Building Plug & Play Power Applications Using Abstract Object Modeling

Marck Robinson
marck@powerdata.com
PowerData Corporation
15193 SE 54th Pl
Bellevue, WA 98006

Chris Mosier
chris@powerdata.com
PowerData Corporation
15193 SE 54th Pl
Bellevue, WA 98006

Fabiola de la Peña
faby@powerdata.com
PowerData Corporation
15193 SE 54th Pl
Bellevue, WA 98006

Robin Podmore
robin@incsys.com
Incremental Systems Corporation
194th Ave SE
Issaquah, WA 98027

Abstract

This paper addresses the computer industry’s general inability to develop complex software applications in a manner that promotes easy integration, reusability and low maintenance. Most complex applications are built for a very specific environment and include far too many dependencies, making them difficult to maintain and expensive to integrate. A development methodology known as Abstract Object Modeling is discussed that overcomes this limitation and a simple power application known as a linear powerflow is used as an example. The powerflow is first discussed in terms of a traditional approach, then in terms of the abstract object modeling.

Introduction

Developers of complex systems are faced with a very difficult dilemma every time they set out to solve a problem: How can existing tools and technologies be best utilized to arrive at the most effective implementation of the solution? Picking the wrong approach at this critical juncture could mean the difference between a solution that stands the test of time and one that simply gets forgotten as technology evolves. It can also mean the difference between a solution that can be reused in a variety of situations or a solution that exists primarily stand-alone.

Traditionally this has been a very difficult decision processes to go through; which database, which operating system, which hardware, what kind of middleware, what type of user interface. It is at this point that the solution to the problem being solved starts to be imbedded into a specific set of technology, eventually becoming indistinguishable from that technology. Each assumption made at this stage becomes a dependency, each dependency becomes a barrier that limits deployment, hinders future integration, creates potential maintenance and devalues the resulting solution. In addition, the solution ages and grows out of date as the technology it depends on grows out of date, regardless of the robustness and applicability of the algorithm itself.

Having dependencies of this type are often viewed as such a severe constraint that many researchers and vendors choose to build their own environment as well as their own applications. This too has pitfalls. A complete and mature set of infrastructure is very expensive and time consuming to build. It can tie up valuable resources and forces developers to become experts in many areas when their passion is really focused in a few specific areas. More often than not the result is a compromised set of infrastructure, built to the minimum specifications needed by the supported applications and not robust enough to allow the true potential of the primary work to be realized.

This paper presents an alternative approach, using abstract object modeling, that allows a developer to build applications that can take advantage of other technologies, but not depend on them. It allows applications to be combined in ways never imagined by their authors and it insulates the investment in the solution from the specific technologies available at the time of its implementation.

In order to illustrate this process we will examine the development for a linear powerflow application. First examining an implementation based on a more traditional approach, followed by an examination of that same implementation based on the abstract object modeling approach. It is important to remember that the linear powerflow application is chosen as an example because it is simple to illustrate these principles, however, any component in a system can be designed this way, including the user interface and database systems.
**Powerflow Requirements**

The linear powerflow application will be expected to run continuously, reacting to any change in the system conditions by calculating the branch flows and angles. It will do this by examining each island in the system one at a time. If the island is energized it will examine each bus in the island to find out the total load and total generation on that bus. Then each branch will be examined to find out the impedance and reactance. From this all the bus angles will be calculated along with the branch flows. These results will be stored in a way that they can be viewed and utilized by other applications.

One very common traditional approach is to use text files to pass the data in and out of the application. This is a convenient and appropriate approach in many situations, especially where batch oriented processing is acceptable. However, we are attempting to raise the level of integration for this example application so that it can immediately react to system changes and is able to process only the changes that it needs rather than starting over every time. This requires us to integrate the application more tightly and does not permit the text file approach.

The resulting powerflow application has the following basic interface:

**Traditional Approach**

Using a traditional approach the powerflow application will need to integrate tightly with the system containing the real-time information. This will require the developer to answer some important questions:

- What API is used to monitor for events?
- What API is used to access the data?
- How is the data formatted?
- How does it need to be altered before it can be used by the powerflow application?
- Where should the results be stored?
- How should the results be formatted to meet the needs of the data model?
- Is a middleware used or is the access local?
- What performance issues are there to worry about?

The powerflow application from Illustration 1 is adapted according to the answers obtained above. The following diagram illustrates the general changes that are needed. The sections marked in yellow will need to be maintained any time that the database model is changed. The sections marked in green will need to be changed any time the data storage facility is changed, or if middleware is needed.

