BPTrends ARTICLE

Enterprise Rule Enactment Service Patterns

January 7, 2014
Vitaly Khusidman

Abstract

This Article describes two enterprise architecture patterns for Rule Enactment Service that enables externalization of business rules. The first pattern promotes loose coupling by ensuring independence of the service consumers from service provider implementation. The second pattern enables reuse of service provider implementation across decision types. These patterns can be used in any enterprise architecture with an objective to improve agility and reusability of the solutions that externalizing decision making business logic into separate components exposing service interfaces. The main business benefits of using these patterns include shorter time to market, lower initial cost of the subsequent decision type implementations and lower maintenance cost.

Background

Evolving enterprise wide solutions vs. building individual solutions is analogous to managing a program vs. a project. While a project is generally concerned with satisfying its requirements and being on time and on budget a program objective is to improve organizational KPIs, i.e. increase revenue, decrease cost, improve customer satisfaction, etc. According to Project Management Institute [1], "… A Program is a group of related projects managed in a coordinated manner to obtain benefits and control NOT available from managing them individually…"

Similarly the enterprise wide solution aims at the improving organizational qualities and, therefore, deals with aspects NOT in scope of the constituent individual solutions. These enterprise aspects are widely covered in the industry publications. This Article focuses on two important enterprise aspects, namely *loose coupling* and *reuse*, as they are applicable to building enterprise rule enactment services. Loose coupling promotes separation of concerns and eliminates dependencies between components, thus, allowing replacement of a given component without affecting other ones that collaborate with it. Loose coupling is usually achieved with the help of standard interfaces between collaborating components.

Reuse is another fundamental architecture principle exploiting economy of scale in the context of the enterprise wide solution. Reuse can be achieved with different degrees of coupling and binding methods between components as shown by Vitaly Khusidman and Dave Bridgeland [2]. Service Oriented Architecture (SOA) is currently the most common way to achieve reuse at the enterprise scale; however, even greater level of reuse can be achieved by using parameterized service interface.

Rule Enactment Services are responsible for making decisions based on business rules. Enactment service consumers include delivery channel BPM and non-BPM applications. They also include other services encapsulating business logic or any other enterprise architecture components making decision based on the rules which need to be externalized.

The next sections describe two enterprise architecture patterns for Rule Enactment Services using architecture pattern template defined by the TOGAF [3]. The first

pattern promotes loose coupling by ensuring independence of the service consumers from service provider implementation. The second pattern enables reuse of service provider implementation (e.g. code/script) across various use cases.

Pattern 1

Name
Agile Enterprise Rule Enactment Service

Problem
The agility of the enterprise solution is impacted by the need to modify enactment service consuming applications every time when the service provider implementation changes. This is a result of tight coupling between the rule enactment service consumer and the service provider implementation.
The objective of this pattern is to increase solution agility by applying loose coupling architecture principle. This will reduce time to market, preserve organizational investment and reduce maintenance cost.

Context
In a typical enterprise scenario there are multiple projects scheduled over a multi-year timeline that are rolling out applications leveraging given category of rules. Some of these applications require rules that are not complex and/or not expected to change over the lifetime of the application. Other applications use rules of moderate complexity that require infrequent (e.g. once a year) changes. Yet another category of applications requires complex rules which undergo frequent (e.g. once a month) changes. The application requirements to rule agility may change over time as the application and its users gain more experience and responsibilities or in case the application environment changes.
Additionally, the enterprise can have a project independent schedule for rolling out (and possibly replacing) rule-based technologies. Therefore, projects have to leverage rule-base technology at the level of maturity that is available according to the enterprise technology roll out schedule.
All the above mentioned factors contribute to the decision on selection of the rule enactment service provider implementation approach in the context of a given project. There are three distinct approaches a project can select from: Hardcoded, File/Database-based and Rule Engine-based (see Figure 1 below).

