GeneXus™

業務知識に基づくシステム開発

(KNOWLEDGE-BASED DEVELOPMENT)

GeneXus の思想と理論上の基礎

平成 19 年 5 月翻訳
平成 26 年 3 月加筆修正

By Breogán Gonda and Nicolás Jodal

May 2007

＜目　次＞

概要：この資料は，**GeneXus** の思想と理論上の基礎を，簡単に，そして系統的に説明したものです。ここに書かれている大半の内容は，**GeneXus** ユーザの間では周知のことですが，これまでは断片的に記載された資料しかありませんでした。

紹介

この資料の目的は，**GeneXus（ジェネクサス）**の本質や将来への開発方向性と同様に，**GeneXus** を実現可能とした基本的な考え方と，理論上の基礎をしっかりと理解して戴くことです。

GeneXus の最初の目的は，確固としたデータモデルを作ることでした。
注意深くこのモデルが作られた後に，モデルを拡張し，ルールや式，画面等といったデータに関するユーザのビュー（業務）を説明し，定義するための要素を加え，業務システムに関する総ての知識を含めることが可能となります。

なぜ業務系システムに関する総ての知識を含めようとするのでしょうか。それは，これがどんな会社の業務システムであれ，完全な自動化によって設計や生成や保守を行うために必要な条件だからです。結果として，他の IT コミュニティがこの道を選択するのは確実であり，我々は出来るだけ早くリードを広げるべきである，という方針が正しいと思えたのです。

しかし，どうすれば業務系システムに関する総ての知識を格納することが出来るのでしょうか。様々な業務システムの異なった状況を，どうすれば格納することができるのでしょうか。

GeneXus の最初のバージョン（100%定義だけ）では，プログラムの 70%を自動生成し，保守できることを目的としていました。これらのプログラムはシステム,によって自動的に生成されていましたが，残りの 30%は手作業でプログラムし，保守しなければなりませんでした。

最初の段階では，問題の 70%を自動的に解決できることは分かっていましたが，保守まで自動的に可能かどうか，疑問を持っていました。**GeneXus** を最初に採用された方々は，開発の生産性向上のみを期待されていました。数ヶ月後に，それらのお客様は開発生産性が著しく向上したことに満足されましたが，むしろ，自動的に保守可能な点に **GeneXus** を採用した利点を見出したのです。

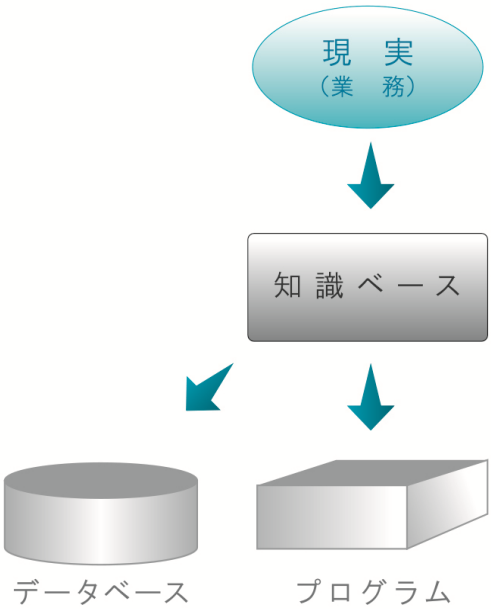
販売を開始して 1 年後に，**GeneXus** プロジェクトはお客様からの強い圧力に直面し始めました。つまり，「自動的な保守を実現した」という成功が，更に，お客様の「総てを自動的に保守できる」という要求を引き出すことになり，そのために「総てを自動的に生成する」ことが必要となったのです。これは世界市場を見渡しても，前例の無い要求でした。そして，何年もの後に，**GeneXus** はこれらの要求を解決できる世界で唯一つの製品となったのです。

では，今日の **GeneXus** の目標は何でしょうか? **GeneXus が目標としているのは，業務システムに関する知識（業務知識）を上手に自動的に管理し，常に業務システムを最善に保つことです。**

それは **GeneXus** の本質的な機能です。一旦，この目的が達成されると，以下の様な幅広い一連の機能を製品化することが可能となります。

- 会社が必要とするデータベースと業務プログラムの自動的な設計，生成，そして，保守
- より短期間に，安いコストで，異なったソースからの複雑な知識の統合

GeneXus の利用者は，知識ベースに蓄えられた（業務の）知識を使って，将来利用可能となる未来の IT 技術を採用したシステムを生成できます。**GeneXus** は，現在の IT 技術に依存しない，適切に格納し管理されている純粋な（業務）知識を使って動作します。



開発はあるガイドラインに従って行われます。そして，この資料は，それらのガイドラインの概念的な説明を簡単に行うことを目的としています。

新パラダイム: プログラム作成の代わりに現実業務を記述

どの様にして，現実の業務を記述するのでしょうか。

"プログラムを書く" 代わりに **"現実の業務を記述"** しようとします。**現実の業務を定義するための記述（What）を最大にし，処理のやり方に関する仕様の記述（How）は最小にしようとしています。**

なぜ，**"プログラムを書く"**代わりに，**"現実の業務を記述"**しようとするのでしょうか。

この目的を実現しようとするとパラダイムの変化を伴い，カルチャーショックも引き起こします。パラダイムの変化を伴う課題は，受け入れられるまでに時間がかかります。しかし，その新しいパラダイムが幾つかの事象で正しいことを理解してもらえれば（疑問を挟む余地が無く），結果として，急速に，幅広く受け入れられていきます。

