Taming the IT Beast with Decision Centric Processes
Increase Business Agility, Reduce System Complexity
The primary subject of this article is *process*, a word that is generally both indefinite and nuanced when applied to systems development. In this article we describe how process as a concept becomes both simpler and more definitive when it is integrated with decisioning. The combination of process and decisioning extends the ‘decision centric’ development concepts that we have evolved over the last 15 years. These concepts combine into a proven, practical, and robust methodology that leverages decisioning and agile techniques to fundamentally simplify commercial software development.

Our earlier published articles have chronicled the evolving concepts, starting with a 2006 article in the Business Rules Journal, “Decisioning: A new approach to Systems Development” ¹. More recent articles are online at both Modern Analyst and the Business Rules Journal, and are referenced in this document.

- Requirements and the Beast of Complexity² (2010, 22,600 downloads as at March 2015)
- Decisioning – the Next Generation of Business Rules³ (2013)
- The Role of SQL in Decision Centric Processes⁴ (2014/15)

The core of the decision centric approach is a tightly scoped process that we call a ‘business transaction’. The purpose of a business transaction is to manage the states that comprise the full lifecycle of the business entities that represent the value proposition of any organisation. At the heart of each business transaction is a decision model, which is a codified and executable proxy for the business policies that govern the entity, and the value it represents.

In so doing, the business transaction binds the business policies to the entities they govern, thereby becoming the definitive mechanism for creating value within the business.

This combination of entity and business policy embodied in a business transaction strips complexity out of commercial transaction systems, simplifying the implemented system image and its development.

When the methodology is used to address complex commercial transaction processing, the *demonstrated results* include substantial improvements in the key development metrics of time, cost, and risk; and often, similarly substantial improvements in runtime performance. These encouraging results are accompanied by similar scale improvements in business agility, better alignment between business strategy and operational systems, and systems that are more transparent, more agile, and more durable.

Everything presented herein is based on common sense and a desire to simplify our overly complex IT world; there is no magic. The approach has been applied to many domains including lending, insurance, investment and pension administration, payroll, utility and health billing, clinical health, logistics, and government at all levels. It does work, and it does deliver outstanding results!

Mark Norton, IDIOM Limited

---

INTRODUCTION

Our dog Max is a master at intercepting Frisbees.

Our city (Auckland, NZ) tries to be a master at optimizing traffic flows through a series of traffic lights.

These two tasks will serve to introduce our initial observations regarding process.

What is a process? This Free Dictionary definition\(^5\) is typical:

“A series of actions, changes, or functions bringing about a result”.

It may sound banal to observe that a line of computer code on one extreme, or life itself at another, are processes according to this definition; but therein lies the problem. Everything that happens through time (‘a series of’), and which has an observable ‘result’, is a process. Even more disconcerting, every ‘process’ can be grouped recursively into a network of bigger, overlapping processes. The bottom line is that all human activity is a continuum of process – one big complex über process that is made up of an interlocking network of smaller and smaller processes.

So what exactly is being studied when we do process analysis? The only fixed point available to us in the above definition is a ‘result’. A result in this context is something that we want to achieve (we assume that ‘result’ means an intended result, not just any result), so the point of process analysis is to identify results that are relevant to the business we are in (whether commercial or not), and then the steps required to achieve them.

Scoping the process appropriately is critical if we want to specify and build the process. If the process scope is too small, the result has no business relevance unless and until the excessively small processes are bound together into larger units of work. If we scope too big, then we introduce more results, more steps, and more pathways through those steps. In either case, we increase the process complexity relative to the intended result. Like Goldilocks’ porridge, we need to scope ‘just right’ if we are to minimise complexity.

This is the point of the dog and the traffic illustrations. Both have simple solutions, but only when we scope the process right and only include those activities that are immediately proximate to the result.

For Max, catching the Frisbee is a simple problem\(^6\). Dogs don't have PhDs in mathematics even though the problem being solved – to be at the right place at the right time as the Frisbee ducks and weaves through the air subject to gravity, wind and rotational forces – is mathematically complicated. All Max does is move his body so that the line of sight to the Frisbee is more or less constant. By doing this his

---


\(^6\) Hat tip to Andy Haldane as the source of the analogy of the dog and the Frisbee. [http://www.bankofengland.co.uk/about/pages/people/biographies/haldane.aspx](http://www.bankofengland.co.uk/about/pages/people/biographies/haldane.aspx)
mouth will intersect with the Frisbee. Simple enough for a dog to understand, even though a mathematician would be challenged.

For the city traffic planners, there is also a simple solution. For each set of lights, increase the allocated time for the direction of travel that has more cars, and reduce it for the direction of travel that has fewer cars. This simple solution is now patented\(^7\), with claims of substantial (>30%) improvements in overall traffic flows, reduced fuel consumption, etc.

**Types of Processes**

For productive analysis of processes, we need to ensure that the scope of each process that we want to analyse is relevant to our business need. To do this, we need to focus on the desired *business results*.

**Business Transactions**

For the purposes of this paper, we assert that every meaningful *business result* must cause a change in the *state* of a related *business entity*. A business entity is something of self-evident value to the business. For example, a loan is a business entity, and a change in its state may result from it being approved.

Ergo, changes of state on a business entity are the ‘results’ that we are looking for.

With the result now in focus, we then set the scope of the process to only include those activities that are required to achieve the change of state. In general these activities will be synchronous (but not always mandatory) activities – that is, a set of activities that are performed in sequential order inside the boundary of the process.

These tightly scoped processes are the smallest units of activity that it is possible to have while still achieving a meaningful business outcome (because we can’t have half a state change!) In deference to the importance of the word ‘business’ in the previous sentence, we will now call these processes ‘*business transactions*’.