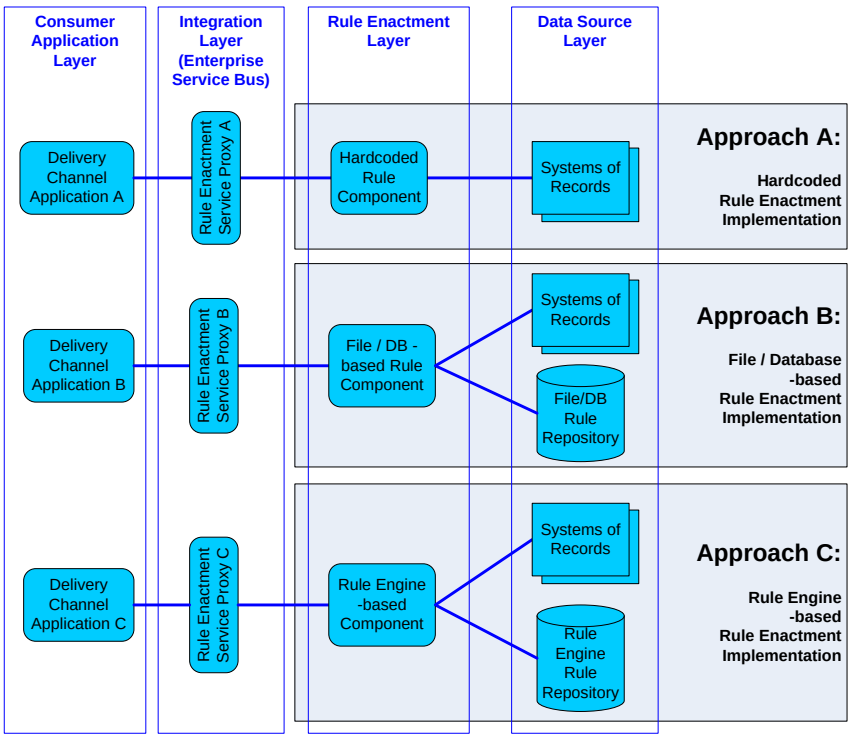

Figure 1 The existing context with Rule Enactment Service implementation approaches

In the above diagram the alternative approaches are indicated by horizontal blocks marked Approach A, B and C. The participants of service consumer-provider collaboration are allocated to the following four layers:

- Consumer Application Layer – service consuming applications belong to this layer
- Integration Layer (a.k.a. Enterprise Service Bus) – this layer houses service proxies responsible for exposing service interfaces to service consumers, as well as for enforcing security and performing service call routing, message transformation and protocol mediation
- Rule Enactment Layer – this layer houses components that execute business rules
- Data Source Layer – this layer houses repositories of rules and Systems of Record (SoR) storing an operational copy of the data the business rule are operating upon

The three alternative approaches for rule enactment service implementation are described as following:

Approach A: Rule enactment service is implemented as a hardcoded component using a programming language (e.g. Java, C#, etc.). This approach is the least agile and often the least costly in initial implementation. However, if rules need to change the code must be modified by going through the entire software development life cycle; therefore, the maintenance cost for this approach may prove to be significant.

Approach B: Rule enactment service is implemented as a rule interpreter (very simple custom version of a rule engine) and rules themselves captured using a custom notation and stored in a repository (e.g. file or database). This approach is more agile then Approach A because changing rules does not necessarily mean rule interpreter code change – code has to be modified only in cases when the change is

beyond variability provisions set by design. The initial implementation cost varies from slightly higher to significantly higher than for Approach A depending on the rule interpreter complexity. The maintenance cost is lower than for Approach A because most of the anticipated changes can be accomplished by merely modifying rules in the repository without a need to modify the rule interpreter code.

Approach C: Rule enactment service is implemented by leveraging a Commercial Off the Shelf Software (COTS) rule engine executing rules defined in respective rule definition language and stored in the rule repository. This is the most agile approach with the implementation cost largely dependent on the rule engine license and infrastructure cost. This approach has the lowest maintenance cost in the case of frequently changing rules.

Forces

One dimension of the problem is related to selection of the rule enactment service implementation approach. This selection is often dictated not by functional requirements but by availability of technology or project time/budget constraints. However, a tactical selection of the implementation shall not diminish the ability of the application to transition to a more appropriate approach once it becomes needed and/or available. Thus, the application will be able to leverage a different but functionally similar (i.e. executing the same business logic) implementation in the future without any modifications to the application itself.

The main forces for introducing this pattern are: extensibility, evolvability, maintainability and reusability of the service interface.