それらの事象とはどんなことでしょうか。今日，ほとんどのシステムは手作業で開発され保守されています。しかし，過去４０年間（2006 年までのデータ）に市場はどの様に進化してきたのでしょうか。

システムの複雑性は 2,000%も増加しています

プログラム言語の生産性は 150%増加しています

ビジネスとして，この様な状況を受け入れることは出来ませんが，コストは急速に増加しています。

それでも，**時間**と**コスト**という二次元の軸を使って，これらを分類できます。

速くて安価な通信手段が実現したおかげで，会社は低賃金国で技術者を雇用してシステム開発と保守を行うことにより，**コスト節減（コストの軸）**が可能であることに気付き，近年では広く利用されています。

実際には，コストの本質的な問題は何も解決されておらず，代わりに，測定の尺度が変えられただけです。必要な労働時間は基本的に変わらず（同じワークロード），時間当たりの単価は遙かに安くなっています。もし，金額の尺度で評価するなら，劇的にコスト削減できたのです。

時間軸で見るとどうでしょうか。上記の尺度は時間軸には当てはまりません。

業務系システムでは，新たな必要性（新しい装置，新しいユーザ，新しい利用方法，新しい統合の可能性）を記述しなければなりません。ですから，業務システムはますます複雑になり，会社は自らが決めることのできない，市場が要求するタイミングに合わせてサービスを提供するしかありません。結果として，非常に多くの会社は，現在の新しいソリューションによる長期間の開発と保守によって，逆の影響を受けています。今日，特に近い将来は，適切なシステムに依存することなく，素晴らしいビジネスを継続していくことは不可能です。

この様な状況を長い間続けていくことはできません。最初は徐々にでしたが，現在ではより早い

ペースで，多くの事象が出てきています。

つまり，先進国のほとんどの会社は，アウトソーシングのシステム開発に依存する割合を増やしています（これは即ち，低賃金の国にオフショア開発させることを意味しています）。しかし，アウトソーシングを行った経験に満足できずに，より多くの企業が別の方向に向かおうとしています。

より多くの会社が，自社の技術者を解雇し，遠く離れた国に住む外国人に仕事を移すのが困難であることに気付き始めています。この様な状態で仕事を進めるには，アウトソーシングを実施するのとほとんど同じくらいのコストを社内にかける必要があるのです。この様な事を避けるには，もっと高度なレベルの技術を採用しなければなりません。

生産性を劇的に増加させる必要がありますが，プログラミングの生産性はほとんど増加していません（安定期に入っています）。

この問題は，もっと素晴らしいプログラム言語を採用すれば解決するのでしょうか。
この問題はプログラム言語に依存しているのではなく，プログラミングという行為そのものに依存しているのです。

では，どうすれば必要とされる生産性の増加を達成できるのでしょうか。答えはシステム開発の考え方を，プログラミングをベースとするものから，業務知識をベースとするものに変えることです。つまり，**アルゴリズムをプログラムする代わりに，業務を記述すれば良いのです**。

アルテッチ社は，近い将来に業務知識をベースとしたシステム開発の市場が爆発することを確信し，そのための準備も怠っていません。

データモデル

歴史的に見て，IT コミュニティは厳格な物理データモデルを作ろうとするところから始まりました。後に，より柔軟性を高め，恒久的に変化しない記述を多少取り入れるために，他のモデルを加えました。こうしたコミュニティによる最も傑出した成果は，1978 年に **ANSI SPARC[1]**グループが紹介した三層モデルのレポートでしょう。

外部モデル（External Model）： 外部のユーザビュー（業務）が表現されたもの

論理又は概念モデル（Logical or Conceptual Model）： 現実の業務の抽象化によって得られた別のモデル（通常 E-R モデル）

物理モデル（Physical Model）： 基本的にデータベースのスキーマ

このアイデアは，プログラムは外部モデルとだけやり取りをし，論理モデルでこれらのプログラムとデータベース（物理モデル）をマッピングさせ，三つのモデルを並行して開発することでした。こうすれば，プログラムはデータベースから独立したものになります。

このレポートは，当初は賞賛されましたが，後には，もう少しで忘れ去られそうになりました。80 年代前半には，この三層モデルのスキーマを実現しようと試みた業者は一社だけでした。その会

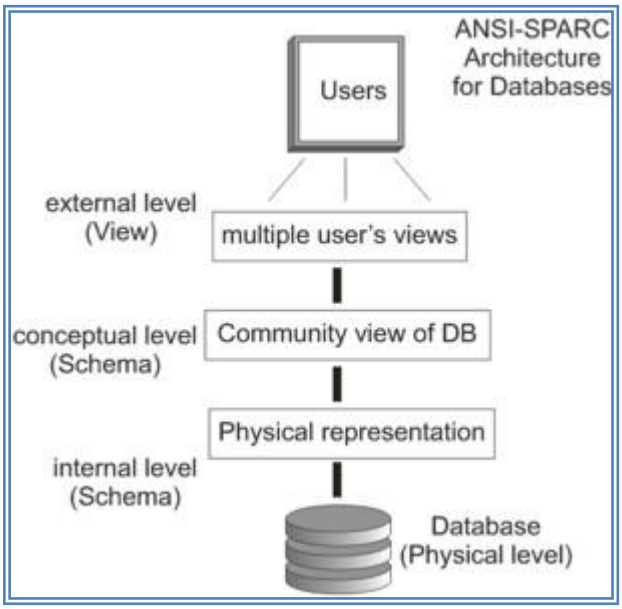
社は Cincom システムで，製品は **SUPRA** と言い，その **SUPRA** データベースにアクセスするのに，専用の言語を用いました。

