Alternatives for Rule-based Application Development

By Nelson Lin

Disclaimer: the information contained in this document represents the author’s view of the issues discussed as of the date of publication. This view may change substantially from time to time based on more understanding of the individual products, their methodology, and their newer releases. Because the author has to respond to changing product releases and information gathered, it should not be interpreted to be a commitment on his part, and he cannot guarantee the accuracy of the information after the date of publication.

Summary

When considering software development tools to develop the next killer application for your enterprise wide deployment, you should consider the following cost factors affecting the Return On Investment (ROI).

- **The application should be developed and deployed quickly.** It cannot be over-emphasized that time-to-market is a crucial factor for success in today’s business world. If development is too slow the competitive edge may be lost or the technology may become obsolete by the time the application is delivered.

- **The application should be easy to modify and maintain.** For most applications the cost of modification after development is greater than the cost of initial development. Inflexibility in architecture and design may cost the enterprise a lot of time, resource, money, and frustration.
The application should have long useful life. An application may become obsolete as its technology platform ages and its business rules change. The easier it is to adapt the application to changing technology and business needs, the longer is the application life and thus the longer the amortization period.

In today’s application development arena, there are many development tools available and each has its own merits. In general, we can distinguish them as “conventional” and “rules-based” development tools. Conventional development tools use 3rd GL and 4th GL development based on thin-client or thick-client models. Their rules-based counterparts can be one of four distinct application architectures and methodologies:

- Service-oriented
- Data-oriented
- Framework-oriented
- Code generation

Products are available today to support the first three categories. Services are available for the last category, to generate code from meta-definitions, but tools to support this approach are only beginning to appear. This document includes Code Generation as an alternative rule based application development approach. The merits of having development tools to support it are discussed.

In sum, I believe:

- Information technology tools can significantly enhance the value added to organizations
- Different companies derive different value from different approaches to software application development, and therefore need a customized set of technology tools
- Organizations need an objective assessment of their technology needs and potential solutions that is not tied to any single product, package, or tool
Analysis

Just as there is no “one-size-fits-all” solution, there is, for most companies, no single technology or tool that would support all application development needs. I have identified eight characteristics for each of the five categories of development tools. By comparing these characteristics we can tie them back to the three major cost factors affecting the ROI of software development.

The Development Tools

Conventional Programming Language.

There are a variety of powerful 3rd Generation Language (3GL) and 4th Generation Language (4GL) tools to choose from. They provide maximum capability but generally require a lot of programming effort. Application architecture is often unique and designed from scratch. Examples are Java, C#, VB, Delphi.

Service-oriented Rule Engine

This type of rule engine captures “structured” English business rules in its proprietary data model in an isolated environment. Applications call the rules service for execution of rules and control is returned back to the calling routine for data manipulation. Examples are Ilog, Corticon, HNCBlaze, Haley, Logist.

Data-oriented Rule Engine

This type of rule engine captures “pseudo” English business rules and integrates them tightly with the data model of the application. Rules are often executed on “Create, Read, Update, or Delete (CRUD)” operations before persistent data is stored. Examples are Versata and Ness (formerly Usoft).

Framework-oriented Rule Engine

This type of rule engine essentially provides a framework and a proprietary language so that business rules can be specified and executed in a controlled
environment. Often, a decision-table mechanism is employed. Examples are AION and Sapiens

**Code Generation from Meta-data**

All the design components including the business rules of the application are captured as portable meta-data. A code generation engine takes the portable meta-data and generates the necessary code in a selected conventional development language. So far only services are available in this category. Products to support the approach are only beginning to appear.

**The Characteristics**

Contrasting some of the major characteristics of each development tool should provide a clear understanding of how each individual development tool fits into individual situations.

**Business Rule Representation.**

When a conventional programming language is chosen for the development of an enterprise application, the business rules are often embedded in the programming code. Programmers are often required to put the business rules in a distinct “logic” layer, but this discipline may be difficult to follow and there are no automated mechanisms to enforce such programming practice.

A rule engine requires that all the business rules be entered into the engine in a proprietary format. This ensures all the business rules are captured in one place for easy documentation and change.