Solution

The pattern enables the organization to change the rule enactment service implementation without the need to modify the consuming applications.
This can be achieved with the help of a stable rule enactment service interface that supports a family of consuming applications requiring the same category of business rules but potentially leveraging different enactment service implementation approaches. This interface must have an enterprise scope and be resilient to changes within a foreseeable future.

The solution replaces implementation specific Rule Enactment Service proxies with the Rule Enactment Enterprise Service Proxy exposing the enterprise interface that a) supports all consuming applications (within the enterprise) requiring given category of business rules, and, b) supports service implementations following all alternative approaches.

The Rule Enactment Enterprise Service Proxy interface consists of a single operation *executeRule()* with request and response composed from data elements supporting all anticipated use cases involving all relevant consuming applications and service implementations. This interface is defined for specific decision type representing business domain, decision category and optional cascading decision subcategories, e.g. ‘retail banking’ business domain, ‘client eligibility’ decision category and ‘mortgage eligibility’ decision subcategory.

Resulting Context

The resulting context with the Rule Enactment Enterprise Service Proxy is shown in the Figure 2 below. This proxy exposes a single enterprise interface supporting all use cases involving for the given decision type.

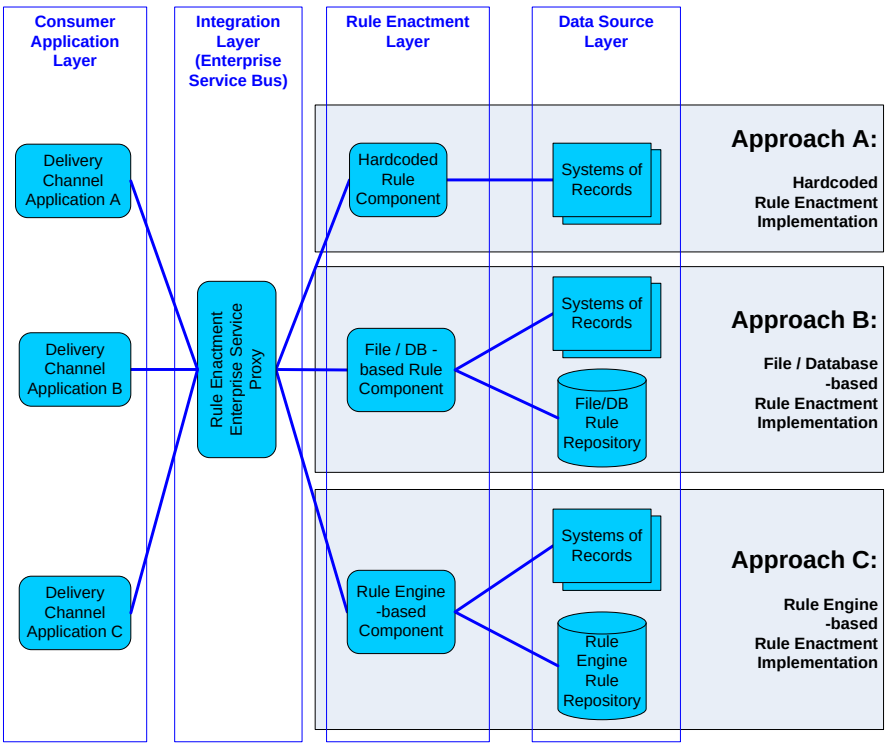

Figure 2 Resulting context with Enactment Rule Service enterprise interface

Example

A common example of the Rule Enactment Enterprise Service is Client Eligibility Service which can be found in numerous business domains were organizations offering products and/or services to the clients. Organizations may have eligibility rules that defining whether a client is eligible for a given product and or its feature. The eligibility rules can be defined by the organization or by a regulatory authority. The request and response messages of the *executeRule()* operation of the *Client Eligibility* service for a decision type ('retail banking', 'client eligibility', 'credit card eligibility') may include the following data groups (structured collections of data elements):

