Currently there is a lot of excitement and competition in the Business Process Automation space, however, despite the plethora of BPA platforms available, many such projects end up over budget, provide very little business flexibility when changes are necessary and fail to reach performance, transparency, and reliability goals.

The purpose of this paper is to describe an alternative framework for the agile development of reliable, disposable interfaces and services, specifically aimed at providing maximum business flexibility and interface quality while reducing implementation and long-term operational costs.

Traditional development approaches (particularly in fixed price, outsourced projects) are normally based on a contractually-fixed set of requirements and relatively heavy change-control processes. Movements in the IT industry to formulate new development approaches to match more realistic business scenarios where requirements are continually refined and updated in the course of a project, have resulted in agile development methodologies, the most prominent of which is Extreme Programming.

Agile methodologies aim to remove bureaucratic barriers to producing tangible results without sacrificing quality. Agile development methodologies provide a disciplined approach to IT development intended to produce usable results faster and with higher quality than traditional approaches. Fundamentally, agile methodologies try to remove bureaucratic barriers to producing tangible results without sacrificing quality.

Fast and flexible development of disposable interfaces is particularly relevant in service-oriented architectures; services should reflect the current needs of the business processes they are servicing; changing business requirements often require corresponding changes to technical processes. For example, this concept is reflected in Jini, a Java-based SOA framework, that implements utility helper classes to define temporary software dependencies.

Open-source Jini and the majority of proprietary BPA solutions are based on Java. One of Java’s strengths lies in allowing the definition of precise internal and external interfaces. However, this can also be a drawback, as a significant amount of design and implementation time is spent on doing just that. For exam-
ple, if every modification to internal interface definitions requires a corresponding modification to all of the dependent code as well, this can add up to a significant amount of time and effort and can be considered a bureaucratic obstacle to producing working results when viewed in the context of rapid prototyping or developing disposable internal interfaces.

While the requirement of defining detailed interfaces is meant to ensure correctness and to allow larger project teams to collaborate on building a much larger system, larger project teams require exponentially greater coordination effort as the number of developers grows. Eventually, the flexibility to react to changing business requirements is lost.

Furthermore, many software projects are planned with a life span of three to five years in mind to amortize the cost of development. This is not realistic for most service-oriented environments or for business process integration projects. More flexibility is needed.

Instead of using a language and application framework designed to enable large projects teams to collaborate, the language and application framework should empower smaller teams to do more in less time. To better achieve agile development goals, the language and the application framework should do everything possible to remove bureaucratic obstacles to producing tangible results and should instead facilitate rapid prototyping and the generation and controlled decommissioning of disposable interfaces and services.

As Ruby on Rails provides an agile development framework for web applications, this paper aims to describe a similar framework for the development of integration and business process automation projects. As Ruby on Rails leverages the expressive power of Ruby, an object-oriented, dynamically-typed interpreted language, within a web application framework, allowing for very fast implementation of powerful web applications and open-ended customization of the behavior of all elements of the system, this paper proposes the integration of a dynamically-typed interpreted language within a rigid framework to facilitate the rapid creation of reliable interfaces and services for business process automation in SOA and messaging environments.
This framework must do more than satisfying basic technical needs such as database connectivity, transactional integrity, data transformations, protocol conversions, and messaging integration. These subjects and more must be covered by any practical BPA solution. In addition to this, the framework must provide maximum flexibility to changing requirements, robust error-handling to ensure reliability of the solutions delivered on the platform, operational and configuration transparency to enable quick identification of the flow of logic in interfaces and errors in the source code, and must meet performance needs without requiring exorbitant amounts of hardware to do so.

If these features are not guaranteed by the framework, then in typical IT projects when budget and/or time are tight due to unclear or delayed business or technical requirements, under-estimating the effort required, or other problems common in real business environments, these critical features are often postponed to a post-launch phase, and, in the worst cases, never delivered at all.

Furthermore, these features are even more critical in an agile development process, because, as long as they are satisfied by the framework, they do not take the form of bureaucratic obstacles to generating tangible results and can never be postponed to a later delivery phase. The presence of these features in the framework keeps programmers focused on generating only the code that must be customized for the particular job at hand.