こんな経緯があっても，ANSI SPARC レポートはまだ生き続けています。今日の状況が以前と違っている点は，近年言われている様な顕著な技術革新の結果として，様々な技術が利用可能となったことです。

現在の技術から見て，モデルには幾つかの種類がありますが，我々の結論は，**ユーザと開発者にとって最も重要なモデルは外部モデルである**ということです。つまり，外部モデルとは純粋な業務知識を集めたものであり，そして，その他の有用なモデルは，総てこの外部モデルから自動的に推論されるに違いない補助的なものなのです。

この論文では，純粋な業務知識を含んでいる**外部モデル**と **GeneXus** の本質と独自性に焦点を合わせています。新たな技術の可能性と新たな顧客の要望に直面したとき，最初にやるべきことは，この外部モデルを拡張することです。

訳者注）ANSI-SPARC Architecture（三層モデル）の説明をインターネットで探すと，以下の図が出てきます。



(http://en.wikipedia.org/wiki/ANSI-SPARC_Architecture)

この絵では，複数のユーザは複数のユーザ・ビュー（業務）を持ち，それを external level（外部レベル）と称しています。そして，複数のユーザのビューを整理すれば，そこから DB のコミュニティ（共同体）としてのビューが得られ，それを conceptual level（概念レベル）と称しています。３層目は Physical representation（物理表現）で，internal level（内部レベル）と称しています。この internal level が物理レベル（Physical level）のデータベースとなります。この論文では，外部モデル，論理又は概念モデル，物理モデルとしています。

言い方を変えれば，外部モデルとは「複数ユーザのビュー（業務）」そのものです。この「複数ユーザのビュー（業務）」を整理し，全ユーザが使える形に正規化されたものが論理又は概念モデルで，一般的には E-R 図等で表現されます。しかし，この論理又は概念モデルでは具体的な DBMS までは考慮されていません。そして，具体的な DBMS を考慮に入れたものが内部モデルとなります。

必要なモデルの機能

GeneXus のオブジェクト・タイプは，将来にわたって進化を続けていきますが，その際の幾つかの原則を下記に説明しましょう。

しかし，その前に一つ質問が有ります。会社の中で，誰が業務知識を持っているのでしょうか。

実際の詳細な業務知識は，業務の遂行にあたって，データを使用し，必要とし，そして，欲しいと思っている総てのレベルのユーザだけが持っているデータのビューです。データ自体には，なんら客観的な業務知識は存在しません。

ですから，**抽象的な議論ではなく，具体的な議論を大切にしなければならない**のです。会社の中には，異なったプロフィールと役割を持った多くのユーザがいます。彼らは抽象概念ではなく，彼らを取り巻く具体的な業務の作業を行っているのです。ですから，業務知識の表現は簡単で，詳細で，そして，具体的でなければならないのです。

これらのビューを集めた外部モデルに頼るのは妥当だと思えます。外部モデルは，それから自動的に推論されるファイル，テーブル，エンティティ，エンティティの関係，インデックスといった，物理的，又は，内部的な要素を一切持つことができません。

GeneXus ではどの様な要素が必要で，どの様な要素が不要なのでしょうか。

下記の特徴を持つモデルを作成しようとしています。

自動的に管理できる業務知識：コンピュータ・ソフトウェアで自動的に管理可能なモデル。つまり，モデルに関する知識（業務知識）は，自動的に理解され処理されるべきです。

一貫性： 常に一貫性を保ったモデル。

オーソゴナリティ（直交独立性：Orthogonality）：モデルを構成するオブジェクトは互いに独立していなければなりません。オブジェクトの追加，変更，削除を行なっても，他のどんなオブジェクトにも影響を及ぼさないのです。

訳者注）外部モデルのオブジェクトを追加，変更，削除しても，他のオブジェクトへの影響を気にする必要は有りません。外部モデルの変更による影響分析は **GeneXus** が自動的に行い、論理モデル、物理モデルにその影響を波及させていきます。

データベースとプログラムの自動設計，生成，そして，保守：データベースとプログラムの自動設計，生成，そして，保守はコンピュータのソフトウェアがモデルを使って自動的に行わねばならない不可欠な作業です。

拡張性：前項はモデルが持っていなければならない定性的な特徴について述べています。しかし，大きな問題を解決するには，問題の大きさに関わりなく，効率的に働くための，モデルの格納方法，アクセス，そして，推論メカニズムを必要とします。

参照の枠組み

どうすれば，正確に知識（業務の知識）を表現できるでしょうか。
現実（業務）を表現し，自動的に理解し，そして，操作可能な，事前に定義された一群のオブジェクト・タイプの作り方に依存します。

通常，業界で行われているやり方は，E-R 図，又は，同様なデータモデルから始めることです。しかし，どうすれば正確なデータモデルを入手できるのでしょうか。企業のデータモデルには，数百や数千ものテーブルが必要です。データモデルは伝統的に，会社の業務について必要な知識を持っている何人かのエンドユーザとやり取りをしながら，モデルの中にエンティティと関係を設定するために，関連するオブジェクトと関係を識別するという，抽象化の過程を通して設計されてきました。

この抽象化のプロセスというのは，非常に主観的なものでした。つまり，会社の中でデータに関する深い知識を持っているのは誰でしょうか。誰が，モデルに入力するデータとなる関連する対象物と関係を，本当に知っているのでしょうか。答えは明確で，しかも，決定的です。つまり，そんな人は誰もいないのです。ですから，様々な人達の異なった業務知識に訴える必要があり，それぞれの担当者の抱える知識を少しずつ追加していくしかないのです。