**Use Cases**

We now introduce another class of process. These are usually bigger units of work that include one or more business transactions that collectively achieve the purpose of the organization. We will refer to these aggregate processes as ‘use cases’, which Wikipedia\(^8\) describes as follows:

> “In software and systems engineering, a use case is a list of steps, typically defining interactions between a role (known in Unified Modeling Language (UML) as an "actor") and a system, to achieve a goal. The actor can be a human, an external system, or time.”

In our interpretation of ‘use case’, the business transactions making up a use case will be asynchronous – that is, they can act independently, provided that the business dependencies between them are respected.

**Process Orchestration**

The boundary between business transactions and use cases is fluid. Activities that might one day be included inside one larger business transaction might subsequently be separated out into a business transaction in their own right and vice versa. Our goal in implementing the decision centric transaction

---

\(^7\) [http://www.google.com/patents/US20080094250](http://www.google.com/patents/US20080094250)

The concept is to maximize agility and to minimize constraints for the business, by providing the business with an ability to adapt its processes quickly and appropriately to changes in its own requirements, or to changes made by itself or others (such as a Regulator) within the over-arching network of processes that we called the ‘über process’.

Let’s illustrate this with an example of how the activities in business transactions and use cases might usefully be rearranged.

Assume an existing business transaction for ‘make a loan’ that evaluates security and customer data to approve, decline, or refer a loan:

a) Assess the quality and value of the security;
b) Assess the credit risk of the borrower;
c) Assess the repayment ability of the borrower;
d) **Decide on the change of state** (i.e. approve, decline, or refer), and if approved, the maximum amount to lend, the interest rate to charge, and any terms and conditions to be applied.

In our example, a local bank has been very successful with a loan product that moves the boundary between the above business transaction and its use cases. For this new product, the bank evaluates the security value of every residential property in the country, in advance. This is a state change on the security. They then separately evaluate the borrowing capacity of every bank customer, also in advance. This is a state change on the customer.

The critical step d), the actual agreement to lend, along with calculation of price and terms and conditions, is only determined when the customer is onsite at a property. The customer simply uses a mobile phone app to indicate which property they are looking at, and the bank responds immediately with an approved loan amount for the purchase. This is a state change on a loan.

By using an agile approach for the assembly of both business transactions and use cases, this new variation alongside the prior lending process is easily achieved, as are other variations and extensions that are easy to imagine – for instance, upsells for insurances or other services, etc.

With the growing popularity of service oriented architectures, processes are no longer constrained within system or organisation boundaries – for instance, crossing from the bank to government agencies for the property valuation data; bank to insurers for upsell products; etc.

And the processes can also span time periods to manage workflow – for instance loan reviews, interest and repayment schedules, and so on.

If we had implemented the original business transaction with a bespoke computer program we would have constrained both business flexibility (the ability to achieve the result in different ways) and agility (the speed and ease with which the business can utilize the flexibility). The flexibility represents the range of options available to the business, while the agility represents the ability of the business to rearrange those options quickly and easily at any point in time. When properly implemented, flexibility and agility should allow an organisation to adapt in a way that is nimble, continuous, and perpetual.

This article outlines an approach to systems development that focuses on processes that are dynamic and extensible, and which can adapt continuously and perpetually.

**Agile Processes**

What do these agile processes look like?
They are meta-processes that are materialized on a just-in-time basis using decision controlled workflow inside each business transaction. To an external observer, the exact nature of the materialized process is only apparent in hindsight, and even then its full extent is unlikely to be visible.

Only at the time of execution is the full scope of the process resolved by the decision models that are controlling the workflow.

Firstly, the various ‘activities’ that comprise each business transaction are dynamically determined and invoked according to the current context and needs of the business transaction.

And then the business transactions that comprise the use cases link to each other, one at a time. The full extent of the chaining is not known to any one business transaction or in fact to any other actor. Each business transaction only knows its immediate successor business transactions (if any) because they trigger them. The linking between activities and transactions can cross system and organizational boundaries; and they can link through time using future dated links. The concept of a process boundary becomes nebulous and is in fact irrelevant. The community of processes becomes nimble, continuous, and perpetual. Because of this simple one-to-one linkage between activities and transactions, we refer to this as ‘binary BPM’.

**Figure 2**

**Game Changing Agility**

What we mean by agility is a lack of constraint on the business; its systems should support new products and new services in market time, not in systems development time. Some markets may demand a high rate of change; some may not. Either way, we need to be mindful that to be truly agile, the system is by definition perpetually in a state of continuous change. In order for this to not become a constraint, we need to actively consider continuous, perpetual versioning of the entire system, and we need to think of processes in new ways that support nimble, continuous, perpetual change.

Having a system wherein processes are materialized as a series of dynamically selected activities, while at the same time adapting the fundamental building blocks that form those activities, may seem like a recipe for chaos. In practice, the activities are functionally discrete, tightly engineered, and fully tested software components that form part of the underlying fabric of the system.
These otherwise static components are activated by business transactions at the behest of policy driven decision models inside each transaction, which evaluate the context data to determine whether and when an activity is required.

These same decision models also cause each business transaction to link to its dependent business transactions (if any), causing a dynamic chaining of activities into the larger processes (that we called use cases) without limit.

Provided that we individually test the decision models behind every business transaction for completeness, consistency, and correctness, then the development process is robust and risk averse. Notwithstanding, we usually further mitigate development risk by routinely regression testing all existing business data through all relevant decision models whenever any changes to the system are made. These regression tests are twofold: a) to confirm the accuracy of changes made; and b) to confirm the absence of unintended consequences. With the help of appropriate tools, we can routinely test millions of complex test cases in a few hours on commodity, low cost systems.

