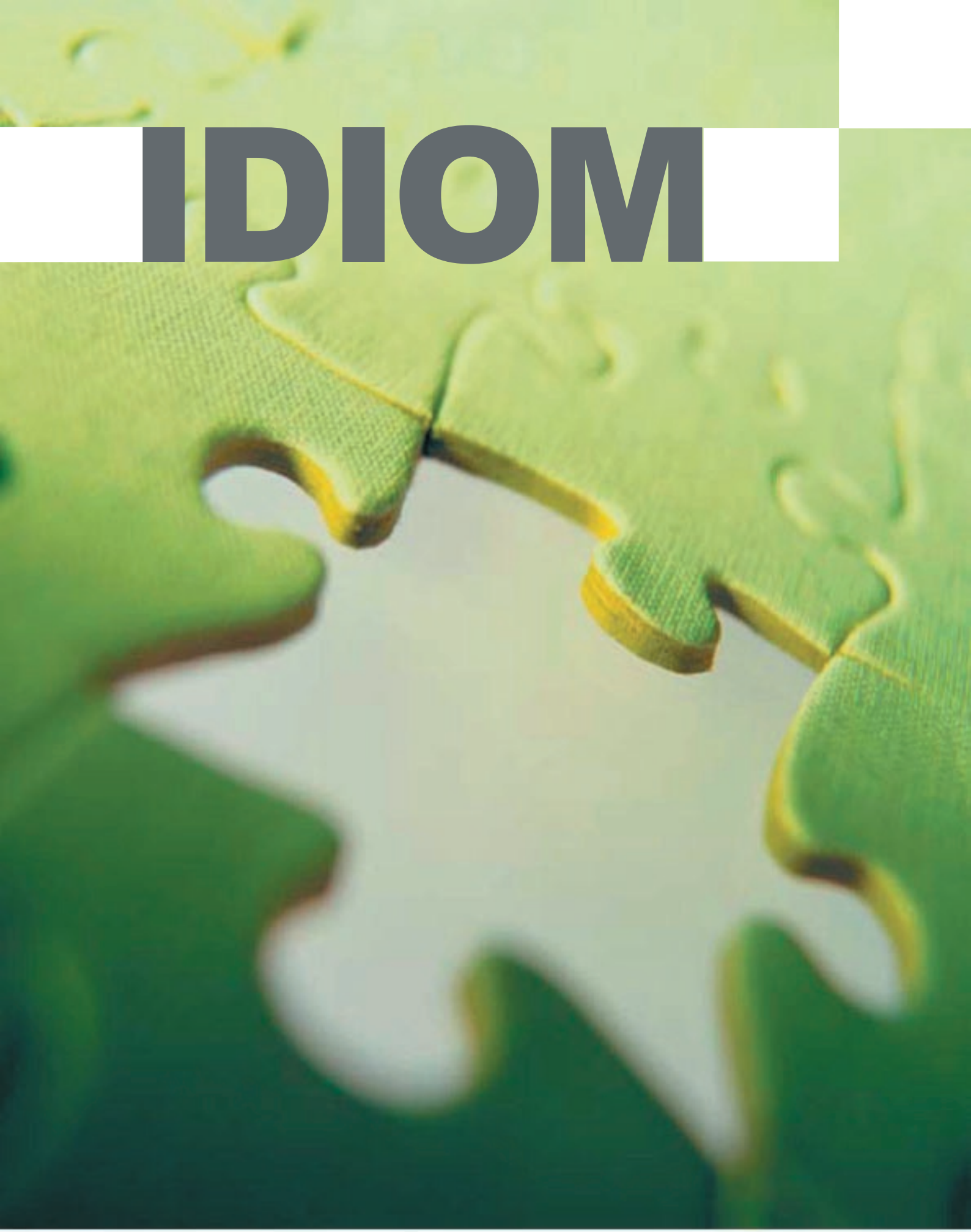
IDIOM



An Introduction to IDIOM Architecture

Introduction

The normal practice for implementing business rules is to encode the business rules within the application using the same programming language as is used for the implementation of the technical aspects of the application. However, the business rules become intertwined with other aspects of the application logic, including persistence, user interface and integration logic. This leads to several problems:

a) Verifiability – business rules implemented in a standard programming language (e.g. Java, C#) are difficult for non-technical users to understand. Therefore, it is difficult to verify that the implementation of the rule matches the requirement.