Similarly, code generation from meta-data requires that the rules be captured explicitly, and separately from other specification components. One point worth mentioning is that business rules captured in meta-data are separate from the code generation engine. The same meta-data can be provided to different vendors for input to code generation on different engines.

**Natural Language Level**

Currently there is no general-purpose translation engine available to convert business rules written in structured English into the conventional
programming language of your choice. For development with a conventional 3GL or 4GL, you will need to:

- Either, develop your own rule translator or interpreter
- Or, hand code all the business rules in the programming language (custom coding).

In contrast, a service-oriented rule engine often provides a very high level structured English syntax for the business rules. A user-friendly screen would enable business rules to be constructed graphically by pointing and clicking. Some even enable business rules to be written in other natural languages, such as German, French or Hebrew.

Similarly, a user-friendly point and click interface would be available in most of the data-oriented and framework-oriented rule engines. The major difference is that the representation would be closer to programming language than to English.

Business rules written in SQL-style syntax and decision tables are also found in some data-oriented rule engines and framework-oriented rule engines.

Currently, code generation services usually require business rules in SQL-like form.

In the future, generic structured English rules such as those being promoted by Business Rule Solutions could be translated to SQL. ESI from Israel has also expressed interest in sharing its structured English rule syntaxes with the development community.

**Business Rule Type Specialization**

The technologies under consideration have different strengths in the way they support different types of rule. The diagram below provides a simple representation of the knowledge components in a typical application.
The rules usually considered to be “business rules” can be classified into four mutually exclusive categories:

- **Inference Rules.** An Inference rule establishes a new fact upon finding one or more conditions to be true. Inference rule may involve only one table, but usually involve more than one table. An example is “a preferred customer has no past due outstanding invoices”

- **Computation Rules.** Perhaps Computation rules are the easiest to understand. A computation rule can involve data from a single row in a single table, or multiple rows from multiple tables. An example is “the total invoice is the sum of all items in an invoice”.
- **Constraint Rules.** These often define validation or acceptance criteria. A simple constraint rule may use just a single database column; for example “a customer must not have more than 10 open orders at one time”. Another example would be referential integrity rules, which define constraints between two tables. For example “An customer must not be deleted if there are any accounts still existing for that customer”.

- **Action Enabler Rules.** An Action Enabler is a complete statement that triggers a business event, message, etc. upon evaluating one or more conditions to be true. An example is “start the internal audit process if a customer defaults more than $5M”.

Most rule-based products available today focus mainly on servicing business rules. Presentation rules and workflow rules are also significant when it comes to application development.

- **Presentation Rules** enable certain fields to be visible, read-only or for data entry, based on the input data. An example is “hide all commercial columns when user applies for a personal loan”.

- **Workflow Rules** guide the user through a process without the user needing to be aware of the whole process ahead of time. This technology can also be used for training new workers. An example is “client information and consent must be obtained prior to getting credit report from credit bureau”.

Conventional programming languages do not provide a prescribed form for any of the above types of business rule.

Service-oriented rule engines often specialize in inference type applications. Data-oriented rule engines can generally handle computation and constraint rules very well.

Framework-oriented rule engine and code generation from meta-data have no specialization at all. They basically provide a framework for all types of rules to be captured and fired as instructed.

**Architecture.**
A conventional programming language does not provide any built-in architecture. Each individual development may be handled as a unique situation within its own enterprise architecture.

A rule engine, whether it is service-oriented, data-oriented or framework-oriented, will assume a predefined, product-driven architecture.

One of the advantages code generation is designed to offer is architecture independence. While architecture is still largely product driven, different vendors providing different code generators should be able to offer a variety of architecture for the same meta-data.

**Data Model**

A service-oriented rule engine generally requires a separate data model to be constructed, to be used inside its enclosed environment. This business
rule data model may be a near-duplicate of the application data model de-normalized for support of natural language processing.

The other types of development tool usually have a data model to support the entire application including business rules.

**Design and Coding Structure**

Most developers when using conventional programming languages would embed all design information, including business rules, within the program code.

A service-oriented rule engine offers a separation of business rules from the program code.