Our ultimate goal is for “game-changing agility for the business in its chosen market”, with appropriate consideration also given to quality, reliability, scalability, compliance, security, and transparency, which are all functions of the toolset used to support the approach. In a later article (to be published shortly) we will describe a candidate set of tools that fully supports the decision centric development approach as described in this article and its predecessors.

**Why it Works**

Why do we observe profound improvements in both business agility and development performance with this approach?

While there are many reasons discussed throughout this article, the small number of components involved is one outstanding factor – fewer components means a simpler system.

The total number of activities that need to be built into the underlying system is very limited – no more than a few hundred for a full service bank or similar organization. This small number of activities may be surprising. One reason is high reuse – for instance, all business transactions can be manifested by exactly one underlying activity, which is a generic transaction container. The numerous business transactions that appear at the business layer are all executed by this single transaction activity, which adapts to its specific entity management roles by virtue of the decision models that are loaded within it.

The primary task of a business transaction is to manage its entity’s life cycle by orchestrating the activities it requires. For the sake of clarity, the transaction’s decision models manage the *entire life cycle* of the entity, not just one specific state change. Therefore there is just one decision model (or a small collection of models acting logically as one decision model) for each entity type, not one per state change.

Furthermore, in the case of product related entities (for instance, a loan, an insurance policy, etc.), many products can be described by one entity type, so that the decision models are usually designed to manage an entire *class* of product.

In these cases, the decision model usually adapts to the needs of each product within the class by inspecting a product *configuration document* that belongs to the class. How many products can we support with a single decision model? We have examples within our insurance customers where the entire life cycle of dozens of insurance products are supported by a single decision model, which manages the full cycle of quote, bind, amend, endorse, cancel, lapse, and renew. In essence, one
versatile decision model can transform a single low level activity into a universal product engine for the organization.

The above outline indicates how a small and concise system image containing only a few moving parts can appear to materialize many, many business processes. It is a system simplified!

**BUSINESS TRANSACTIONS – BACK TO THE FUTURE**

**What is a Business Transaction?**

The transaction was once recognized as a mainstay of systems development and design; it was a more important concept for systems development than now. However from a purely business perspective – what can be called the business model – the reality remains that successful business requires successful business transactions. It cannot be avoided.

Today a web search returns many definitions of transaction that are less certain of their ground, for instance, this rather loose definition from a self-proclaimed computer science dictionary:

> “An action or series of actions carried out by an application program or user which reads or updates the contents of a Database.”

UML takes a different tack, defining a transaction as a multi-part process that performs tasks on behalf of multiple parties, although a single logical commit is inferred.

From the UML Business Process Definition Meta Model:

> “A transaction is a kind of Embedded Process which enclosed activity (ies) can be rolled back by means of an Actor”

And from the UML Business Process Model and Notation:

> “A transaction is a Sub-Process that is supported by a special protocol that insures that all parties involved have complete agreement that the activity should be completed or canceled”

> “A Sub-Process that represents a set of coordinated activities carried out by independent, loosley coupled systems in accordance with a contractually defined business relationship.”

There is an echo of the above in this definition from the Microsoft Development Network library:

> “A transaction is a group of operations that have the following properties: atomic, consistent, isolated, and durable (ACID).”

In each of these definitions, which cross the breadth of mainstream development, the definition is silent about the basis for identifying and grouping the ‘actions’, ‘activities’, and ‘operations’ respectively. None of the definitions provide a business rationale for the transaction concept they describe.

One purpose of this article is to promote the concept of a transaction back to its role as the fundamental building block of systems. Why is it so important?

---

9 [http://wwwcomputingstudentscomdictionaryword=Transaction](http://wwwcomputingstudentscomdictionaryword=Transaction)
10 [UML Business Process Definition Meta Model para 6.5.2.29; it has no agreed BPEL construct para 9.8.](http://wwwcomputingstudentscomdictionaryword=Transaction)
11 [UML Business Process Model and Notation v2.0.2 Table 7.2.](http://wwwcomputingstudentscomdictionaryword=Transaction)
12 [UML Business Process Model and Notation v2.0.2 Glossary.](http://wwwcomputingstudentscomdictionaryword=Transaction)
13 [http://msdnmicrosoftcomen-uslibraryaa366402VS85aspx](http://msdnmicrosoftcomen-uslibraryaa366402VS85aspx)
Because the transactions we are describing are the fundamental building blocks of business itself. A state change on a business entity is the essence of value creation in any business. By way of contrast, if there is no change in any business entity, then no value can have been created.

For the purpose of this paper, we now define a ‘business transaction’ as follows:

Business Transaction: The activity that is initiated by an event that changes the state of a business entity, and which when complete leaves all systems compliant with all business policies relevant for that entity in that state.

This modern day transaction is bigger than its predecessors. It increasingly uses services to extend beyond its own host organisation and interact with users and other activities in real time (e.g. to acquire data, or invoke supporting activities such as predictive analytics engines). It may be recognizable by internet users as a user session of the sort that concludes with a credit card payment. And because of its expanded size and scope, it may even include ‘auto save’ and ‘restart’ capabilities within the business transaction itself. But it will still adhere to the traditional ACID concept: it will complete a valid state change on a business entity, or it will leave that entity unchanged.

The Decision Model

The business transaction is controlled by a decision model, which is a model of the business logic that implements the business policies that determine the business outcomes. As one of our customers has commented, ‘decision models are the DNA of the business’.