b) Maintainability – Implementing the business rules as part of the core application logic leads to an increase in the complexity of the application. This makes it difficult to vary one aspect without affecting others, thus decreasing the maintainability of the application. For example, changing the implementation of a business rule may adversely affect the scalability of the application. Conversely, changing a technical aspect of the application (e.g. persistence) may impact the correctness of the business rule.

c) Normalisation – As systems evolve and are adapted to new needs, business rules are duplicated to cater for the new requirements. This can result in multiple implementations of logic for one rule.

d) Documentation - As systems evolve, it is normal for any external business rules documentation to become unsynchronized with the implementation. The implementation becomes the sole repository of the business rules. This can be a major problem when migrating from a legacy system to a new application architecture as it can be very difficult to reverse engineer the intent of the business rules from the implementation.

These problems, caused by implementing business rules within the application using a standard programming language, are exacerbated by the use of modern object-oriented design methods and programming languages. Object-orientation is very good at managing the complexity of an application by abstracting and encapsulating data and the functions that operate on that data inside classes. However, business rules typically operate on more than one business object (or class). Therefore, when business rules are implemented using object-oriented programming languages they can become fragmented amongst multiple classes. This makes it difficult to alter or reconstruct a business rule. A similar problem occurs when implementing business rules in relational database management systems (RDBMS). Typical RDBMS-stored procedure languages are data-oriented, so rules tend to be expressed in a data-oriented manner. Furthermore, a common technique used by SQL programmers is the use of triggers. These are stored procedures that are executed asynchronously when certain database operations occur. The heavy use of triggers to implement business rules can seriously impact the maintainability and comprehensibility of a business rule.

The primary purpose of IDIOM is to enable a business to capture a specific class of business rules (the business decisions) without error, and to deploy them efficiently and at low cost into existing or new computer applications. IDIOM resolves the problems discussed above by separating and externalising the business rules from the technical aspects of the application. It allows organisations to define business rules in a business-oriented, declarative language that is easily verified and tested by business users. A generation process is then used to generate executable implementations of the business rules that can be deployed to the target application.

IDIOM Architecture

IDIOM has four major components. These are:

- Decision Manager
- Generator
- Test Executive
- Decision Engine

A diagram showing the conceptual architecture of IDIOM is shown in **Figure 1**. IDIOM decisions are developed against a definition of the business objects to be used within the application. This definition corresponds to the terms, facts and action assertion business rules[1], and is supplied as an XML Schema. Therefore, the first step in developing decisions is to import the business object definitions into the Decision Manager.

Once the definition of the business objects has been imported, the analyst, or rules designer, can use the Decision Manager to develop decisions and attach them to the appropriate nodes in the business object definition. The imported business object definition and decision definitions are stored in the Decision Repository. The IDIOM Test Executive is used to test and

[1] See the "Introduction to IDIOM Global" white paper for a full explanation of business rules.

© Idiom Software Ltd

validate decisions as they are developed. Once the decisions have been developed and verified they can be exported to the IDIOM Decision Generator. This generates the runtime implementation of the decisions. The generated code can then be deployed to the target application. Within the target application, the IDIOM Decision Engine provides an interface between the application and the decision implementation code and executes the decisions on behalf of the application.

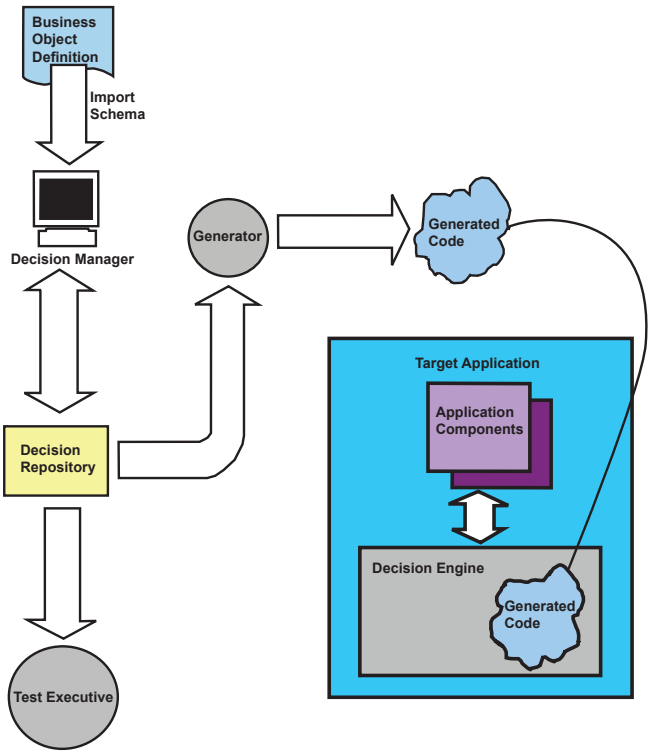


Figure 1 Conceptual architecture of Idiom Decision Suite

The key architectural features of the IDIOM runtime are:

- Passive – The Decision Engine only executes decisions when explicitly invoked by the calling application.
- Stateless – The runtime component does not store any state between activations. All dynamic data is input to the Decision Engine at invocation. The Decision Engine modifies the state of the input business objects and returns them in its output.
- Decisions are compiled – all business decisions are compiled into the native programming language of the target platform. There is no runtime generation or interpretation.
- Deterministic – Decisions are executed in a well-defined order. For a given input and rule set, the path to derive the output is defined and testable.
- Simple static interface – The runtime interface to the IDIOM Decision Engine remains the same no matter what the rule set, or input data. It does not change if the decisions are changed or regenerated. The application is isolated from the implementation of the decisions.
- Single point of integration – The Decision Engine does not (and cannot) access any external applications or data. All data required by the Decision Suite to execute a set of decisions is supplied in the call to the server.
- Decisions are written against a business object model defined by an XML Schema.

These characteristics of IDIOM differ substantially from those of more traditional rules engines. In contrast to IDIOM, traditional rules engines are "stateful", active and non-deterministic. Traditional rules engines maintain an active knowledge base of facts and rules. They execute continuously, firing rules in response to changes in input conditions and the state of the knowledge base. These rules engines are nondeterministic. A rule that has previously been determined not to be executable may subsequently be fired as a consequence to a change in the state of the knowledge base.

With any reasonably complex set of rules it is very difficult, if not impossible, to statically determine the flow of execution of business rules. As a consequence of their active and stateful nature, traditional rules engines are difficult to integrate with applications. They tend to drive the architecture of the system. Because these rules engines become the centre of the architecture to which they belong, they also tend to become an application server. This results in the business rules language being used not just for implementing business rules, but also for non-business-oriented application logic.

Quality Attributes

The architecture of IDIOM has a direct effect on many of the quality attributes of a software system. These quality attributes include quantitative non-functional requirements, such as performance and scalability, and also the more qualitative architectural or design principles.

Table 1 summarizes some of the most common and important non-functional requirements of a system and details how they are supported or affected by the use of IDIOM. In contrast to functional requirements, such as performance and scalability, architectural principles are qualitative in nature and more difficult to attach a metric to. These are principles such as flexibility, maintainability and usability. Each business must establish and prioritise

© Idiom Software Ltd
INVESTORS GUARANTY GLOBAL ALLIANCE

a set of architectural principles. The software architecture must then specify a design that attempts to achieve the functional requirements of the system whilst satisfying the least compromised set of quality attributes.

In general, it is not possible to design a system that completely satisfies all of the architectural principles and non-functional requirements. Therefore, the architect must select the best compromise design.

Table 2 provides a summary of some common architectural principles and details how IDIOM supports their achievement.

Table 1. Major non-functional requirements

Requirement	Supported by
Performance	• Decisions are compiled into the native programming language of the platform and execute at the same speed as handwritten code • The Decision Engine interface is a lightweight API with very little processing overhead • Generated code is stateless and thread safe and can support concurrent execution by multiple clients • Complex rule sets execute in milliseconds, not seconds
Scalability	• Statelessness of the Decision Suite gives very high scalability • Scalability is limited primarily by the scalability of the encompassing applications • Uses no expensive external resources such as database connections • Thread-safe implementation supports multiple concurrent operations
Availability	• Dictated by the availability of the encompassing application

Table 2. Typical architectural principles

Principle	Description
Configurable	• The only configuration required for the Decision Engine is the specification of which decisions to execute; this is passed into the Decision Engine upon each invocation.
Flexible	• Decisions can be changed and new rules developed and deployed without having to change the application. • The Decision Engine can be integrated in the manner which best suits the application. • Can be used in many problem domains

© Idiom Software Ltd

INVESTORS GUARANTY GLOBAL ALLIANCE

Table 2. Typical architectural principles continued . . .