Data-oriented and framework-oriented rule engines go one step further to offer a separation of user-interface and workflow design from the program code but they are usually cohesive - that is, one cannot take the design and use it elsewhere.

The main design goal of code generation from meta-data is not only to separate all user interface, workflow, presentation rule, and business rule from program code, but also make them portable so other code generators can use them.

**User Interface & Workflow**

A conventional programming language can be used to develop user interfaces and workflow control. It can also augment a service-oriented rule engine, which usually would not provide any specific user interface and workflow tools.

Data-oriented rule engines often provide user interface and workflow tools, fully integrated with the business rule engine, and may enforce their use.

A framework-oriented rule engine may also provide user interface and workflow tools but may permit (or encourage) users to use other user interface and workflow tools to interface with the business rule engine.

Code generation from meta-data would enforce the user interface and workflow tools that are supported.
Documentation.

When a conventional programming language is used, design documentation is separate from the code. It might not be synchronized with application changes during development. Sometimes documentation is written after the application is developed, so that it does reflect the system as delivered, but it is common for such documentation not to be kept up to date as the application is developed further.

With a rules engine, the application, as specified in the development tool, is documented at a higher level (i.e. in a form more understandable in business terms) than 3GL or 4GL code.

Service-oriented rule engines rely on other user interface and workflow tools, so documentation is generally synchronized only to the business rule level.

The other types of rule engine, and code generation from meta-data, require the design to be documented before or as the application is developed, and documentation can be synchronized with all application changes.

Summary Table of Characteristics

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Business Rule Representation</td>
<td>Custom coding</td>
<td>Proprietary</td>
<td>Proprietary</td>
<td>Proprietary</td>
<td>SQL-based</td>
</tr>
<tr>
<td>Natural Language Level</td>
<td>Custom coding</td>
<td>Business rule in structured English</td>
<td>Business rule in pseudo English, SQL, etc.</td>
<td>Business rule in pseudo English, decision table, etc.</td>
<td>Business rule in SQL, etc.</td>
</tr>
<tr>
<td>Business Rule Type Specialization</td>
<td>N/A</td>
<td>Inference</td>
<td>Computation and Constraints</td>
<td>No Specialization</td>
<td>No Specialization</td>
</tr>
<tr>
<td>Architecture</td>
<td>N/A</td>
<td>Product dependent</td>
<td>Product dependent</td>
<td>Product dependent</td>
<td>Provided</td>
</tr>
<tr>
<td>Data Model</td>
<td>Integrated</td>
<td>Duplicate</td>
<td>Integrated</td>
<td>Integrated</td>
<td>Integrated</td>
</tr>
<tr>
<td>Design and Coding Structure</td>
<td>Design embedded in code</td>
<td>Cohesive</td>
<td>Cohesive</td>
<td>Cohesive</td>
<td>Design separated from code</td>
</tr>
</tbody>
</table>
The Return on Investment Factors

Each of the above characteristics would affect at least one of: the speed of development and deployment, the ease of modification and maintenance, the life span of the software application.

**Speed of Development and Deployment.**

"A magic wand that can develop any enterprise application your heart desires in a split second would be worth a lot to the business community"

It goes without saying that speed of development is a crucial factor in delivering a positive ROI. Development cost is a factor (when usually less resources being consumed in a fast delivery), but more important is the impact on the business of having the application sooner, to enhance the competitive position and, in turn, bring in more profit.

Business rules represented in structured English would encourage direct user participation in development. Instead of the users educating and communicating to programmers all the business rule requirements, the users can specify the business rules themselves. Representing business rules in structured or pseudo English, or simply providing a framework for business rules entry would cut down development time, eliminates communication errors, reduces testing, and speeds up development.

When the architecture is provided, it is usually well tested. This can save a lot of time in development and deployment.

One factor to consider with service-oriented tools is that developing applications with two similar and dependent data models can be a challenge. Not only does it slow down the development process, but modifications
during the development phase may require synchronization of the two data models. This may also add complexity to the deployment process.

It is often difficult to transfer business knowledge to technical professionals and technical knowledge to business people. A design interface that requires little coding or no coding is enables business analysts to design and construct the application with little help from the technical people. Therefore significant communication errors and training can be saved.

