to interact with a single server. Most do not, so work-arounds need to be developed to prevent clients from overwriting each other’s commands. Indeterminacy addresses the question of what happens when things go wrong such as aborting a batch or when a server cannot satisfy a client request temporarily. Finally, concurrency deals with situations where multiple clients need to interact with servers simultaneously or one at a time. Even though the process may be piped for shared usage, the software must accommodate this behavior.
Software Patterns

When component software is defined using interfaces, applications take on a familiar look and feel. This familiarity is referred to as a “Pattern”. These patterns can be obvious, as in the case of transfer and receive, or more complex. For example, distillation involves controlling three units as shown in the figure below:

![Figure 3 - Domain-Specific Pattern - Distill Operation](image)

This module uses a Vacuum Services unit to pull vacuum in both the Source and Receiver units. Condensate from the Distillate Source is drawn off into the Distillate Receiver unit. The operation stops when the level in the Source reaches a prescribed target. The Source and Receiver programs have an interface almost identical to Transfer and Receive. The interface between the Vacuum Services and Distillate Source programs is much closer to a classic Client-Server model with a command-driven protocol.

The figure below shows a completely different module used to wash a tank:

![Figure 4 - Domain-Specific Pattern – Tank Wash Operation](image)

A Solvent Dispensing (Charge) unit is used to flush an empty tank and the liquid from this tank simultaneously drains into a waste tank. It should be intuitive from these examples that patterns contribute immensely to reuse within an application. In practice, design documentation is needed at the pattern level, as well as the component level, in order for this type of reuse to succeed.
**Application Frameworks**

The ISA S88 Batch Control Systems standard has established general agreement among system developers on how to describe batch control problems and to structure their solutions. From this foundation, several software products have emerged which provide a framework for solving batch control problems in a cost-effective and reliable manner. The term “framework” is used here in its object-oriented sense because one of the outcomes of the standard was a vocabulary that readily lends itself to object-oriented design and implementation. One definition of “a reusable, semi-complete application that can be specialized to produce custom solutions.” Before going any further, let's look at how Batch Control fits in a high-level framework:

![Batch Control Application Framework](image)

**Figure 5 - Batch Control Application Framework**

The first sub-system of interest is the Batch Recipe. It contains the process data and step information necessary to make a product. Another important sub-system is the Batch Library. The library contains definitions of the available equipment and executable phases. The Batch Record sub-system maintains a history of what actually occurred during manufacturing of a product. Finally, the Batch Schedule sub-system is responsible for ensuring that products are manufactured in a systematic and efficient manner.

Frameworks are not restricted to the system level of operation. It is useful to define frameworks-within-frameworks in order to achieve reuse at multiple levels. An analysis technique that assists this process is called Domain Analysis. A Domain can be thought of as a cohesive group of problems that have been studied in detail in order to identify components and patterns that combine to form a robust solution. Domain Analysis can be applied across similar applications (vertical domain analysis), as well as related parts of the same application (horizontal domain analysis). The examples shown previously exist in the horizontal application plane.

This organization of components into frameworks is an important step in assuring the likelyhood that will occur.
APPLICATION REUSE LIBRARY

In the past, reuse within our batch control group has been ad-hoc and consisted primarily of a library of generic or parameterized software modules and subroutines that were customized or used as-is on new projects. Our experience shows that a code-centric approach addresses only a small percentage of the total output of a development effort (analysis, design, code and test). Furthermore, it is not able to address solutions that exist only within a narrow domain. Defining a domain, or context, for a solution allows the designer to avoid having to needlessly over-generalize a solution in order to add it to a reuse library. This requires design level documentation be included so that other developers can understand the solution context.

We can see now that an application can be defined in terms of frameworks, domains, components, interfaces and patterns which incorporate content from all stages of a project. The potential for reuse increases greatly as batch control applications become more modular. The following figure shows what a Reuse Library might look like:

![Figure 6 – Application Reuse Library](image)

The Analysis area is concerned with identifying and documenting Domain-specific frameworks. The emphasis is on modeling requirements for the application. The Design area is focused on defining Patterns, Components, and Interfaces within one or more domain. Quality Components and Patterns exhibit the dual properties of good coupling and cohesion. The designs should be documented using pseudo-code or sequential function charts that make the design as implementation independent as
possible. The Code area contains all implementation in the form of source code files. Finally, the Test area contains test plans and procedures used to verify that code conforms to the functional specification.

The hierarchical nature of frameworks, patterns, and components (represented by the dashed connecting lines) suggests that most library searches would not need to go below the Analysis or Design level in order to identify reuse potential. All of the associated Source and Test documents would be linked directly to the more abstract components.

CONCLUSION

Application frameworks are collections of software components that are assembled through the process of exploring application domains and the software patterns found within them. Some frameworks are domain-neutral, meaning the opportunity for reuse is potentially greater. One of the keys to insuring that reuse occurs in practice is to create application reuse libraries that contain products from the entire software lifecycle and not just source code.

The practice of developing component software has clearly matured to the stage that it has proven its potential for accomplishing reuse in non-trivial applications.

REFERENCES

2. Johnson, Ralph E. Frameworks = (Components + Patterns), Communications of the ACM 40, 10, (October 1997), 39-42.