Principle	Description
Modular	• The Decision Engine is a single coherent application module that does not depend on other application components for operation. • Applications are isolated from decision implementations, which can be enhanced / changed without modifying application code. • Follows service-oriented architecture principles
Cost Effective (TCO)	• The Decision Suite is a low cost solution that lowers the effort required to implement business decisions • Implementing and modifying decisions using IDIOM takes less effort than a standard programming language
Customisable	• IDIOM can accommodate any domain business object model and any decisions that are appropriate for that object model
Usable	• Simple API • Intuitive 'point and click' wizard-assisted environment for decision development and testing • Decisions are automatically transformed into English representation for verification and audit
Normalized	• Decisions and formulas are implemented once and can be referenced many times by other decisions.
Sustainable	• Decisions are easily adapted and maintained as business demands change.
Functional	• IDIOM can be invoked wherever a business decision is required by the application. Business processes do not have to be altered to accommodate the use of IDIOM.
Testable	• IDIOM decisions are testable and verifiable by business users prior to, and independent of, deployment to the application
Maintainable	• IDIOM increases the maintainability of a system by separating the business logic from the technical aspects of the application (e.g. presentation, integration, persistence, and concurrency). • The decisions can be changed and enhanced without altering the application logic. • Conversely, technical aspects such as the user interface or persistence mechanism can be enhanced without affecting the business logic

© Idiom Software Ltd

Application Integration

This section gives a brief description of how the IDIOM Decision Engine can be integrated into an application. It is not a detailed design description but provides a high level description of the general manner in which the Decision Engine can be used.

Because the IDIOM Decision Engine has a simple local call interface, it can be integrated in the manner that most suits the target architecture. Possible methods of integration include, but are not limited to:

- Direct integration using local programming language API
- Wrapping IDIOM in a stateless session enterprise Java Bean
- Providing a loosely coupled interface via message-oriented middleware such as Microsoft MQ, IBM MQ Series, Sonic MQ or Java Message Service (JMS).
- Using IBM Business Rule Beans Architecture[2]
- COM callable DLL
- SOAP-compliant web services interface

This flexibility allows the IDIOM Decision Engine to be integrated into an application in the manner that bests suits the particular architecture, rather than having to adapt or contort the architecture to accommodate the Decision Engine.

The primary interface to the Decision Engine is a method that takes as its primary arguments a list of business objects to operate on and a description of the decisions to execute. Upon completion, the method returns an updated and enhanced list of business objects. The input and output business objects must be XML DOM Level 2 compliant documents[3].

Another point to note is that the Decision Engine has a single point of integration - it integrates with the application entirely via the chosen call interface. The Decision Suite does not require integration with a database or any other external resources. All data to be used by the Decision Engine in the execution of decisions is supplied by one of two methods:

- Static reference data can be loaded into the Decision Engine at initiation. Decisions can then reference this information during their execution.
- Data that may change between invocations of the Decision Engine, or are specific to the business objects being operated on, must be included in the business object set that is provided as input. This single point of integration provides three major benefits:
  - It simplifies the integration of the Decision Engine to the application.
  - It enforces the separation of concerns between business logic and data persistence.
  - It allows one set of decisions to support and be used by multiple applications on diverse platforms, including, for example, business partners' applications. Conversely, one application can execute decisions from multiple 'knowledge suppliers'.

At first glance, the requirement to provide all necessary data as input to the Decision Engine may seem to impose a performance burden on the architecture. In practice, this is not the case, especially if the Decision Engine is invoked via the local call API and the business objects are passed to it by reference. However, one impact of this feature is that for any reasonably complex implementation, it will be necessary to precede the invocation of the Decision Engine with an object assembly step, to construct an input form of the business objects with the essential information that is required for the decisions to execute correctly. In the following sections, more detailed descriptions on how to integrate the Decision Engine are provided. Specific examples provided are:

- Direct API integration
- Service-oriented web service

It is expected that these two architectural styles will be the most common.

Direct API Integration

Although there are many possible mechanisms for integrating the Decision Engine within an application, the simplest and most efficient is via direct API integration. The diagram in **Figure 2** depicts this integration style.

The Object Assembler is responsible for the transformation between the internal business objects representation and the DOM style representation required by the Decision Engine. It is also responsible for loading any lookup data from the database into the business objects, and potentially for trimming any unnecessary data from the business objects. On return from the Decision Engine, the Object Assembler performs the

---

[2] Please refer to the IDIOM and Business Rule Beans document on our website

[3]The Java Decision Engine can also take JDOM compliant documents.

opposite function, transforming from the business object(s) to the internal representation used by the application. When the number of application clients calling the Decision Engine is very low and the business objects required by the decisions are small and simple, it is possible to invoke the Decision Engine directly without the Decision Facade or Object Assembler components. However, in general, these two components simplify and improve the integration with the Decision Engine.