モデルを構築するのに必要な詳細な知識の元となる対象物はどこに有るのでしょうか。これらのデータに関する知識は，確かに社内には存在しませんでした。しかし，どんなレベルであっても，総てのユーザは，自分が使用し，必要とし，又は，欲しかったデータのビューは良く知っているのです。**誰もデータのことなど知りませんが，普通にこのデータのビューを使って業務を遂行しています。**

これらのデータのビューは，モデルへの基本的な入力として選ばれたものですが，一つの疑問が残ります。つまり，それらのデータをどうやって具体的に記述するのでしょうか。

答えは，他の記述の基礎となる確固とした参照可能な枠組みを作ることでした。

この参照用の枠組みは**項目属性**で構成されています。

基本的な意味（セマンティック）要素：項目属性

各項目属性は，名前や各種の特徴，そして，意味を持っています。

大きな表現力を有するモデルを手に入れるためには，IT のコミュニティでは良く知られている構文上の要素以外に，意味の要素を表現する必要があります。しかし，意味論（セマンティックス）によって自動化を実現するのは困難です。

意味の要素を自動的に管理するためには，構文上の表現を与えなければなりません。

我々のモデルでは，これを表現するために，たった一つの意味の要素だけを選択しました。即ち，

項目属性の**意味**です。

ですから，二番目の基本的な要素は意味であり，モデルでは以下の様に定義されます。

つまり，**意味 （attr） = 名称 （attr）．**

項目属性の名称は，モデルには**不可欠なものです → **モデル内での一貫性を確保するためには，明確なルールが必要です。

このために，**URA（Universal Relational Assumption：普遍的な関係の仮定）**を採用しました。つまり，どこでも同じ項目属性は同じ名前を持つという意味です。そして，同じ名前を共有する違う項目属性（違う意味を持つ）は存在しないのです。他のユーザにとって，名前だけで中身を理解でき，理解し易いということは重要なのです。

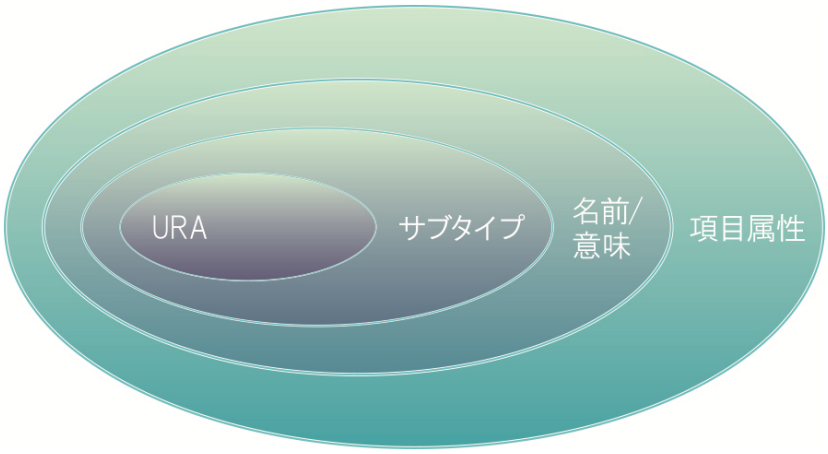
		項目属性								
		顧客番号	顧客名	請求書番号	請求書日付	請求金額	請求明細数量	請求明細価格	製品番号	製品名称
トランザクション	Customers	●	●							
	Invoice	●		●	●	●				
	Invoice Line						●	●	●	
	Product								●	●

ですから，三番目の基本的な要素は URA（Universal Relational Assumption：普遍的な関係の仮定）です。

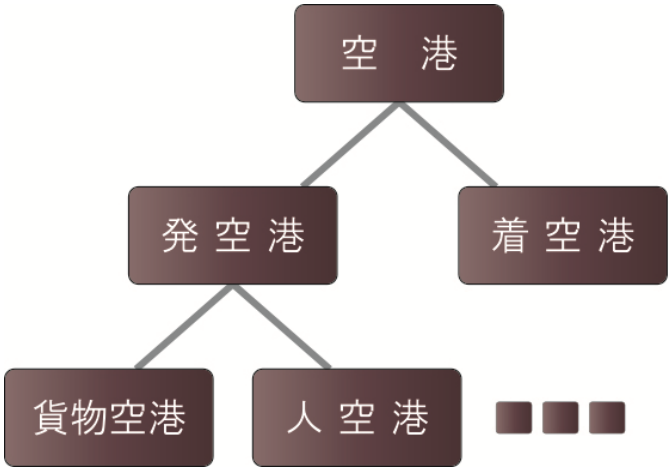
実際には，URA だけでは十分ではなかったので，**項目属性のサブタイプ**を作成し，後には，**項目属性グループのサブタイプ**を作成しました。

四番目の要素はサブタイプによるものです。

まとめ：ここで説明した（**意味**と **URA** と**サブタイプ**を考慮に入れた）**項目属性**は，**GeneXus** 理論の基本的な意味の要素です。**項目属性**は，**GeneXus** のモデルを作る際の，参照用の枠組みを構成しています。