Table 1 Request and Response examples for an Eligibility Service

Groups of Data Elements	Description
Request	
a. Decision Type	a. Decision type data group, e.g. *business domain* = 'retail banking', *decision category* = 'client eligibility' and *decision subcategory* = 'credit card eligibility'
b. Implementation Version	b. Business rule implementation version – allows applications transitioning to new version of rules on their own schedule
c. Party	c. Party involved in the eligibility rule, e.g. *client ID* and *financial institution ID*
d. Arrangement	d. Arrangement Information, e.g. *account type* = 'cash reward credit card'
e. Party-Arrangement	e. Arrangement associated with the party, e.g. *client ID*,

	financial institution ID, *client account type* = 'regular checking', *account number* and *account status* = 'overdraft'
Response a. Party	a. Eligibility decision for the party involved in the eligibility rule, e.g. *client ID*, *financial institution ID* and *decision* = 'eligible'
b. Arrangement	b. Eligibility decision for the arrangement, e.g. *account type* = 'cash reward credit card' and *decision* = 'not eligible'
c. Party-Arrangement	c. Eligibility decision for the arrangement associated with the party, e.g. *client ID*, *financial institution ID*, client *account type* = 'regular checking', *account number*, *account status* = 'overdraft' and *decision* = 'not eligible'

Rationale

Rule enactment service will expose web service interface supporting decision making in a specific business domain for a given decision category and optional cascading subcategories (e.g. 'banking' *business domain*, 'client eligibility' *decision category* and 'primary mortgage' *decision subcategory*). Limiting the interface to the selected *business domain* and *decision category/subcategories* allows creating a stable interface definition supporting all anticipated use cases within the selection. These use cases have enough in common to warrant that the interface input and output data can be defined using the same generic schema.

This interface provides an abstraction layer that will allow replacing rule enactment service implementation to leverage more advanced technology (e.g. upgrade from file-based to rule engine-based implementation) and/or a different platform (e.g. from Drools to IBM ODM) without a need to modify the consuming application. Such an approach increases application agility resulting in serious cost and time-to market benefits. This is especially significant in case of changes in the solution requirements or as a matter of adjustment to the enterprise rule-based technology rollout schedule.

Known Uses

The Agile Enterprise Rule Enactment Service pattern was refined by Princeton Blue for the 'retail banking' business domain 'client eligibility' decision category. Consequently it was successfully implemented by Princeton Blue clients for different decision subcategories and alternative implementation approaches including hardcoded (Java), file/database (Oracle) and rule engine (Pega) -based solutions.

Related Patterns

This pattern is leveraging the following published patterns:

- Layered Application [4]
- Service Interface [4]
- Service Layer [5]
- Separated Interface [5]
- Proxy [6]

This pattern can be used in conjunction with the following pattern:

- Reusable Rule Enactment Service Implementation – Pattern 2 described below.

Pattern 2

Name

Reusable Rule Enactment Service Implementation

Problem

A typical Rule Enactment Service implementation using a COTS rule engine leverages rule engine capabilities to evaluate business decision which represent just one step in the entire sequence of steps of the service internal flow. However, the other steps including input validation, data object load and response generation are implemented with custom scripts. Therefore, every new decision type has to be designed, coded and tested individually. The agility of the Rule Engine-based approach can be improved if rule engine capabilities are leveraged for all steps of the service internal flow. This would reduce time to market as well as solution initial and maintenance cost.

Context

A typical Rule Enactment Service implementation using a COTS rule engine is shown in Figure 3 below. The Consumer Application and the Rule Enactment Service Proxy are allocated to the Consumer Application and Integration Layers respectively. All service implementation constituent components are allocated to the Rule Enactment Layer. SoRs and Rule Repository are allocated to the Data Source layer.