The decision model itself is unaware of technology, systems, or processes; it is strictly business owned and operated to evaluate the transaction data against the broader business intentions, know-how and options. A decision model is comprised of smaller, logical units that we call decisions, defined as follows:

Decision: A single definitive datum that is derived by applying business knowledge to relevant data for the purpose of positively supporting or directing the activity of the business.

Decisions are defined in the context of a decision model, which is defined as:

Decision Model: An ordered assembly of decisions that creates new and proprietary information to further the mission of the business.

Single Entity Focus

Our definition of business transaction implies that it executes in favor of a single business entity, which contrasts with the UML perspective at least. The single entity concept arises because we focus on the point where the critical change in business value occurs. Even if multiple business entities in multiple systems are going to change state as a result of an external event, if the primary value change does not occur, then the dependent entity updates should also not occur.

By focusing the initial business transaction on the primary entity where the value change will occur, followed by a series of dependent, secondary business transactions, an otherwise larger and more complex process is naturally broken down into atomic business transactions that align exactly with the reality of existing business entity and/or system boundaries, all of which are processed in accordance with the rules governing the primary business transaction.

This can be a major simplification. Usually, the secondary transactions are simply registering the fact that the primary transaction has occurred; in normal circumstances they have no ability to constrain the
primary transaction – for instance, standard ledger updates; registering a loan against a security; notifying acceptance of a risk against a reinsurance treaty; etc.

The complexity of the downstream transaction workload on the secondary systems is limited because the secondary updates are guaranteed to be compliant with the new state of the primary entity, so they can be processed using relatively simple logic within their respective transactions.

However, the secondary systems are independent actors, so that yet further downstream processing may occur unbeknown to the primary business transaction. This is both correct and appropriate.

The net effect of the implied cascade of business transactions, which are all ultimately triggered by a common external event, agrees with the description of a use case offered earlier. The use case is the sum of the business transactions that occur in response to the external event, not all of which may be visible to any particular actor.

Our approach for managing these downstream transactions is to generate new events as ‘change vectors’ inside the primary business transaction; this ensures that all changes that are required by the original state-change, in all systems, for all parties, both now and in the future, are compliant according to the same set of rules at the same point in time.

We then extract these change vectors from the primary business entity when it is saved and persist them into that entity’s database within the same commit boundary as for the entity itself. This gives the updates ‘state’ – from where they can be safely processed asynchronously into all affected systems while still achieving the ACID objectives for both the primary and secondary business transactions.

Before we move on, we will expand on the four headline concepts used in our business transaction definition:

- Business Policy
- Business Entity
- Event
- Activity.

**Business Policy**

Most definitions of a business transaction infer that the system should be in a valid state on completion. Business Policy is the ultimate arbiter of how this ‘valid state’ is defined.

Business decisions must be made in accordance with business policy, so policy is what dictates proper decision making within a business transaction.

Business policy is distilled from a range of sources, including but not limited to, regulations, contracts, and other legal commitments. The pro-forma contracts that we call a product or a service can be described as groups of policies bundled together in a way that is meaningful for a customer.

Business policies must align with the objectives, values, and other strategic statements that define the business and its value propositions. The core business entities should be self-evident from these value proposition(s); and because value can only be created when there is a change of state in the business entity, the value proposition must also describe their critical state changes i.e. their life cycles.

In practical terms, the value proposition is distilled into the organization’s products and services, so that product and service contracts define how value is created for both the business and its customers.
In each case, the business policies must define whether and to what extent the product or service related state changes will occur. These are the primary decisions (the green boxes in Figure 3) that will determine whether the state change will be allowed; and if so, what its value will be; and what terms and conditions might apply. These decision elements are common across most domains.

These primary decisions will typically be collated with other supporting decisions into a complete decision model for the business transaction (see adjacent diagram):

- Precursor decisions will usually be present to assess the extent, quality and validity of the event data as presented, using the existing entity data as context. Additional supporting data may need to be acquired by triggering appropriate (synchronous) activities.

- Decisions will transform the event and entity data into the idiom (viz. ‘a specialized vocabulary used by a group of people’)\(^\text{14}\) of the business policy itself, which is also, by definition, the language of the primary decisions that implement it. The idiom includes the nouns and noun clauses of the proprietary language that we assert is always present when describing business policies\(^\text{15}\).

- The primary decisions execute the business policy using the idiom of the business.

- Follow up decisions then determine what follow-on events need to be generated as a result of the state change (a proxy for workflow).

Dynamic transformation of raw data to data that is compliant with the business idiom is needed because a) it is rare that the event data is already in the required state for adjudication by the policy based decisions, and b) the business idiom is fluid and ever changing – it is the essence of the business rules, changing whenever the business changes.

---

\(^\text{14}\) [http://www.thefreedictionary.com/idiom](http://www.thefreedictionary.com/idiom) [a. ‘A specialized vocabulary used by a group of people’]

We find that these idiom driven, transformational decisions usually form the bulk of the business transaction’s internal work effort by a large margin. For instance, in our earlier loan example, the assessment of the security, credit risk, and repayment ability all require significant transformation of raw data before final decisions can be made.

All of these decisions, but particularly the primary decisions, can be defined and validated by the business as part of the distillation and refinement of business policy. They are not dependent on any system, process, or implementation and can (and should) be done as an ongoing business task rather than a project related development task – this decisioning is the heart of the business because it defines specifically and exactly how value is created. Systems require this policy defined decision making if they are to manage the creation of value for the business. Policy defined decision making that is fully described and empirically tested to ensure that it is complete, consistent, and correct becomes the pre-eminent definition of the system, even before there is a system.