訳者注）前項の絵は分かりにくいので補足しましょう。
例えば，下記の図をイメージして下さい。



業務を分析している過程で，上記の図の様なエンティティが抽出されたとしましょう。実態は総て「空港」ですが，業務によって「発空港」，「着空港」，「貨物空港」，「人空港」と意味が違ってきます。**GeneXus** には業務によって異なる総ての名称を登録します。これが**URA（Universal Relational Assumption）**です。

しかし，業務的な呼び方が異なるだけで，総ては同じ「空港」なので，**サブタイプ**によって同じ項目属性であることを示します。

また、「空港」，「発空港」，「着空港」，「貨物空港」，「人空港」は，**名前で意味**を表しています。この様な**名称**，**サブタイプ**，**意味**を含んだものが**項目属性**です。

ですから，業務分析の結果として **GeneXus** に投入する情報は，URA である「空港」，「発空港」，「着空港」，「貨物空港」，「人空港」です。次に，実態が同じであるものを**サブタイプ**として定義します。

GeneXus は URA である項目属性を使ってプログラムを生成し，**サブタイプ**によって同じ実態である項目属性を考慮し、正規化されたデータベースを生成します。

現実（業務）の説明

参照用の枠組みを紹介した後に，オブジェクト・タイプのインスタンスのセットを通して現実（業務）を表現しましょう。それらは，表現される現実（業務）の特性を集めるのに使われる，項目属性の基盤上に定義されます。

トランザクション（TRANSACTIONS）

総てのユーザは，彼等が日々の業務で使用するデータについて，一つかそれ以上のビュー（業務の見方）を持っています。

それらのビューの中から，データを管理するのに使用する最初のグループを作ることができます（制約されたアクセスによるデータの追加，変更，削除，そして，照会によって）。これらのユーザのビューを**トランザクション**と呼び，**GeneXus** の最初のオブジェクト・タイプになります。

訳者注）例えば，注文伝票，出庫伝票といった業務上のユーザのビューがトランザクションです。

各トランザクションは要素のセットからできています。

トランザクション・データ・ストラクチャ（Transaction data structure）：データはストラクチャの中に記入しますが，これらのストラクチャの情報は正確に集めなければなりません。以下は，通常送り状（Invoice）として良く知られているトランザクションのストラクチャ（構造）です。

```
[INVOICE]
        Invoice_Code *
        Invoice_Date
        Customer_Name
        Customer_Address
                (Product_Code *
                Product_Description
                Invoice_Product_Quantity_Sold
                Invoice_Product_Sales_Price
                Invoice_Line_Amount)
        Invoice_Subtotal
        Invoice_Discount
        Invoice_VAT
        Invoice_Total
```

この表現には，三つの項目属性のグループがあります。つまり，一番上に一度だけ現れるプロローグやヘッダー，何度も繰り返し現れるボディ，最後に一度だけ現れるエピローグやフッターです。このヘッダーでは，***Invoice_Code***（送り状コード）項目属性の横にアスタリスク（*）が付いています。これは，各**送り状**毎に（ヘッダーとフッターの各繰り返し毎に）たった一つの ***Invoice_Code***（送り状コード）が存在することを意味しています。

次に，括弧の間に一群の項目属性（複数の）があります。これは，この一群の項目属性が各送り状（***Invoice***）の中で何度も繰り返し現れることを意味しています。通常，この繰り返しのグループは「ボディ」と呼ばれます。ボディには各送り状（***Invoice***）の行を示す識別子が無ければなりません。この場合，それは製品コード（***Product_Code***）で，アスタリスク(*)を右側に付けることで示されています。その繰り返しのグループの後に，フッターがあります。

このデータ構造図と現在 **GeneXus** で採用されている，よりグラフィックな特長を有するものは，ワーニエ-オール（Warnier-Orr's **[2]** ）やジャクソン（Jackson's **[3]**）の研究成果として紹介されたダイアグラムに基づいています。

ルール（Rules）：多分，トランザクションには以下に示すような複数のルールが存在します。

同じ送り状コード（***Invoice_Code***）を持つ送り状（***Invoices***）は二つと存在しません。

送り状コード（***Invoice_Code***）は相関が有るものとして割り当てられます。

送り状日付（The ***Invoice_Date*** ）のディフォルトは発行日です。

送り状日付（The ***Invoice_Date*** ）は送り状（***Invoice***）の発行日より前であってはなりません。

もし，製品在庫（***Product_Stock***）がマイナスになるなら，送り状（***Invoice***）は拒否されます。

もし，顧客の未払い残高（***Customer_Debit_Balance***）が顧客のクレジット限度額を超えると。送り状（***Invoice***）は拒否されます。

送り状（***Invoice***）を承認することによって，各製品の，製品在庫（***Product_Stock***）＜製品再発注在庫点（***Product_Reorder_Point***）となることが決まると，供給（***Provisioning (Product)***：製品）ルーチンが起動されます。

式（Formulas）：たぶん，トランザクションには以下の示すような複数の式が存在します。

Invoice_Line_Amount = ***Invoice_Product_Quantity_Sold*** * ***Invoice_Product_Sales_Price***

Invoice_Subtotal = sum of ***Invoice_Line_Amount*** of the ***Invoice*** (***Invoice_Code***)

Invoice_Discount = discount calculation function (***Invoice_Code***)

Invoice_VAT = VAT calculation function (***Invoice_Subtotal – Invoice_Discount***)

Invoice_Total = ***Invoice_Subtotal*** – ***Invoice_Discount*** + ***Invoice_VAT***

Customer_Debit_Balance = sum of unpaid invoices (***Customer_Code***)

