Alternative Technologies

On The Declaration of Integration

David McGoveran

EAI has experienced something of a revolution since 1995. Prior to that time almost all application integration efforts were point-to-point and highly custom efforts. Needless to say, they were very costly. Perhaps only those of us who worked on such projects will remember that they also required considerable effort to maintain. Without the benefits of a common infrastructure, standard interfaces, integration brokers, and so on, function changes to applications could easily damage the integration.

By introducing standards and shared integration components, the level of abstraction was raised and it became possible to consider strategies for large scale integration efforts. While we may debate the relative benefits of various integration architectures (point-to-point, hub, multi-hub, distributed, etc.), they each serve to separate the means of accomplishing a goal from the goal itself. By creating integration services (messaging, data transformation, routing, etc.) with well-defined interfaces, we begin to gain obtain the benefits of architectural abstraction. Arguably, it is this encapsulation of generalized and reusable functionality that has made the EAI revolution possible.

As many have discovered, there are further methods by which we can improve upon this abstraction. Rule-driven methods can be used to change the behavior of integration services. Often changes can be made while systems are online, greatly lowering deployment time and costs, and offering tremendous flexibility. For example, rule-driven routing enables implementation and control of message flows without requiring shutdown, programming, or re-deployment. As a further example, process model driven integration as found in a BPMS (business process management system) takes this to a higher level of abstraction, enabling the flow of *business* events and activities to be described and implemented graphically. Although few BPMSs are so sophisticated, it is now clearly possible for someone with no knowledge of computing to define and implement business processes.

With every increase in the level of component abstraction and each move toward a distributed services architecture, the ability to deliver a function with fewer design and development errors has improved. This not merely a consequence of simple component reuse: It is also one of making the relationships among components simpler so that they can be combined more reliably.

Certainly we have come a long way from the days when application integration involved so much detailed work. Many mission critical applications were then developed with APIs, and these had to be added to each program before communication between programs could be

considered. Adding an external API to a program that was not designed to have one was a difficult task. It meant finding a place in the code that would provide access to data while maintaining integrity and performance. Once this was done, all today's familiar problems of synchronizing data formats, semantics, and events between programs remained but were solved through custom code rather than integration services. On the one hand, we now have the advantage of being able to purchase packaged enterprise applications and so no longer have to maintain these complex programs. On the other hand, we have given up access to the special knowledge of internals and so are entirely dependent on the vendor to provide the appropriate set of APIs.

Even with all the advances, the fundamental problems that drove application integration in the first place have not been solved. Initially, the need for timely response to business events meant that batch data transfer between applications was unacceptable. As this problem was solved, ever greater systems interconnectivity (including between businesses and with consumers via the Web) encouraged a change in the nature of business requirements.

Unthinkable a mere fifteen years ago, business managers now seek to permit consumers and business partners to interact directly with business software systems in an effort to become more responsive and presumably more profitable. Indeed, the drive to real-time business interactions and increasingly fleeting business opportunities has created a need to for IT to respond to rapid changes in *strategic business* requirements. In a sense, we are on the threshold of a new world in which IT automation efforts move from well-defined business operations to enablement of transient business strategies. Given the critical importance of this new responsibility, we must understand how best to achieve it. Surely we cannot afford the to repeat the costly software engineering mistakes of the past.

There are a number of problems that have stood in the way of application integration, and remain barriers to IT's ability to respond to the new requirements of business.

- Resource Dependence

  Every programmatic dependence upon the specific characteristics of a resource introduces not only fragility, but an opportunity for error in translating business requirements. There are several types of resource dependence that need to be avoided.

  - Physical Data Dependence

    Physical data dependence is dependence on specific data organization including structure and location. There are many reasons that physical data organization and their associated access methods may need to change. The trade-off between storage efficiency and access efficiency is a well-known problem, so that selecting one over the other may require a change to data organizations. There may be many potential access methods for reading, modifying, destroying, and creating, each of which has different performance and memory characteristics. Ordinary encapsulation goes a long way toward solving the problem of physical data independence, but does not provide selection of the best access method for any particular data structure nor the more difficult problem of selecting the best data organization. Solving these problems requires automatic selection via optimization techniques.

- Logical Data Dependence

  Logical data dependence is dependence on how data is presented to users and programs. For example, introducing a new use of data often requires changes to other programs. As long as no information is lost, including the logical relationships among data elements, there should be no barrier to changing data presentation for new users and new programs. Solving this problem requires a standard data presentation model and means to translate operations on this model into operations on any information-equivalent model such as the physical, stored data.

- Platform Dependence

  Platform dependencies include hardware and operating system dependencies. Few software applications, whether packaged or custom, are impervious to changes in platform. Device dependencies such as changing a monitor, printer, or a disk drive, let alone a network, often led to serious application failures and required subsequent program modification until the advent of standard drivers. It is this type of dependency that has driven most of the interest in portability, leading first to language and interoperability standards, and eventually to Sun Java and Microsoft CLI. The problem is far from being solved, although we certainly understand many principles that must be followed.