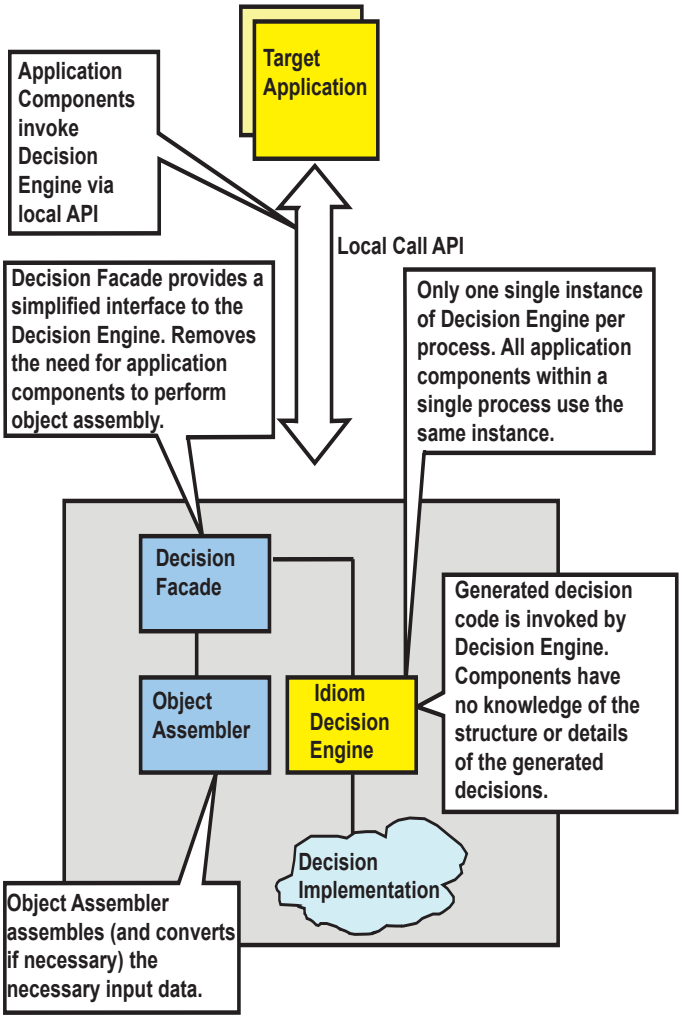


Figure 2 Local API integration of IDIOM

Although the overhead of calling the Decision Engine is very low, the business objects passed to and from it can be quite substantial. This could impose a performance overhead when communicating with the Decision Engine remotely (e.g. via EJB or web service). Therefore, the direct integration API provides the most efficient integration with the lowest overhead.

The disadvantages of direct integration of the Decision Engine are that:

- This method requires the application to be implemented in one of the supported platforms
- Scalability is limited by the scalability of the application

Web Service Integration

The IDIOM Decision Engine is versatile in that it can be targeted to any application architecture that supports either Java or .Net platforms. However, sometimes a business wants to integrate IDIOM with an application that does not directly support either of these platforms. In this case, it may be desirable to integrate the Decision Engine via message oriented middleware or a web service.

In many respects, the IDIOM Decision Engine is well suited for use as a web service:

- It has a simple stateless interface
- It is document-oriented due to the XML based business object interface
- It is location transparent

However, in practice, the Decision Engine interface is quite fine-grained. It is intended to provide a business rule execution service to an encompassing application, rather than act as a standalone business application. Thus, to obtain maximum benefit and performance from the Decision Engine in a web services environment, we recommend the Decision Engine be deployed using a web services facade architecture, as seen in **Figure 3**.

The key feature of the web services façade architecture is that it provides a coarse-grained, business-oriented interface to the underlying service.

The key components of the web services facade architecture are:

- SOAP interface – it provides a SOAP compliant interface to the underlying facade
- Web Services facade – this provides the coarse-grained, business-oriented interface that client applications interact with
- Object Assembler – responsible for transforming data from an external representation to the internal representation used by the Decision Suite.
- Session Manager – manages any inter-invocation state that may be required

© Idiom Software Ltd

INVESTORS GUARANTY GLOBAL ALLIANCE

- Orchestration Manager – responsible for managing the workflow and routing within the facade. Can use IDIOM decisions to manage the workflow
- IDIOM – the IDIOM Decision Suite runtime component and generated decision code.