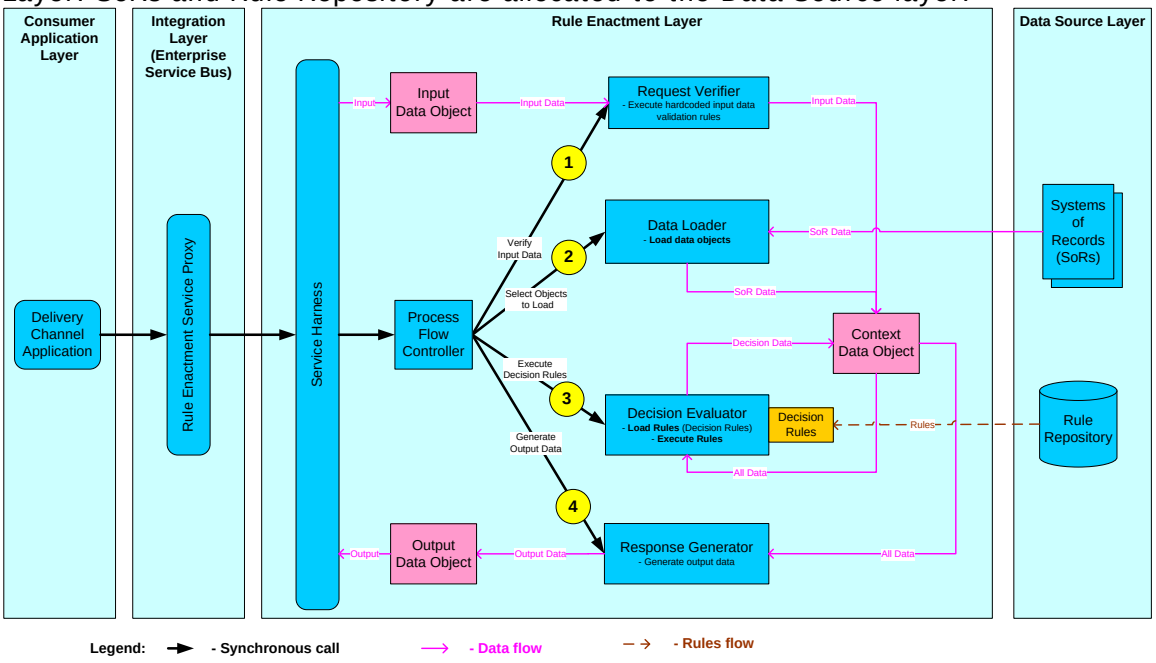

Figure 3 The existing context with Rule Engine-based implementation

The Service Harness component is usually a standard off the shelf capability of a rule engine. Rule Enactment Service Proxy invokes the Service Harness component using Web Service call with the input data packaged as the request message and stores it in the Input Data Object. It passes the control to the Process Flow Controller which executes its flow and returns the control back; after that the Service Harness

component retrieves the output data from the Output Data Object, constructs the response message and returns it back to the Rule Enactment Service Proxy.
The Process Flow Controller component sequentially invokes the following four components: Request Verifier, Data Loader, Decision Evaluator and Response Generator.
The Request Verifier component verifies the correctness of the input data in the request message and saves it in the Context Data Object.
The Data Loader component retrieves the required for decision making data from the SoR(s) and saves it in the Context Data Object.
The Decision Evaluator component retrieves the decision rules from the Rules Repository, retrieves input and SoR data from the Context Data Object, executes the decision rules on this data and saves the decision results to the Context Data Object.
The Response Generator component retrieves data from the Context Data Object, generates output data and saves it in the Data Output Object.

Forces

In a typical Rule Enactment Service implemented by leveraging a rule engine the only rule-driven step is the Decision Evaluator execution. Other three steps including validation of the input data, SoR data retrieval and response data generation implemented using custom scripts for the given decision type. Therefore, implementing the service for the new decision type or even refining service implementation due to the changes in input, output or SoR data will require script modifications as well as new functionality and regression testing. This reduces agility and increases time to market and cost.
The objective of this pattern is to leverage native rule engine capabilities in order to eliminate or greatly reduce need in custom scripting. The main forces for introducing this pattern are: reusability, extensibility, evolvability and maintainability of the service implementation.

Solution

The solution leverages native rule engine capabilities to eliminate or greatly reduce the need in custom scripting for validation of the input data, SoR data retrieval and response data generation steps. The business logic for the Request Verifier, Data Loader and Request Generator is captured in business rules stored in the Rule Repository.

Each of these components will retrieve, cache and execute the respective rules. As a result implementing the service for the new decision type as well as refining service implementation due to the changes in input, output or SoR data will not require script modifications and regression testing. All changes will be implemented by adding/modifying the respective rules in the Rule Repository. Possible exceptions are limited to binding/mapping to new objects in SoR and the Context Data Object.

Resulting Context

The resulting context for the Reusable Rule Enactment Service Implementation pattern is shown in Figure 4 below. The pattern solution is enhancing the existing solution by adding rules governing execution for the Request Verifier, Data Loader and Request Generator components.