Following this approach dramatically simplifies the traditional requirements process, as well as providing requirements certainty that reduces development risk overall. In fact, in a properly architected ‘decision centric’ system, these requirements can be implemented as content in a dramatically simplified host application container that we will describe shortly.

## Business Entities

We can derive directly from the business value proposition the business entities that are relevant to the business; these entities are the entities that define and register value for the business. As such, they can often be found in the balance sheet. And because of their unique role as the abstraction of value in the business, they will usually be uniquely identified by the business – the existence of business assigned identifiers is a strong indicator of a relevant business entity.

The data describing a business entity can be substantial (hundreds of thousands of data nodes per entity is plausible). For the sake of clarity, we assume that the definition and structure of this data complies with the rules of [normalization](http://en.wikipedia.org/wiki/Database_normalization). However, this is not to be confused with deconstructing the entity data into normalized data relations (aka tables) in a relational database. This data is tightly bound to its owning entity, and can never be used independently of the entity identifier by any business transaction, by definition. Therefore, storing all of the business entity data as a single complex data-type is both logical and practical.

This simple design practice again reduces the number of moving parts, usually collapsing the scale of the application database by a large margin, with corresponding reductions in development cost, time, and risk; and it offers substantial improvements in runtime performance as described in our SQL article[17].

## Events

Primary events arise when an actor[18] interacts with the system. These are the events that create value for the business by initiating changes in the value of a business entity.

The specific nature of the event is relatively unimportant because the response is always the same – it is defined by the business policy relative to the existing state of the entity, and is implemented by decision models executing within the business transaction. An event is often accompanied by some

---

[16](http://en.wikipedia.org/wiki/Database_normalization)

[17](http://www.modernanalyst.com/Resources/Articles/tabid/115/ID/3109/The-Role-of-SQL-in-Decision-Centric-Processes.aspx)

[18] A person, another system, or a time based event, as described in the Wikipedia definition of a use case described earlier.
event related data. This data must also be obtained by the business transaction, and will be processed by the transaction (and its decision model) within the context of the existing business entity data.

Provided the event is authorized, the approach is agnostic about the event mechanism. An event can be triggered by request of an authenticated user, email, message, or by any other means whatsoever.

The primary business transaction may generate secondary events, which are (by definition) valid according to the business rules that just created it. The event data for these events is stored as part of the ‘change vector’ that represents the event itself. These secondary events are then processed asynchronously via services as separate business transactions for both simplicity and operational resilience.

It is a simple matter to also place a desired time of execution on secondary events to orchestrate activities across time. In particular, each and every business transaction can generate recursive events that can be used to manage the entire life cycle of the business entity. By storing these future dated ‘bring-up’ events pending a time of execution, we can easily manage workflow on a large scale.

Activities

The definitive activity that gives effect to the state change is policy defined decision making, implemented by decision models within the otherwise generic business transaction.

This ‘core’ decision making might require access to other supporting activities that are accessible within the business transaction’s host application. For instance, let’s assume that our bank in the earlier loan example now has an agreement with a credit agency to provide credit scores. A web service to request and receive the credit score is built and published as an available activity within the host application.

The decision model that is controlling the ‘make a loan’ business transaction is now able to assess the customer and loan data and determine whether a credit score should be purchased. If so, the business transaction’s host application will respond to this decision driven request and acquire the credit score from the agency in real-time. This new data is now added to the transaction’s context data and the credit assessment continues. Ditto for predictive analytics, or any other supplementary data that is required for decision making within the business transaction.

Similarly, the completion of the business transaction may require post approval activities – for instance, generation of loan documents and their delivery to the (possibly waiting) customer. The same business transaction’s decision model can generate a request for these documents, so that the document generation and delivery can also logically occur within the boundaries of the business transaction if required.

The development of activities within the host application is infrastructure development that occurs independently of the business transaction process development; the business transaction simply uses the published activities that the host application offers.

The total number of different activities that the host application needs to sustain is surprisingly limited. Often the same activity is performed in many different ways and this can give the illusion that the number of underlying activities are greater than they really are. In our experience the number may vary between several dozen and a few hundred – either way it is manageable.

The following diagram (Figure 4) shows what the ‘make a loan’ business transaction might look like when it is ‘fully loaded’ with decision models (in this example, the core, secondary, and workflow decision models are executing as a single logical model under the control of a ‘control model’).
The resurgence of the powerful but fundamentally simple business transaction concept is rooted in the value proposition of the business itself. The secret to reaping maximum benefit from the approach is to separately build a) a host application that treats business transactions as content, and b) decision models that define and control those business transactions. Note that a) is a traditionally coded computer application, while b) are logical models that can be developed and owned by business subject matter experts [SMEs] with no programming required (only business knowledge).

The host application performs pre-defined, generic, and relatively utilitarian functions. As we noted in our Decisioning – the Next Generation of Business Rules19 article:

“Service oriented architectures are inexorably trending towards separation of ‘doing’ components from ‘deciding’ components. The ‘doing’ components [can] be commoditized and outsourced. The truly proprietary code, the code that captures business knowledge and IP, is being increasingly concentrated in the ‘deciding’ components.”

The German bank Fidor may be an emergent example of the outsourcing of ‘doing’ components; it employs only 27 people and in addition to running its own operations, it offers a generic platform at low cost for other aspiring banks to plug into.

A traditional application is likely to include many, many individually coded business transactions and activities in one large and complex system. By way of contrast, our proposed service based and transaction agnostic host application is both small and simple. Our experience is that it will represent no more than 20% of the code base of the complete system image, and usually far less. It can therefore be built with much less cost, time, and risk.