Similarly, the more user interface and workflow are automated, the faster the development.

It is important to have documentation on applications. Almost all applications developed using conventional programming language and methodology have documentation that lags behind development and is not synchronized with changes. Systems people seldom rely on them for development or making changes to the applications. In the case of rule-based development tools, the design is the documentation in most cases, and this speeds up the development process.

The impacts of the product characteristics on Speed of Development are summarized in the table below.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Natural Language Level</td>
<td>Slow</td>
<td>Fast</td>
<td>Fast</td>
<td>Fast</td>
<td>Fast</td>
</tr>
<tr>
<td>Architecture</td>
<td>Slow</td>
<td>Fast</td>
<td>Fast</td>
<td>Fast</td>
<td>Fast</td>
</tr>
<tr>
<td>Data Model</td>
<td>Fast</td>
<td>Slow</td>
<td>Fast</td>
<td>Fast</td>
<td>Fast</td>
</tr>
<tr>
<td>Design and Coding Structure</td>
<td>Slow</td>
<td>Fast</td>
<td>Fast</td>
<td>Fast</td>
<td>Fast</td>
</tr>
<tr>
<td>User Interface &amp; Workflow</td>
<td>Slow</td>
<td>Slow</td>
<td>Fast</td>
<td>Fast</td>
<td>Fast</td>
</tr>
<tr>
<td>Documentation</td>
<td>Slow</td>
<td>Fast</td>
<td>Fast</td>
<td>Fast</td>
<td>Fast</td>
</tr>
</tbody>
</table>

**Ease of Modification and Maintenance**

"Life is about continuous improvement – and so is business"
Business changes rapidly. Some organizations develop applications with a Waterfall approach, which requires all design specifications to be frozen prior to coding and testing. This does not work in most situations because the business is evolving while the system is being developed. By the time the application is written the specification has already changed and the application may be unusable.

Organizations need an approach in which:

- It is easy to recognize the impact of making changes
- It is easy and fast to implement the changes

Such an approach would not only earn and save the organization a lot of time, resource and money, it would prolong the product life of the application.

Custom applications built with conventional 3GL/4GL can be costly to modify, maintain, and procuring well-trained professionals may be difficult. Business rules are often embedded in the code. Changing something may have side effects on other parts of the application, perhaps causing malfunction. Having design separated from code has tremendous advantage when it comes to modifying the applications.

Representing business rules in structured or pseudo English, or simply providing a framework for business rules entry enables direct user participation in defining modifications of the applications with ease. Using the rules directly in the application supports fast implementation of the changes. Having specialization or structured business rules means that users may easily identify the right place to make modifications, thus making modifications and maintenance much easier.

A service-oriented rule engine does not provide user interface or workflow tools, but the design is the documentation itself when it comes to business rules handling. Having a separate data model for rule representation requires extra work in modification and maintenance, but the rules engine can be used in different technical environments for user interface and workflow. With data-oriented and framework-oriented rules engines, and code generation, having user interface and workflow automated does enhance the ease of modification and maintenance, although it restricts the architectural choices.
The rule-based development tools have documentation synchronized with the design automatically, thus making modification and maintenance easy.

The impacts of the product characteristics on Ease of Modification and Maintenance are summarized in the table below.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Business Rule Representation</td>
<td>Easy</td>
<td>Not easy</td>
<td>Not easy</td>
<td>Not easy</td>
<td>Easy</td>
</tr>
<tr>
<td>Natural Language Level</td>
<td>Not easy</td>
<td>Easy</td>
<td>Easy</td>
<td>Easy</td>
<td>Easy</td>
</tr>
<tr>
<td>Business Rule Type Specialization</td>
<td>Not easy</td>
<td>Easy</td>
<td>Easy</td>
<td>Easy</td>
<td>Easy</td>
</tr>
<tr>
<td>Data Model</td>
<td>Easy</td>
<td>Not easy</td>
<td>Easy</td>
<td>Easy</td>
<td>Easy</td>
</tr>
<tr>
<td>Design and Coding Structure</td>
<td>Not easy</td>
<td>Easy</td>
<td>Easy</td>
<td>Easy</td>
<td>Easy</td>
</tr>
<tr>
<td>User Interface &amp; Workflow</td>
<td>Not easy</td>
<td>Not easy</td>
<td>Easy</td>
<td>Easy</td>
<td>Easy</td>
</tr>
<tr>
<td>Documentation</td>
<td>Not easy</td>
<td>Easy</td>
<td>Easy</td>
<td>Easy</td>
<td>Easy</td>
</tr>
</tbody>
</table>