From the point of view of business customers, the framework must provide the following high-level features:

- Flexibility: quick response to changing business requirements: short development times while maintaining high quality
- Reliability: unavoidable technical errors such as application, database, and network outages are handled gracefully and are recovered automatically when possible; the possibility for error recoverability must be guaranteed by the platform by tracing task statuses and providing a mechanism for correcting errors and restarting the suspended tasks
- Transparency: the status, configuration, and even the source code of automation tasks is readily visible and easily accessed through
network-available APIs to make changes easier and to facilitate a short feedback loop for change control.

- **Performance and Scalability:** must not require exorbitant amounts of hardware to meet performance expectations; solution should scale on today’s and tomorrow’s multi-core and multi-processor hardware.

These features can be realized by executing user-defined code written in a dynamically-typed programming language within a rigid framework that guarantees reliability and transparency. Such a system would make a powerful and flexible solution for agile development in SOA and message-based environments.

In such a system, both integration tasks and services would be made up of objects with code attributes written in this language, executed in carefully controlled and limited contexts the integration framework that provides reliability and transparency.

In this way, the inherent flexibility and expressive power of the dynamically-typed language are embedded in a framework that covers all the bureaucratic elements of interfacing such as: loading and unloading services, tracking statuses and data, facilitating synchronous and asynchronous messaging, providing shared data and messaging resources to the system, and more. This results in a system that provides the quick reaction and development times to stay on top of changing business and technical requirements while providing maximum reliability and fault-tolerance of the automation tasks built upon it.

The structure of the framework, when coupled with executing the dynamically-typed code in limited and controlled contexts as attributes of integration objects, will tame the expressive power of the dynamically-typed language and focus it narrowly on the integration task at hand, allowing rapid development of powerful “reliable-by-design” interfaces and equally powerful disposable services.

Therefore, the best of both worlds could be achieved: rapid development times leading to increased business flexibility and reduced development costs, while the inherent reliability of the solution leads to increased process quality and customer satisfaction, also while reducing long-term operational expenses.
To achieve this goal, the dynamically-typed language would have to be designed to support a rather unique set of features. Furthermore, the framework would have to be tightly integrated with the language, and would have to have the correct design to deliver on promises of true automated error handling in order to provide the high levels of reliability to allow large volumes of complex tasks to be managed by small operational teams.

The high-level, dynamically-typed language would have to have the following features:

- **Powerful Integration Capabilities**: Database access with transaction management, character encoding, XML and common protocol and web-service protocol support, publish-subscribe and point-to-point messaging capabilities, and more.

- **Logic Embedding/Encapsulation Support**: Visibility of data in the embedded code must be controllable; the framework must restrict access to features of the language that could compromise system integrity. Embedded code must be containable in discrete objects that can be created and destroyed on demand, and code objects must not interfere with other objects unless by design, in a controlled and safe manner.

- **Resource Control/Tracking**: Allows the framework to catch and log programming errors and automatically free resources; to provide a safe execution platform for the dynamically-typed code.

- **Clean Threading Model, SMP Scalability**: The solution will require very fine-grained threading to be scalable on current and future multi-core and multi-processor systems.

- **Exception Handling**: Allows high-level, complex actions to support robust user-defined error handling.

This feature set should allow an automation framework to be developed that provides a safe platform for the execution of dynamically-typed code in an enterprise integration environment by focusing the expressive power of the language on integration tasks.

The framework must track the execution of each element in an interface’s flow as executed from dynamic user-defined...
code and react when errors occur to enable error conditions to be automatically recovered or be flagged for manual intervention. Additionally, the framework should provide a flexible automated solution to complex error conditions, such as lost response messages to successfully-processed outgoing messages with a non-repeatable action (for an example, see the callout on the left). Furthermore, the framework should provide a structure for handling difficult tasks such as asynchronous messaging/event processing in a safe way where the system itself performs most of the hard work and the programmers simply define code attributes in pre-defined objects.