In practice it may be desirable for the facade to include more business features than just a simple interface to the Decision Engine. The same web services facade architecture supports integrating additional business services within the application service.

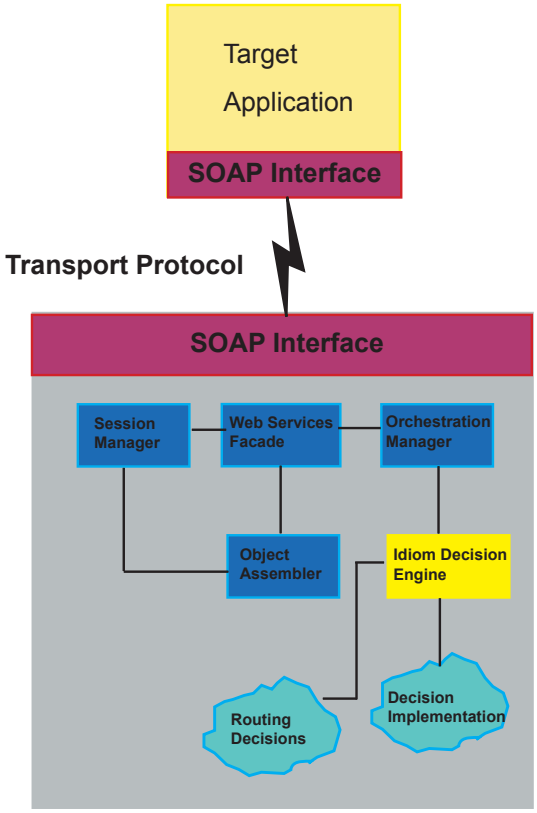


Figure 3 Web Services (SOAP) Interface to IDIOM

Development Process

Another area where IDIOM has a major impact is on the software development process and lifecycle. No matter what software development process is used in the development of a system, it is essential to gather requirements prior to coding and testing. This is true regardless of the process used: whether the process is based on the waterfall model, where all requirements are analyzed prior to construction; or an iterative process such as Rational Unified Process; or an agile process such as Extreme Programming (XP), where requirements analysis and construction are performed together in very short iterations. In each case, you can't build what you don't understand. One of the problems with software development is that it is extremely difficult to obtain all requirements, to understand them fully, and to get them right in the first iteration. For example, use case modeling is good for identifying the coarse-grained behaviour of an application, but it is not so good for identifying all of the business rules associated with the use cases. Attempting to build an application early, without understanding all of the associated business rules, faces the risk of significant rework as the business rules are refined.

Using IDIOM within an application architecture has a significant impact on the software development lifecycle.

Application construction can start early, without knowing the business rules, as long as it is known where the business rules are needed.

- Business rules analysis and development can proceed in parallel to the application development.
- Business rules and application infrastructure can be tested independently. A change in a business rule is not going to require regression testing of the entire application. Likewise, a change in the user interface or persistence components does not entail regression testing of the business rules.
- Changes to the business rules can be made late in the development lifecycle, and at any time thereafter, without undue effect on the application infrastructure.

The above benefits combine to reduce both the time required to develop the application and the risks associated with the lack of understanding of requirements.

Summary

IDIOM provides an architecture that supports the delivery of a high quality, agile application without constraining the application's architecture. It provides a high performance, scalable runtime environment for executing business decisions that have been declared in a high-level business-oriented desktop tool. The Decision Engine can be integrated with an application in the manner that best suits the target architecture, rather than forcing the architecture to be designed around the Decision Engine. A summary of some of these integration scenarios is provided in **Table 3**.

These are just a sample of what is possible and achievable with IDIOM. They give an indication of standard integration techniques and their effect on the quality attributes of the architecture. IDIOM does not significantly constrain the architecture and helps applications to achieve many of the more difficult-to-attain quality attributes. It can help to reduce the development time of applications whilst reducing the risk caused by unexpected complexity and poorly understood requirements. Most importantly, it helps businesses to achieve greater agility by enabling the rapid development, enhancement and maintenance of business decisions externally to the application.

© Idiom Software Ltd

INVESTORS GUARANTY
GLOBAL ALLIANCE