And because it remains agnostic about the business transactions within, it provides outstanding agility for the business. Using appropriate tools20, the SMEs can declaratively develop, extend, and maintain business entities and their associated business transactions, for execution within the otherwise agnostic host application. For the sake of clarity, this means that the SME’s can declaratively define new entity and event data – so that the significant constraints that are imposed by a traditional relational data model implementation are alleviated.

It is an important aside that we can describe this host application without reference to a domain; this emphasizes the point that the ‘host application’ design is generic with respect to the transactions it can execute. We are aware of one customer who is contemplating servicing a completely new domain from their existing host application; at this stage, it is plausible that this can be achieved without any new code in the host application.

On the other hand, the host application must deal with many requirements that are specific to each host organisation. It provides the linkage between the business transactions and the underlying, usually proprietary infrastructure, which includes security, user and session management, resource management, execution location, database, file storage, document services, communications, etc. It also provides all integration needs, including the activities described earlier. Each organization has different integration requirements with legacy and support applications, partner systems, industry and regulatory systems, etc.


20 An IDIOM advertorial describing a candidate toolset will be available on Modern Analyst later this month
Design of the Host Application

Given the dramatically reduced scale of development required to build the host application, the significant variations inevitably required by each organisation can usually be efficiently provided using bespoke, native coded development on the platform of choice, albeit in accordance with the generic design pattern described in this article, supported by a portfolio of standard libraries. This allows for rapid development with more or less unlimited platform flexibility going forward; and equally importantly, zero ‘lock-in’ to any external application vendor. For customers who have had difficulty transitioning away from a legacy system, these features alone justify the approach!

For some luckier customers, sometimes a legacy system can be developed over time to become “host-like”, obviating the need to build a host system. The decision centric approach is well suited to enhancing and extending, then hollowing-out and eventually replacing a legacy system, all as a byproduct of normal day-to-day systems development.

To achieve the generic design objective that differentiates the decision centric approach, the design of the host application depends on a few simple but important concepts.

a) The host application must not be aware of the internal structure of the business entities or of any business transaction which processes them.

b) The host application will be aware of specific activity requests and secondary events that might be generated within the primary business transaction. These activity requests and events map to specific integration points that are known to the application – the decision model that is creating them is acting as an unseen controller for the host process.

c) The host application is a meta-application, and is able to manage and understand the meta-data that describes itself.

As noted, the host application is purpose built in two distinct layers:

a) The host application itself, which is inherently bound to a specific technical infrastructure, and which provides a ‘virtual application’ for the business transactions. The host application is business agnostic, and to a high degree is unaware of the business purpose that it serves.

b) The business transaction layer, which is a set of business data, rules, forms, and documents operating under the control of decision models to fully automate the organization’s products and services.

The business transaction layer is deployed as ‘content’ into the more or less domain agnostic host application. Third-party, peer and/or legacy systems are easily integrated at the application level (e.g. using web or message services, even batch updates), with the business transaction layer servicing the data needs of these integrated systems via the respective change vectors. These change vectors become the payload of web or message services or similar, and their contents should not be known to the host application or the services. By definition, they will be understood by the recipient business transaction for whom they are generated and to whom they are addressed.

100% of the proprietary business elements (business data, rules, forms, and documents) in the business transaction layer can be built declaratively using appropriate tools21. These tools should fully support declarative development, versioning, testing, and deployment of these elements. The role of the host application is to provide generic systems capabilities on the chosen infrastructure for the benefit of the business transaction layer. The host application capabilities are technology focused and bind the...

---

21 An IDIOM advertorial describing a candidate toolset will be available on Modern Analyst later this month
otherwise ‘portable’ business transaction layer to a specific technical implementation. And to the extent that the business transaction layer is portable, there need never be another legacy system.

The following list outlines the key responsibilities of the host application:

- Persistence and retrieval, including record ‘search and select’ of the entity records.
- Authentication and authorization, including menus and navigation.
- User session management, possibly including transaction auto-save for each user.
- Integration services, including sending and receiving of messages, the contents of which will be generated by rules in the business transaction layer.
- Management of events, and their processing by business transactions.

The Data

It is desirable that all of the data describing an entity throughout its entire life cycle is contained in a single record, specifically an XML document, which is itself defined by an XML Schema. Some examples of XML based entities that are supported today on systems built on these principles include: insurance policies, loans, insurance claims, public sector entitlements, accounts, funds, contracts, patient episodes, patients (as in lifetime clinical pathway), and itineraries. As we can see, an entity can be many things; and it can be big and complex, extending to hundreds of thousands of nodes with many nested levels of collections in its data hierarchy. This entity XML can be thought of as a relational database for a single entity.

The specific nature of the entity need never be visible to the host application – it can be entirely confined to the business transaction layer – and from this stems much of the power of the decision centric transaction approach.

The above now implicitly defines a generic system that can service virtually any business transaction requirement, provided that the business policies that are relevant to the management of the entities are modelled appropriately, and are used to drive the supporting host application activities via their respective meta-data.

We can now lay out the key design features of the decision centric transaction approach:

- Business policy is developed, defined, and implemented by SME’s using a mix of decision models (rules) and product configuration documents, all of which can be stored in the host application database.
- The entity ‘context data’ that describes the current state of an entity in its entirety is stored extant as XML in an ‘XML data-type’ column in the database.
- External events are passed to the business transaction so that its decision model can process it against the business entity, thereby changing its state.
- XML request/update messages are generated by the decision model to align any and all dependent systems, (e.g. self, legacy, accounting, partner, reporting, etc.) with the latest entity state. These ‘change vectors’ are extracted automatically when saving the entity, and are routed to integration endpoints in this and other systems.
- Workflow is fully managed by the decision model, which creates forward looking actions and events – for instance ‘bring-ups’ (automated future events) and ‘actions’ (current and future events requiring operator intervention). These workflow triggers are generated and captured in the database for each business transaction on the entity – they are updated every time the model
is run, so that the entire life cycle of the entity is managed on a rolling transaction-by-transaction basis.