**Abstract Object Modeling Approach**

Abstract Object Modeling takes a different approach, it is based on the idea that the application developer can assume the application is interfacing to sets of virtual objects. These virtual objects can have any definition in terms of behavior and attributes that the developer chooses. What objects really exist in the system and where they exist is not really important and is resolved later.

This may sound rather extreme until one compares this
approach to the already very popular practice of using text files for data exchange. It is not very unusual for an application to specify a file format for input, how that file is generated is of no consequence to the application, just as long as it meets the format requirements. The developer may also share results through one or more output files, the format of which is assumed by the application developer, again leaving unanswered how that output file will be blended into the overall operation of the system. As long as these files are clearly documented this is usually an acceptable practice.

Text files can also be very successful in representing a set of objects in a very simplified manner, just imagine a text file for specifying the busses for the powerflow application, which could be formatted as:

```
BusNumber, TotalLoad, TotalGen
1, 100, 0
2, 0, 100
```

This simple comma separated value file could be seen as a set of two bus objects. Of course, these bus objects don’t have any behavior, only state. However, this is still a very useful representation.

It is also easy to imagine that this representation could be generated by tools from any system where a powerflow application was needed. In other words, it is perfectly reasonable for the developer to simply assume that these objects can be produced as specified without needing to know any detail about the target system ahead of time.

Another aspect of using text files is that nearly every computer language on any platform can read or write a text file that can be used by any other language on any other platform. This type of consistency is very important because it means that no special tools or facilities are being assumed by the developer that can become out of date or unavailable as the application is moved from system to system. There are other issues, such as 16 bit character sets versus 8 bit, but in general these practices work without any hitches.

Looking at this another way, the text file approach allows an application to use a simple, consistent API across any system from any language to represent any type of object. This is an interesting observation when compared to object oriented systems such as CORBA or COM where each class of object is expected to implement a different API.

Each class having its own API is quite informative when studying the interface, however, this creates a lack of consistency which prevents the construction of tools that filter, re-map and dynamically generate objects. Where systems constructed around a consistent API can do this quite naturally. This becomes a significant point that is a key component in the theory behind abstract object modeling. Using text files as an example of this principle, there are any number of generic tools and utilities that can filter, parse, read or write the bus objects stored in the simple comma separated value file that was presented above. The API for abstract object model based applications preserves this same principle.

In most respects text files seem like a viable answer to our plug and play interface. However, as useful as they are there are still significant limitations that make many implementations difficult, such as asynchronous events, shared access, locking, high-speed access, and lack of object behavior.

This is where the abstract object modeling approach helps take these concepts to another level. In much the same way that a text file can describe a set of virtual or abstract objects, the abstract object modeling API can as well. However, the abstract object modeling API goes much further, allowing access to individual attributes, creation and destruction of objects, event notification, introspection and access to behavior. These operations are all performed through an API that never changes regardless of the makeup of the objects.

The benefit is that the application can rely on only one API and assume one data model regardless of how and where it is being integrated. These logical views doesn’t change, no matter what middleware, persistent storage system or data model is being used in the target system. Instead, these views are mapped to the target system during integration using tools and utilities that are external to the application.

When designing a system using abstract object modeling, the first step for the application developer is to define the sets of objects that will be required by the application. These are the only external objects that exist from the application’s point of view. For the powerflow application, the classes Busses and Branches as described below:

<table>
<thead>
<tr>
<th>Bus ID</th>
<th>IslandID</th>
<th>TotalLoad</th>
<th>TotalGen</th>
<th>Angle</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>100</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>100</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Table 1 – Busses**

The Busses class attributes are:

- BusID – unique identifier for the bus
- IslandID – island the bus belongs to
- TotalLoad – all load on the bus
- TotalGen – all generation into the bus
- Angle – bus angle, output by powerflow
The Branches class attributes are:

- **BranchID** – unique identifier for the branch
- **PBus** – from bus
- **QBus** – to bus
- **R** – resistance
- **X** – reactance
- **Flow** – branch flow, output by powerflow

Table 2 – Branches

<table>
<thead>
<tr>
<th>Branch ID</th>
<th>P Bus</th>
<th>Q Bus</th>
<th>R</th>
<th>X</th>
<th>Flow</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0.1</td>
<td></td>
</tr>
</tbody>
</table>

The next step is to implement the powerflow assuming that these classes exist just as we described them. The following diagram illustrates the powerflow application with its new interfaces. Notice that the yellow data manipulation section has been removed since our object definitions reflect the objects in the exact format that we want to use them in.