表示要素（Display elements）：オブジェクトには，Windows や Web 用の画面形式，グラッフィックやダイアログボックスのスタイルといった，表示要素が対応付けられています。

トランザクションは宣言型で，望ましい現実（業務）の記述の大部分は，その知識を獲得することで得られ，知識ベースの中に体系化していきます。

トランザクションとコッドの関係理論に関する考察

（CONSIDERATIONS ABOUT TRANSACTIONS AND CODD'S RELATIONAL THEORY [4] ）

このオブジェクトを表面的に見ると，我々のモデルがコッドの関係モデルに違反している様に見えるかもしれません。コッドの関係モデルに含まれていない新たな要素（式の様な）や，繰り返しグループは明示的に省略されています。

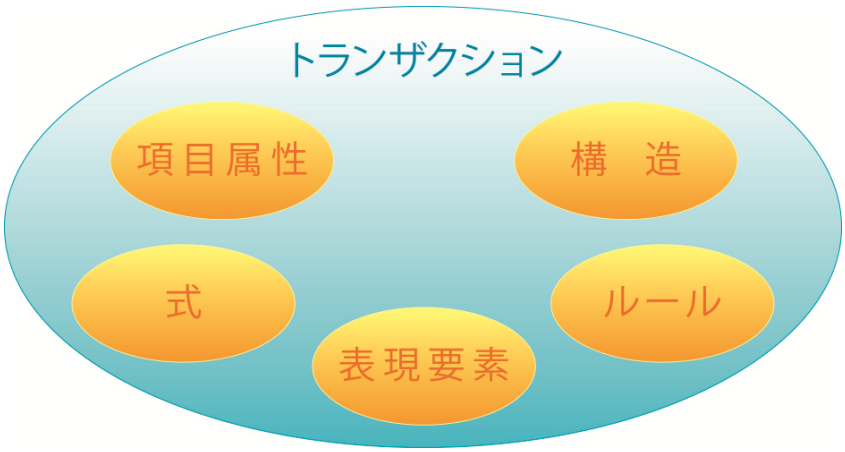
関係モデルと私たちのモデルは，目的が違います。コッドは決して，現実（業務）は関係のセットによって形成されている，とは言わなかったということ（大きな間違いでしょう），は明らかにしておくべきです。彼は，データを格納し操作するのに，関係モデルが非常に良いと言ったのです。

関係モデルは，幾つかの望ましい条件を満たすことによって，データベース内のデータを正確に表現することを目的としています。つまり，冗長性を排除し，データの矛盾の最大の源を避け，適切にデータを管理するための演算子のセットを含めた，小さなルールのセットを導入します，

外部モデルは業務知識を獲得し格納するのに使われます。このモデルは，最も直接的で，且つ，客観的な方法で，現実を表現可能とすることを目的としています。この目的のために，ユーザのビューを集め，モデルにそれらを格納します。次に，推論の能力を最大にするために，その中に獲得された総ての知識を格納し体系化します。

この文脈の中では，関係モデル（又は，むしろモデルのスーパーセット）は，データを表現し，そして／又は，管理するのに使われます。これは，以前述べた ANSI SPARC レポートの中で参照された，いわば内部又は物理モデルに相当します。

我々は理論の中で，関係モデルを幅広く利用しています。



プロシジャー（PROCEDURES）

これまで見てきた様に，トランザクションにはユーザのデータビューを宣言的に記述でき，ユーザが自分のデータビューを管理するのに必要なデータベースやプログラムを自動的に設計し，生成し，保守するために，必要な知識をしまっておくことができます。

しかし，これだけでは十分ではありません。そして，バッチ処理か同様の処理が必要となります。これらの処理は，データビューによって得られた知識からだけでは推論できません。

GeneXus の最初のバージョンがリリースされた時には，トランザクションは対応するプログラムの生成により，課題の約 70% を解決しただけであり，プログラムの 30% は欠落していました。

そして，顧客の要望を 100%満たすために，手続き上の記述（つまり，高いレベルの手続き型言語）が出来るオブジェクトを採用することに決めました。

この言語はどんな特徴を持つべきでしょうか。市場の中で最も良い手続き型言語が検討され，特徴が分析されました。よく知られた **For Each...End For** の様な，単純で，高いレベルのやり方でレコードのセットを使って処理することの出来る，興味深い構造も見つけました。

しかしながら，研究の対象とした総ての言語には，**テーブルといった低いレベルの物理的な要素を参照する必要が有る**，という**致命的な欠陥**を持っていました。ですから，プログラムを書くとき，開発者は，各項目属性をどのテーブルから取ってくるのか，知っていなければならないのです。

もし項目属性が別のテーブルに移ったら，せっかく書いた記述は使えなくなってしまいますが，GeneXus では決して容認することはできませんでした。

別の言い方をすれば，外部モデルにテーブルは含まれていません。ですから，テーブルを含むいかなる仕様もオーソゴナル（訳者注：直交独立）であるとは言えないでしょう。これが，アプリケーションの自動的な保守をできなくする理由です。

解決策は，**最高の言語の最も進歩した構造による構文を用い，その引数が常に外部モデルの一部である様な言語を設計すること**でした。

例えば，テーブルの代わりに，振る舞いを実行するのに必要な項目属性のセットについて話しましょう。自動生成時に，どのテーブルが含まれていて，もし使えるのなら，どのインデックスが，どの様に使われるかを，システムは決定します。これら総てを使って，プログラムを生成します。

言語の構文について，ここでは詳しく扱いません。それは，データベースの構造が変化した後でも有効であり続ける様な，データを管理するための命令を持ち，高度な手続き型言語には通常備わっている，論理演算や算術演算の命令を持つ手続き型言語です。

この方法で，一貫性とオーソゴナリティ（直交独立性）に関する **GeneXus** のガイドラインに従って発見された総てのケースは解決されました。

このオブジェクトの長所と短所とは何でしょうか。..