In order to be agnostic as to the internal structure of the business entities, we have described how we use XML to both define and store the data. Storing the entity data as XML strings in appropriately typed database columns (or even a file system for smaller applications) can result in order-of-magnitude improvements in runtime performance as database load is dramatically reduced.

However, using this xml data approach is sometimes constrained by an existing, fully relational database implementation (i.e. many, many tables already implemented).

Much of the performance improvement that the XML approach delivers can still be claimed by using a simple mapping between these relational tables and their equivalent XML elements. XML allows us to build a complex multi-dimensional structure in memory that can hold collections of dependent database records in a single, schema defined object. This multi-dimensional structure has no equivalent in SQL. Once instantiated, the in-memory XML data object is much, much more efficient than SQL executing repeatedly against a database.

The XML schema and its equivalent runtime objects should mirror the database relations that exist for each business entity, so that the relational integrity of the database sourced data is retained.

The parity of the database and XML images ensures that only ‘simple SQL’ is required – that is, SQL without joins. It is relatively easy for a tool to generate this SQL from a simple mapping configuration, and to then execute a generic, high performance retrieval and conversion of the retrieved SQL results into the XML data structure (and back again if required). This approach removes complexity from the SQL and can improve performance by large margins.

The XML schema is also important because it provides us with a partial ‘context map’ for each and every datum in this structure, a map which can be used to guide and assist the development of the decision models. As described in the SQL article, context is an extremely important concept, with profound implications for business transaction design and performance.

Furthermore, the XML object can be easily and dynamically extended from within the decision models, so that any variables that need to be temporarily stored within the business transaction (perhaps so they can be reused, or while waiting for other values to be calculated) can be simply added to the existing XML object, including (if needed) complex new collections and key combinations that are not supported by the existing database structure.

For instance, payroll and/or billing systems often require aggregating and apportioning values into various artificial time periods (e.g. daily rates, hourly costs, etc.). This usually requires that the time periods be dynamically created and populated within the business transaction, something easily achieved with this design approach.

22 See the ‘Role of SQL’ article for the evidence and reasoning behind this.

23 An IDIOM advertorial describing a candidate toolset will be available on Modern Analyst later this month

24 See the ‘Role of SQL’ article for the evidence and reasoning behind this.
A further advantage of this business transaction oriented, XML based solution is that the business transaction is easily reused: for instance, the same business transaction can be used in an online process and/or it can be bundled into streams for parallel ‘batch’ processing on any scale.

We recognize XML has a poor reputation for memory and processing overhead, but with better than order of magnitude performance improvements actually demonstrated using this design approach (mostly through improved SQL performance), this overhead is easily forgiven. And the ease of use, industry support, and sheer simplicity of the concept make it a favored choice.

Abstraction of Data

The complete abstraction of the data into XML that fully describes the business entity not only mitigates the downside attributable to tight coupling between the database and the application, it introduces some powerful and important new advantages.

1. The calculations and other policy defined logic are now independent of the database and can be developed in an entirely separate and distinct life-cycle by business aligned SME’s. We call this the policy development life cycle.

2. The data and rules can change independently. By extending or swapping the mapping, variations in the data sources can be accommodated. It is even plausible to migrate a system (or version of a system) out from under the abstraction and replace it with a new mapping to a new system. Or multiple existing systems can be mapped to a single new system, leaving all intact.

3. By further expanding the abstracted XML ‘view’ of the data across additional external systems, we can supplement legacy data with other data sources: for instance, in an insurance example, data might be sourced from a doctor’s patient management system and attached to a health insurance application; or customer credit details might be sourced from a credit agency and attached to a loan application. This can expand the scope of the calculations and the utility of the overall system without making any existing legacy system more complex; in fact the existing systems need not know that the data extensions are even occurring.

4. Separation and abstraction of the data into XML means that the system can mix and match data sources by location and by type – database one minute, file system the next (literally). For instance, we can store the XML itself, either in a database, or on the file system, to provide a complete and virtually cost free audit trail for each completed transaction, including all of its supporting decisions and calculations.

5. We can also re-read and re-process the above stored XML, which can lead to the even simpler and more cost-effective XML based data processing cycle described earlier in this section – one read of the XML to obtain all of the data in context! At the very least, we can use this XML for development and regression testing.

6. And with complete XML records available and easily accessible, it is a simple matter to use them for simulations and what-if testing of the business policy and its calculations. This can be integrated into the process of developing the underlying business policy itself, as an active and involved enabler of an agile business.

Simplified Data Model

We have described an application design that ideally includes business entity data stored as XML content within a simplified database structure. For those familiar with data design, the following simplified database schema provides a generic outline of that database, and offers some insight into the underlying design pattern used for the host application. This database design is derived from several
databases actually built and in large scale production use today – as an indication of scale, one such system is onboarding up to 42 million insurance applications per day on a standard PC.

In the adjacent diagram (Figure 5), we can see the business entity data stored as ‘EntityDataXML’ in the Core Business Entity table [blue]. For commercial domains of even modest complexity, this simple design concept can save us from having to build CRUD (Create, Read, Update, Delete) support for perhaps hundreds of database tables; nor are we constrained by the incredible development inertia that is implied by these tables. As a bonus, query performance against this structure is much better than with standard ‘many table’ database schemas because joins are rarely required. Also, some databases support ‘full text’ search on the XML column, allowing Google like search capability over the full extent of the database.