Abstract Object Model Integration

Integration of abstract object model based applications may seem like a difficult task, but it turns out to be quite reasonable. Somehow the abstract objects used by the application must be mapped into the real objects managed by the target system. In order for this to happen the integrator must ask some important questions:

- What API is used to monitor for events?
- What API is used to access the data?
- How is the data formatted?
- How does it need to be altered before it can be used by the powerflow application?
- Where should the results be stored?
- How should the results be formatted to meet the needs of the data model?
- Is a middleware used or is the access local?
- What performance issues are there to worry about?

In fact, this list is exactly the same list that the application developer had to ask in the on the Traditional Approach. If the same issues still need to be addressed, then what is the benefit of the abstract object modeling approach?

The primary benefit is that the application is now independent from any implementation technology except for the language it is written in. The application developer is able to focus only on the core application functionality, without any concern for future deployment issues, yet the resulting application is able to be integrated into any conceivable system.

The application integrator has the advantage of a well defined API and should receive documentation of the abstract objects assumed by the application from the application’s author. The API is an important aspect because it is the same no matter which objects are being represented. This is a similar idea to a relational database. The database API is the same no matter what type of information is logically represented in the tables.

The primary difference between a relational database and the abstract object model approach is that instead of tables the abstract object model API interacts with fully functional objects, including behavior as well as attributes. However, the principle remains the same.

The integrator can approach the integration of a specific abstract object model based application in a number of ways:

- Write custom adaptors that communicate with the database or middleware and then present the information as the application expects. This would be a similar amount of total work to the traditional approach, but with the added benefit that it does not require any involvement from the application developer.
- Write adaptor tools that dynamically map the information from the database or middleware based on configuration files. This allows an integrator to add applications without requiring any new code.
- Purchase third party tools that dynamically map between common data sources, such as Oracle or Ingres databases, comma separated value files, and...
wrap common middleware such as CORBA, COM, XML-RPC or SOAP. Such tools can be obtained from PowerData Corporation and EPRI today and most likely other vendors in the future.

The integrator also has an economy of scale situation, in that the target system interfaces only need to be resolved a single time. For example, if the target system uses Oracle then only one Oracle abstract object model wrapper is needed regardless of the number of applications being integrated.

Existing Abstract Object Model Applications

At this time a number of applications, databases, and user interfaces have either been developed using abstract object modeling or adapted using a wrapper so they can participate in an abstract object modeling system. This work is on-going and the list continues to grow rapidly.

Applications:

- EPRI Operator Training Simulator
- EPRI Topology Processor
- EPRI Linear Power Flow
- EPRI IPFLOW
- EPRI PSADD
- PCA State Estimator
- PCA Operator Power Flow
- Mitsubishi Electric SCADA FEP

User Interfaces:

- PowerVisuals Java base GUI
- Web based utilities

Databases and Persistent Storage:

- PowerData real-time database
- Oracle
- Ingres
- Comma Separated Value Files

Communications (provides a transparent bridge so that an application running on one machine can access objects on another):

- XML over TCP/IP
- CORBA

Conversion Utilities (provides re-mapping services to dynamically convert objects from one representation to another):

- Join Engine
- Expression Engine
- GroupBy Engine
- Filter Engine
- Masquerading Engine
- Sort Engine

Miscellaneous Utilities:

- Security Engine
- Journaling Engine
- Generalized Import/merge Engine
- Generalized Export Engine

Conclusions

The primary objective of an application developer or researcher is to transfer their knowledge in a particular area into an automated form. If this automated result is well packaged and without dependencies it can be reused as part of an ever increasing basic infrastructure that is available to others. However, this objective is rarely realized do to the difficulty in building truly plug & play complex applications.

The abstract object modeling approach is a methodology that overcomes this difficulty, separating the integration problem from the application development problem, resulting in truly re–usable and greatly simplified applications. It is a methodology that applies equally well to user interfaces, database systems, communication systems and application engines. It is also not limited by platform or implementation language. As a result, systems become a collection of building blocks, each provided by different experts without any prior coordination. The system integrator is able to mix and match these blocks to create a system that meets their needs.

Acknowledgments

PowerData Corporation and Incremental Systems would like to acknowledge the work of many other organizations that have helped to contribute to the success of this approach. These companies are: EPRI, Decision Systems International, Mitsubishi Electric, Metso Automation and Best Systems.