大きな長所：手続き型の能力は，トランザクションの宣言型の能力を補完します。そして，（他の **GeneXus** オブジェクトでも見られる様に）どんな業務系システムの問題も確実に解決します。

弱点：手続き型の記述を書く作業は，トランザクションのように宣言型の記述を書くほど生産性は高くありません。

この状況を解決するために，様々な要素が追加されきましたが，近い将来追加される要素は手続き的な記述の必要性を減少させるでしょう。

最も重要なオブジェクト・タイプは**トランザクション（Transactions）**と**プロシジャー（Procedures）**です。それらを使って現実（業務）の正しい記述はできますが，時の経過とともに，新たなオブジェクトの重要なセットが追加されてきます。例えば，それらのいくつかを使えば，記述を完成させられます（**GXFlow** は，ビジネスの処理ワークフローを記述し，特定の管理サーバーを利用するための必要なコードを生成します）。また，新たなアーキテクチャーがサポートするオブジェクト（**Work Panels, Web Panels, GXquery**）として，よりユーザフレンドリーなダイアログボックスが可能となります。更に，他のオブジェクト（**Business Components, Data Providers, Patterns**）は，顕著な生産性の向上を実現するために，知識を簡単に再利用できます。そして，エンドユーザ自身で，データベースやデータウェアハウス（**GXquery**）からデータを検索できるオブジェクトもあります。最も複雑な要求を満たすために，これらのオブジェクトを組み合わせて使うこともできます。

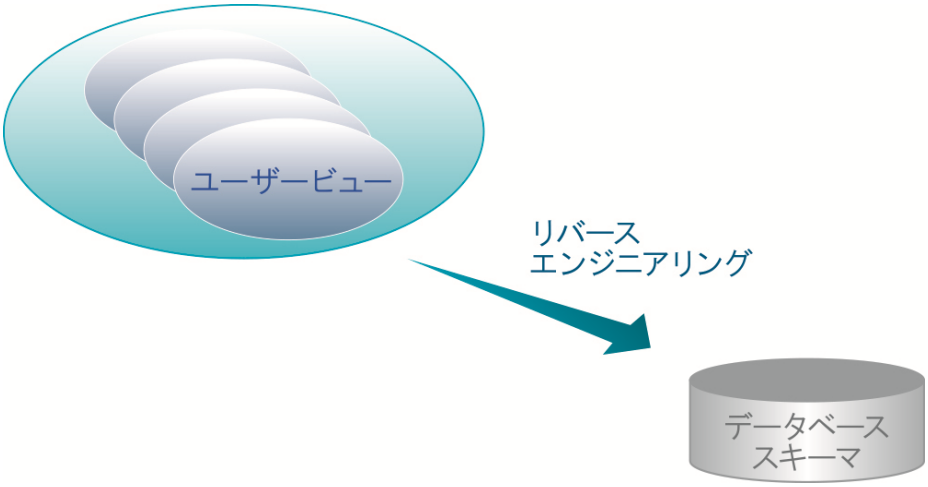
GeneXus のオブジェクト・タイプのセットは完成したのでしょうか。理論上それを示すことは出来ません。実務上は，既に立証していますが。一般的な IT コミュニティが実際に公表しているところでは，世界中で４万人以上（訳者注：現在は１０万人以上）の **GeneXus** 技術者が，日々 **GeneXus** のみを使ってミッションクリティカルなシステムを開発しています。また，**GeneXus** が今日の業務系システムのニーズを完全に満たしていることも示されています。

しかし，**GeneXus** のオブジェクト・タイプは閉ざされた固定的なものではありません。将来，新たな機能が多分必要となるでしょうが，その時は，それらのニーズを満たすために新たなオブジェクトのタイプが導入されることになるでしょう。

知識ベース構築における関係モデルの重要性

与えられた一群のデータビューにとって，唯一の関係モデルが存在すると信じるのは合理的です。ここでそれを立証するつもりはありませんが，もしここで述べることが正しければ，最小の関係データベースのスキーマを導き出すために，一群のデータビューを使ったリバースエンジニアリングの処理方法を見つけ出すことができます。

このリバースエンジニアリングの処理方法を獲得したことが，**GeneXus** プロジェクトの最初の研究でのブレークスルーでした。そして，その処理方法に基づき，必要な知識ベースを構築することに成功しましたが，しかも，その作業はまだ継続しています。



訳者注）これは GeneXus の基礎についての説明です。
データビューを満足させる一つの最小な関係データベースのスキーマが存在している一群のデータビューを考えます。与えられた一群のデータビューから関係データベースのスキーマを推論する方法が必要で、また一方では、関係データベースのスキーマからそれらのデータビューを導出できなければなりません。この推論過程のことを、ここではリバースエンジニアリングと称しています。

外部モデルと知識ベース

当初，知識ベース（**Knowledge Bases**）は一群の推論メカニズムに関連付けられ，どの様な特定のアプリケーションからも独立した一般的なルール**[5]**を含んでいました。ユーザの現実（業務）をオブジェクトに記述する時，記述内容は外部モデル（**External Model**）に蓄えられます。システムは，業務知識を知識ベース（**Knowledge Bases**）に加えるために，自動的に外部モデル（**External Model**）に含まれている総ての知識を獲得し体系化します。さらに，先に獲得した知識の上に，システムは今後の推論をより効率的にするための手助けとなる一群の結果を論理的に推論します。