To provide for maximum efficiency and to allow for the managed lifecycle of disposable workflows and services, user code must be able to be dynamically loaded (from a database for example) and deleted on demand, meaning that the system must support online logic upgrades while guaranteeing the consistency of the platform and not disturbing unrelated objects.

Such an implementation is less open-ended than an application server or even an agile platform like Rails. However, the structure of the system serves to focus the code purely on the tasks that must be customized, and takes programmers further toward the goals of agile development than less open-ended systems in the sense that more overhead/bureaucratic work is removed from the programming domain and placed in the system domain. In this way, this solution could also be considered an intermediate step between application scripting (for example, the SingleView billing platform, which uses its own interpreted language to customize the behavior of the system in controlled contexts) and open-ended agile platforms such as Rails.

The interfacing and service framework described here is not useful without workflow and service definitions and associated user code attributes, and therefore is a step beyond application scripting, where the application is normally usable without user-code customizations, however, it gives more structure and focuses the user code more narrowly on integration tasks than Rails. Rails and Zope, another open-source application server based on Python, another interpreted object-oriented scripting language, must be more open-ended due to the nature of an application server and web application development in general.
However, the interfacing and service framework described here, coupled with a web-service-based API and a client GUI giving complete access to the status and configuration of the system, could potentially give businesses the power and flexibility to realize their automation projects with significantly reduced one-time and long-term costs and with much improved flexibility versus traditional solutions, even if agile development processes are not employed.

One concrete example of a system taking such an approach is the Qorus Integration Engine™ by Qore Technologies. This innovative system embeds the Qore programming language, which is an open-source, object-oriented, dynamically-typed programming language that was specifically designed for the realization of an optimized, fault-tolerant, agile integration platform for service and message-oriented environments as described in this paper.

Qorus allows for total encapsulation of Qore-language code as attributes of discrete objects (for example, the main logic in a step in a workflow); the code is loaded on demand from the system database schema and only executed in a controlled manner in the appropriate context and only given access to the relevant data being processed.

The lifecycle of all user objects in the system is managed in a way that permits live commissioning, upgrades, and decommissioning of interfaces and services. Furthermore, the server process is highly threaded and highly scalable on multi-core and multi-processor systems, in a large part due to the unique internal architecture of the Qore programming language. Additionally, XML-RPC and JSON-RPC, both lightweight web-service protocols, are used to export all system functionality to network clients making integration easy with standard technologies and in existing enterprise environments.

Complex multithreaded workflow and service definition is very easy; the Qorus system takes care of all the hard parts of managing threads, statuses, data, asynchronous messaging/events, error recovery, and more, freeing programmers from the burden of managing these tasks, and focusing their efforts only on the code that must be customized to realize the interface or service being developed.
Qorus supports automatic error recovery even of complex conditions such as lost response messages to non-repeatable actions (as defined earlier in this paper) by defining the appropriate user-code attribute of a workflow step. Due to such features, Qorus supports the Guaranteed Recoverability™ of automation tasks, meaning that, as long as simple design criteria are met and network transports and end applications are eventually available, workflows are guaranteed to be able to run to completion.

Qorus provides the rigid structure for providing lightweight, disposable services and powerful, fault-tolerant workflow execution, while embedding the Qore programming language in discrete objects as user-code attributes of service and automation objects to support agile development in business process automation tasks by providing an unprecedented level of flexibility to react to changing business requirements.

This approach and the Qorus Integration Engine itself has proved to drastically lower development and operational costs while improving reliability and providing very high business flexibility in real business environments. For example, Hutchison 3G Austria successfully re-worked their entire Business Support Systems architecture, using Qorus as the main integration engine for all business-critical processes requiring IT automation. A small team of six developers with no prior integration experience was able to provide flexible fault-tolerant integration solutions for this major Austrian telecommunications carrier in less than six months from design to live production use.

For more information on this approach or for information about the Qorus Integration Engine, contact Qore Technologies at info@qoretechnologies.com, +420 222 521 165, Prague, Czech Republic.