- Connection Dependence

  Connection dependencies include dependencies on invocation methods, inter-program communication methods, and inter-system communication methods., device connectivity, and precedence relations. Over the years there have been tremendous debates (and many changed programs) over the relative merits of subroutines, in-line procedure calls, remote procedure calls, asynchronous messaging, and so on. Similar debates have ensued over the best physical transport mechanism for inter-program and inter-process communication, including protocols and physical resources. Connection dependencies abound in most programs, although careful layering of code and standards have helped.

- Code Fragility

  Code fragility refers to aspects of coding that are frequent causes of error and maintenance. In the 1950s and 1960s, a number of studies were done to determine the types of coding errors that were most commonly encountered. Among the most common were errors of iteration control (such as coding or modifying loop entry and exit conditions correctly), case control (whether coded using if-then-else sequences or case statements), transfers of control, consistent data typing in assignments, levels of indirection (address versus content), data initialization, and sorting. Let's examine these from the perspective of precedence relationships, complexity, and correctness.

- Precedence Relationships

  Precedence relationships control the order of function invocation and completion. These may manifest either directly through control code, or indirectly through data states. Almost all coded order dependencies are implementation-specific and so can only be checked for errors in the context of the particular algorithms chosen by the programmer. Precendence relationships of this type are closely related to problems of coupling and synchronization. Another type of precendence relationship is inherent in the business requirements. Changing precedence relationships like these almost always requires changes to code, assuming they have been correctly coded in the first place. The problems of enforcing and changing precedence relations within code have not been solved. However, by expressing precedence relationships are rules

- Complexity

  As system size grows, the number of errors increases non-linearly. That is, when it comes to design and development, scalability fails. This fundamental principle of software engineering is clearly related to inherent limits in human abilities to conceptualize and remember numbers of entities at one time and in one context. The solution to this problem is three-fold. First, abstraction (or "chunking") enables complex concepts to be treated as a single entity with understandable interfaces and behavior. Second, orderly decomposition permits a complex problem to be divided into manageable portions. Third, function generalization can be used to reduce the number of such portions. Managing complexity is the genesis of various attempts at efficient software engineering (including top-down, structured design and object-oriented methodologies) as well as certain popular architectural approaches such as componentization and service-orientation.

- Correctness

  Guaranteeing that code is correct is a difficult proposition. Traditionally we have sought to prove correctness through a combination of error checking within the code and software testing, both of which depend on completeness (no holes) or coverage. Error checking tends to be limited to local state correctness, while software testing is more focused on functional correctness (a right answer). Neither is very adept at catching errors due to either run-time environment changes or incorrect business requirements capture and translation during design. Software testing is particularly sensitive to component interaction complexity. When components can interact in almost any order, the ability to test all possible sequences of invocation becomes impossible even when the number of components is still quite small. One partial solution is to require that components (especially distributed components) be stateless and that all shared data be transactional, so that component testing is path independent. Functional

correctness can be enforced with integrity conditions or constraints, although few developers are inclined to code assertions or constraints and object to an imposed performance penalty for those checks.

As noted, various techniques and technologies have been used to address these problems individually. Unfortunately, although a particular language and development environment may reduce some of the impact, all the problems stated above are *inherent* in the use of procedural computer languages. By a procedural language, I mean one which requires the developer to specify how to accomplish a task and with what resources. Procedural languages have at least one of procedural constructs (such as conditions and control loops), physical resource specifications (such as structured data definitions), or procedure definition and invocation capabilities. As much as we all love to design and develop software systems, and as much as our employers may enjoy paying for them, we have to find a better way to integrate enterprise applications than the use of procedural languages.

In overview, the use of procedural languages creates three major problems, broadly speaking. First, the more procedural the grammar rules, the more difficult the language to learn *even* if the language is graphical (consider modal versus non-modal graphical user interfaces). Second and as discussed above, procedural elements tend to be the source of errors, causing high maintenance costs and functional rigidity. Third, because procedural elements expose physical organization and structure, changes to that physical organization and structure cause costly and error prone maintenance efforts. Fourth, all these taken together mean that integration using procedural languages does not meet today's rapidly evolving business agility requirements.

Most general purpose languages are procedural, so that the user must know something about physical implementations and specify precisely how to accomplish each task. If they need to access existing data, they will need to know how that data is physically organized. Such languages have a procedural element to them, meaning that they must be able to take advantage of order. (If that's not obvious, try to imagine the concept of "next" or "previous" data element without those elements being ordered. Then try to imagine completely non-physical ordering.) Of course, the average user won't know how to use the procedural elements of a computer language.

Coming back to a point made earlier, rule-based integration components provide a clue as to how to avoid these problems and deliver flexibility into the hands of those who need it most. For example, dynamic rule-based routing provides the ability to direct a message to a particular recipient conditionally based on message sender, time, content, and so on. In effect, rules are a kind of declarative integrity constraint on valid message routes. These rules can be changed or augmented without shutting the systems down and constitute a relatively simple language to learn and use. Some implementations provide a graphical interface for managing the rules. With a bit of careful implementation, messages correspond to business events and the applications that constitute both senders and receivers are identified with specific business functions. This raises the level of abstraction to something a business user might understand and be able to use.