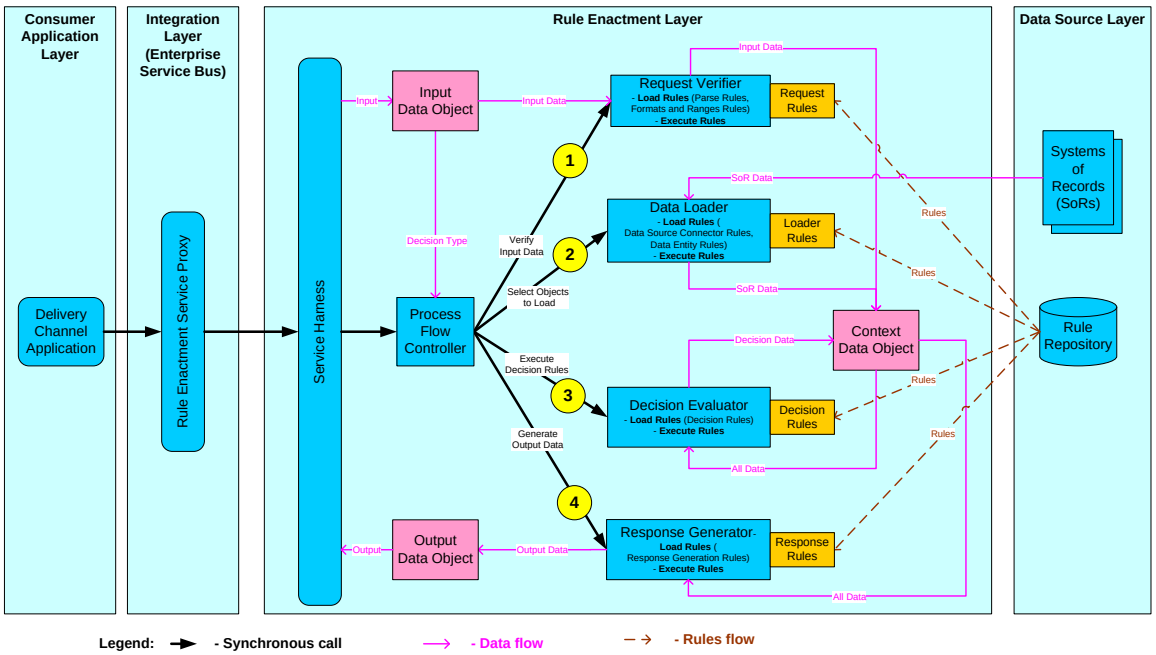

Figure 4 The Resulting context with Rule Engine-based implementation

The Process Flow Controller retrieves the decision type from the Input Data Object and passes it to each component it invokes, thus, enabling them to retrieve appropriate rules associated with the given decision type.
The rules are retrieved from the Rule Repository by each respective component, saved in local cache and executed. The results are saved in the respective data object.

Examples

An example of the Reusable Rule Enactment Service Implementation pattern leveraging Pega rule engine is shown in Figure 5 below. This example is a Pega platform specific pattern specializing the original platform independent pattern, i.e. Pattern 1.

Pega Execution Environment serves as the Rule Enactment Layer. Pega Service Harness serves as Service Harness. The Input/Output Session Context Data Objects located at the Pega Clipboard Page serve as the Input/Output Data Objects respectively. The Main Session Context Data Object located at the Pega Clipboard Page serves as the Context Data Object.