**Useful Life of the Application**

"Useful life of an application can be significantly enhanced if design is well documented independent of the underlying technology"

The thought of having major knowledge of a multi-billion dollar company embedded in obsolete 3GL/4GL code is scary. When conventional programming languages are used, the useful life of the application is not necessarily dependent on the technology. It is dependent on how good the development standards are, and how well they are enforced.

Business rules state what the business wants to do, independently of the technology used to implement them. If they are captured, even in a proprietary syntax, in a near-natural language form that is understandable by business people, knowledge of what the applications do is easily passed on from person to person. This can help in prolonging the life of the applications.
If the business rules are stated in a form that can be directly processed by a rules engine, changes can be directly implemented. If the business rules technology changes, the rules can be migrated to the new technology, although it might require changing the syntax of the rule statements.

Rules for user interface and workflow are generally more product-specific, and it may be more difficult to reuse then on different technology platforms.

Code generation from meta-data compares well with data-oriented rule engine and framework-oriented rule engine in many aspects. It may be better in some cases because of its openness. Where code generation may make a difference is the fact that the design and business rules are separated from coding, providing the opportunity for different vendors to generate code for various programming languages and architectures. This can prolong the life of the applications as technology changes.

Documentation that is synchronized with the state of the application enhances the life of the application.

The impacts of the product characteristics on Life of the Application are summarized in the table below.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Business Rule Representation</td>
<td>Typical</td>
<td>Long</td>
<td>Long</td>
<td>Long</td>
<td>Long</td>
</tr>
<tr>
<td>Natural Language Level</td>
<td>Typical</td>
<td>Long</td>
<td>Long</td>
<td>Long</td>
<td>Long</td>
</tr>
<tr>
<td>Architecture</td>
<td>Typical</td>
<td>Typical</td>
<td>Typical</td>
<td>Typical</td>
<td>Long.</td>
</tr>
<tr>
<td>Design and Coding Structure</td>
<td>Typical</td>
<td>Typical</td>
<td>Typical</td>
<td>Typical</td>
<td>Long.</td>
</tr>
</tbody>
</table>

**Conclusion**

In conclusion, when considering software development tools to develop your next killer application for your enterprise wide deployment, if the existing
tools do not satisfy all your criteria, consider code generation technology as an alternative.

Some day the industry may agree on a standard way of representing all the application design components as meta-data and various vendors would develop various code generators to write programs on behalf of programmers. Application development would become more user oriented. With just the press of a button organizations could switch look & feel, architecture, business language, culture, programming language, operating systems, or even platforms without reprogramming from scratch. But until then, organizations will need the kind of evaluation framework I have outlined in this paper.

**about ... NELSON LIN**

Nelson Lin has been developing enterprise software solutions for more than fifteen years and was winner/finalist of three awards from Microsoft, including the most prestigious "Best ISV application world-wide" award. He graduated from the University of Western Ontario with both an MBA and B.Sc. Computer Science degrees. He is also one of the first few professionals who had achieved both Microsoft Certified Systems Engineer and Microsoft Certified Solution Developer qualifications. He is well trained in Microsoft Solution Framework and had been invited to lecture project management professionals.

In the mid-90’s, he was the pioneer in combining rule-based credit scoring and message queuing with on-line credit bureau integration to provide on-line end-to-end fully automated consumer term lending systems. He has design and programming experience in all three computing paradigms – Host Based computing e.g COBOL/RPG, Client-Server computing e.g. PowerBuilder/VB, Web based computing e.g. Java/C# .

Nelson can be contacted at nelson.lin@robocoder.com