Model-driven process integration goes beyond using business rules to control the point-to-point message flows and enables control of entire processes. Implementing a process as a complex collection of point-to-point is difficult to conceptualize, and may introduce process integrity problems. By using a graphical process design tool to drive a process engine

(essentially a sophisticated rule-based message router), the level of abstraction is increased further. If both the design tool and the process engine support process abstraction and process independence, it becomes possible to implement business processes in terms of activities and process flows that are understandable to a business user. Such a business process specification is tantamount to a set of activity integrity rules that constrain the correct invocation and correct completion of business activities, and a set of process integrity rules that define the precedence relationships among activities. The system then translates the declarative specification into a physical implementation with physical resources (based on a manually defined mapping in most such products).

The creation of declarative languages as an alternative to procedural languages was driven in part by a desire to avoid (or at least reduce the impact of) the problems discussed above. To understand the power of a declarative language, suppose that users could concentrate simply on what they wanted to achieve rather than on how to obtain it. They would simply *declare* the goal, with both the initial state and the goal being defined by constraints or conditions. This approach requires that the system (not a user or programmer) automatically translate the goal declaration into an optimized set of component procedures that are invoked as needed, and produce a result guaranteed to achieve the declared goal. By contrast with procedural programs, declarative approaches are much easier to understand, easier to write and modify, and are much more succinct. Pure declarative languages have associated with them none of the problems described earlier. Indeed, they obtain the very benefits that procedural languages fail to achieve.

The ideal declarative language for EAI would consist of two parts, one for declaring integration goals and constraints, and one for specifying the characteristics of the physical resources that could be used and object references (names). The execution engine would have an optimizer so that resources usage could be optimized and a scheduler so that resource conflicts would be avoided and load balancing achieved. It would access an active repository to translate user references into physical references. Data mapping and transformation would occur automatically as needed whenever one business function was the precedent of another. The language would have nestable constructs and encapsulation enabling abstraction and controlling complexity. The language itself is would be based on a closed algebra, so that all expressions in it are provably correct and unambiguous.

The example integration facilities just discussed have some of the important features of a declarative language and therein lies their power to deliver on the promise of EAI. The view of the system as seen by the user of a declarative language is in terms and units the user understands. Declarative languages need to be semantically rich so that users can accurately express goals. Indeed, a declarative language is all about semantics or intended meaning. The software that processes a declarative user request must hide platforms, physical data organization, and the procedures that manipulate that organization. This means the fundamental operations must preserve information integrity – information is never augmented, altered, or lost except in ways that are specified explicitly by the user.

In recognizing that there is a difference between the way in which business functions and events are most economically and naturally expressed and the physical implementation, location, and names of those business functions and events, it is clear that a repository mapping these is needed. Otherwise, users of the declarative language would be forced at some point to know the physical implementations. Most uses of directory services (such as LDAP and UDDI)

fail to maintain complete separation of logical and physical, and in fact were not designed for quite this purpose.

When a language exposes physical data organization to the user, its declarative power is degraded. For example, URLs are both hierarchical (and so have inherent order) and physical. Worse, there is no semantic model by which a content goal (based on meaning) can be translated into that physical location. XML and the languages and facilities that derive from XML mimic this organization. Query languages for XML are replete with the language of order: occurrences, sequences, paths, steps, descendants, children, and so on. Just because these languages "have no procedures" doesn't mean they are non-procedural (a naïve understanding of "procedural"): if operation order changes results, the language is procedural. Even SQL now has many procedural elements, its declarative power greatly diminished by the failure of RDBMS vendors to implement physical and logical data independence.

Its not that procedural languages aren't useful, but we should limit their use because of the high price we must pay. The types of facilities and operations required for an integration infrastructure are now reasonably well understood, though the variety of implementations will continue to grow. What is more important is that we now understand the business language that specifies that integration: It is the language of business processes. Although we may not yet have a pure declarative language for describing business processes, but we can take advantage of declarative languages within the implementation of integration efforts. In addition to declarative capabilities within integration tools, there are other opportunities. Certainly every data integration and data transformation effort can use an RDBMS to great effect. Rules engines can be used to avoid most procedural code, and there are now a number of rules-driven development tools on the market. These tools generate or incorporate reliable procedural code based on a declarative specification.

We are now facing a future with high training, platform, and maintenance costs that are aggravated by every use of procedural language. Question why a data transformation requirement should be met with procedural language. Question whether or not you have resource independence. Will falling back to a manual activity implementation when the application server goes down break your process integration? If you are using web services for integration, the next time you encounter a broken Web link or a page that no longer contains the information pointed to, or have to change links or queries (whether SQL or X-Query) when you reorganize your data, dream fondly of declarative languages. Better yet, insist on using them at every opportunity.

.