The decision models needed to manage the life-cycle of the entities can be found in the ‘DecisionModelBinary’ column of the Decision Model table [green]; it is plausible to load the decision model dlls or JARs directly from the database.

For data driven rules, various parameters that relate to each particular type or class of entity can also be added, as found in the EntityConfigXML in the Entity Class table [green]. These configuration
documents are automatically supplied to the decision models each time they execute. This feature enables us to manage many dozens of products with a single decision model, so that the business transaction becomes synonymous with a 'universal product engine’.

The Financial Entry, Bring-Up, and Action tables are examples of tables whose rows are generated as change vectors by the rules within the business transaction – these are automatically stripped off by the database handler when the XML is saved, and inserted into their own respective tables under commit control. In this example, the Financial Entry records are updates destined for the General Ledger, and are processed through to the ledger daily.

The Bring-Up and Action tables are used to implement workflow that is generic and controlled by rules:

- The Bring-Ups are processed automatically by the host application on due date, with the Bring-Up record itself supplying the event data. The application will launch a business transaction for each business entity as nominated by the Bring-Up record.
- The Actions are assigned to appropriate roles and seniority by the rules that create them. The host application will automatically present the list of outstanding Actions to users according to their role, seniority, and prior assignments.

**Workflow**

Workflow is inherently supported by the decision centric solution design. Each business transaction’s decision model is able to control all elements of workflow that are relevant to the business entity.

These include:

- **Synchronous Activities** can be requested from the host application by creating nodes in the context XML that define the service required, and which include any meta data that the service needs (parameters, XML content, etc.). The data returned by the activity is simply added to the transaction’s available context data.

- **Asynchronous Activities** are invoked by creating nodes in the context XML that act as events to drive the secondary business transactions. These nodes will also include any required event data (parameters, XML content, etc.). If data is returned, it must be processed as a new event (recursive secondary events of this nature are legitimate).

- **Future Dated Activities** are controlled by generating bring-ups as described above.

- **Manual Activities** are controlled by generating actions as described above.

Each business transaction is responsible for orchestrating its own workflow, with the objective of aligning all systems now and in the future with the current state of its business entity.
CONCLUSION

The benefit of this approach is akin to our traffic lights illustration in the Introduction to this article. Provided that each set of lights is responding to the same traffic flow, they will all align for the common good, even though they are all acting completely independently.

As with the city traffic flow, it is implausible to define, map, and build a coherent ‘über process’ across the entire universe of systems, so we must focus our efforts on smaller units of work acting independently. This article has identified the ‘business transaction’ as this optimal unit of work.

In this approach, provided that each and every business transaction addresses its own process and workflow needs in accordance with the over-arching policy framework of the business and its partners, then the combined effect of all such transactions will align to achieve a coherent and policy compliant ‘über process’.

The arguments presented in this paper describe a powerful new approach that is producing substantial benefits for our customers every day. And it is easily scalable, both from a design perspective and from an operational perspective.

We understand that this approach may seem conceptual, idealized, even hypothetical; whereas the converse is actually the case. Our arguments are made ‘post ipso facto’ – that is to say, we found that it works, and asked ‘how and why?’

First and foremost, this is a practical and proven approach developed on the ‘front lines’ of systems development; the conceptual view described herein is the derivative.

We suggest that you ask yourself, ‘does it make sense?’ If the answer is ‘yes’, then please have a look at the companion paper ‘Taming the IT Beast with IDIOM’25, and see how quickly and easily the concepts can become a practical reality.

Finis

Author’s Bio:

Mark has more than 35 years history in software development, primarily with enterprise scale systems. During the 1980s Mark was actively involved in the development and use of data- and model-driven development approaches that later achieved widespread use. Application of these approaches to the development of an Insurance system for one of the world's largest insurers was formally reviewed in 1993 by the University of Auckland who concluded that "this level of productivity changes the economics of application development."

In 2001 Mark led a small group of private investors to establish Idiom Ltd. He has since guided the development program for the IDIOM products toward even more remarkable changes in "the economics of application development." He has had the opportunity to apply IDIOM's "decision oriented" tools and approaches to projects in Europe, Asia, North America, and Australasia for the benefit of customers in such diverse domains as superannuation, finance, insurance, health, government, telecoms, and logistics.

IDIOM Bio:

Established in 2001, IDIOM Limited is a private company based in Auckland, New Zealand. IDIOM develops and licenses decision-making software that automates business policy on a large scale, making systems more transparent, more agile, and more durable, while reducing development cost, time, and risk.

IDIOM’s innovative business oriented software is used by business users to graphically define, document, and verify corporate decision-making and related business rules; it then auto-generates these into small footprint, non-intrusive software components for use in systems of any type or scale. IDIOM is a pioneer in the development and use of decision automation concepts, and has applied these concepts to develop and automate business policy for customers around the world in local/state/central government, insurance/superannuation/finance, health admin/clinical health, telecoms, logistics, and utilities.

IDIOM automated business policy and decision making extends far beyond mere business rules, so that larger and more complex decision making can be fully delegated to business experts. IDIOM enabled development and management of policy based decision making by policy owners creates a propitious ‘business policy life-cycle’ that significantly improves business agility and transparency.

IDIOM develops and licenses: IDIOM Decision Manager™, IDIOM Forms™, IDIOM Document Generator™, IDIOM Mapper™, IDIOM Tracker™, and IDIOM Decision Manager Workbench™.