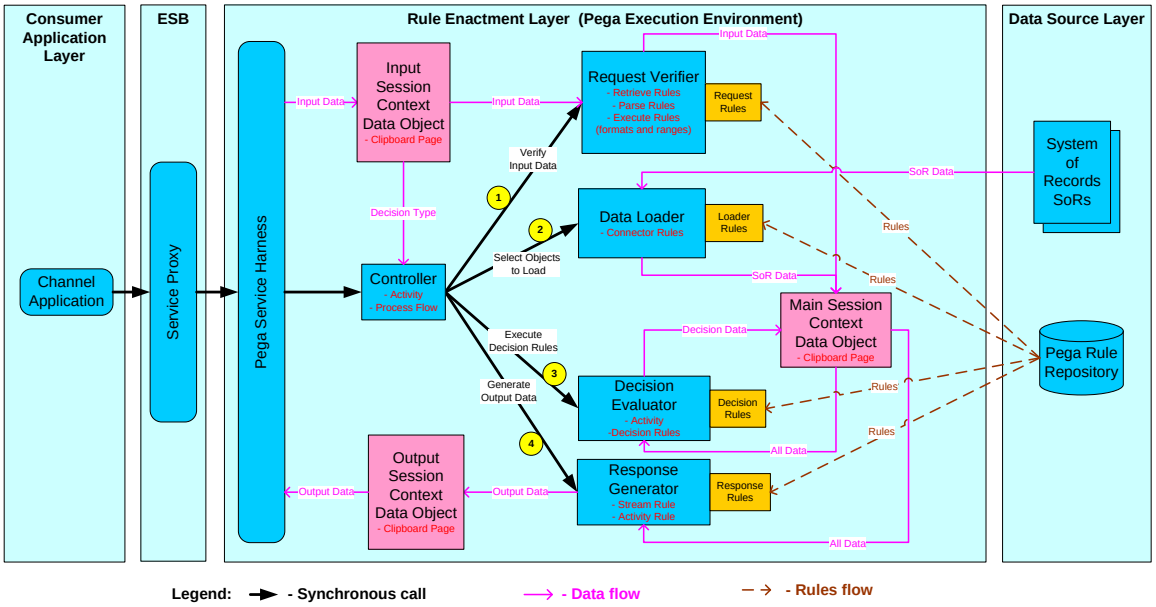

Figure 5 Example of Pega Rule Engine-based implementation

The Request Verifier component retrieves the data verification rules (verifying correctness of data format and values), parses and executes rules, then saves the input data the Main Context Data Object.

The Data Loader component retrieves the data verification rules (defining what data objects to be retrieved from SoR), parses and executes rules, then saves the SoR data the Main Session Context Data Object.

The Response Generator component retrieves the data verification rules (specifying how the output date shall be evaluated), parses and executes rules, then saves the SoR data the Output Session Context Data Object.

Rationale

The solution is leveraging rule engine capabilities for the Request Verifier, Data Loader and Response Generator (in addition to Decision Evaluator) and, therefore, allows creating implementations for new and modified decision types by merely changing rules in the Rules Repository. The exceptions may include minor modifications for Data Loader script and Context Data Object definition.

Known Uses

The Reusable Enterprise Rule Enactment Service Implementation pattern was successfully implemented by Princeton Blue clients for client eligibility decision leveraging Pega rule engine.

Related Patterns

This pattern is leveraging the following published patterns:

- Service Interface [4]
- Separated Interface [5]
- Proxy [6]
- Façade [6]

- Chain of Responsibility [6]

This pattern can be used in conjunction with the following pattern:

- Agile Enterprise Rule Enactment Service – described above Pattern 1

Conclusion

The Agile Enterprise Rule Enactment Service and Reusable Rule Enactment Service Implementation patterns described above can be used in any enterprise architecture with an objective to improve agility and reusability of the solutions that externalizing decision making business logic in separate components exposing service interfaces. The main business benefits of using these patterns are shorter time to market, lower initial cost of the subsequent decision type implementations and lower maintenance cost.

References

1. Project Management Institute (PMI), The Standard for Program Management, 2nd Ed. http://www.pmi.org/PMBOK-Guide-and-Standards/Standards-Library-of-PMI-Global-Standards.aspx
2. Vitaly Khusidman and David M. Bridgeland. A Classification Framework for Software Reuse. Journal of Object Technology, July 2006. http://www.jot.fm/issues/issue_2006_07/article1
3. TOGAF® 9. The Open Group. http://pubs.opengroup.org/architecture/togaf9-doc/arch
4. Enterprise Solution Patterns Using Microsoft .NET http://msdn.microsoft.com/en-us/library/ff650258.aspx
5. Martin Fowler et al. Patterns of Enterprise Application Architecture. Addison Wesley, 2003. http://martinfowler.com/eaaCatalog
6. E. Gamma, et al. Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, 1994.

Author

Vitaly Khusidman, Ph.D., Chief Technology Officer at Princeton Blue, a US based firm focusing exclusively on BPM implementations leveraging SOA. He can be reached at vitaly.khusidman@princetonblue.com