GeneXus は常に知識ベースからやり直します。知識ベースは外部モデルから成り立っているので，知識ベース内の総ての知識は外部モデル（知識ベースの部分集合である）の知識と同じで，外部モデルから独立したルールや推論メカニズム，それに他の要素のセットは知識ベースから推論されます。

開発者は，ユーザの現実（業務）のオブジェクトを変更することによって，外部モデルを変えることができます。それらの変更は，自動的に変更を加えなければならなくなる総ての要素，例えば，知識ベースの他の要素やデータベース，そして，アプリケーションプログラムに波及していきます。同様に，開発者は外部モデルに属していないどんな要素も直接変更することはできません。

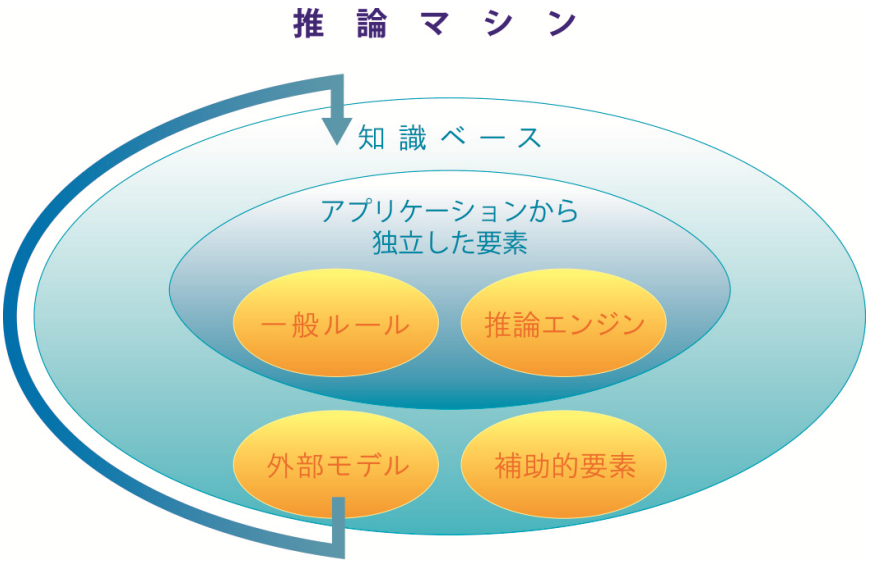
この作業は**外部モデル**（**What**）のみに焦点を合わせます。この作業では，外部モデルを含み，維持し，How を構成し，これらの推論の効率をより良くするのに使われる中間結果と，推論過程の

基礎であり数学的で論理的なツールの複雑なセットである知識ベースを扱うことはしません。**How** に関する知識は，**GeneXus** を旨く使いこなすためには必要ではありません。

What と How は，どちらがより重要なのでしょうか。どちらか一方でも無ければ動きませんが，もし，What が正しくなければ，総てが間違いとなってしまいます。

以上のことから，非常に重要なことが予想できます。それは，**GeneXus** の現在の実装方法を越えることができる，ということです。つまり，**総ての**知識は外部モデルに含まれています。ですから，明日実現するかもしれない全く違ったやり方でさえ**知識ベース**をサポート可能で，顧客の知識をなんら問題無しに再利用できる，ということです。

要するに：知識ベースと関連する推論メカニズムは，大きな『推論マシン』を構成することになります。この良い例は，与えられたデータビューから，それを管理するのに必要なプログラムを自動的に推論することができる，というものです。



D-DAY とプロトタイピング

プロトタイピングで使えるというのは，素晴らしいパワーです。そして，適切に使用されると，最終テストや実装環境への装備を大袈裟に行なうことも避けられます。(総ての事象が起こる，マーフィーの法則が特に適用される D-Day の様に)

訳者注）D-Day とは第二次世界大戦のノルマンディー上陸作戦決行日のこと。

GeneXus の特徴は，何時でも総てをテストでき，総てのオブジェクトを最も便利な時にプロトタイプできることです。このパワーは非常に有効で，**GeneXus** 理論による直接的な成果です。特に，自動的に変更を反映させる機能は，**GeneXus** だけが持っている重要な特徴です。

GENEXUS の基本的な概要

業務知識の自動管理：それは現実のオブジェクトのモデル，つまり，コンピュータのソフトウェア

によって自動的に管理できる**外部モデル**に基づいています。

特に，以下を提供しています。

一貫性： このモデルには常に一貫性があります。

オーソゴナリティ（直交独立性）：外部モデルのオブジェクトは相互に独立しています。
オブジェクトの追加，更新，又は，削除は他のオブジェクトに何ら変更を及ぼしません。

データベースとプログラムの自動的な生成と保守。

外部モデル：本質的な業務の知識は，外部モデルから自動的に推論されるファイル，テーブル，実体（Entity），実体の関係，インデックス，又は，その他の物理的，又は，内部的な要素を含むことのできない外部モデルに対応しています。

統合：GeneXus には，概念的に統合された独自のオブジェクト群があります。そして，次の Rocha バージョンでは，新たな統合の段階として，アルテッチ社や他のサードパーティの開発した他のモジュールや製品を統合する機能を提供するでしょう。

訳者注）Rocha バージョンは **GeneXus X** として，既に，2009 年に提供されています。現在の最新バージョンは **GeneXus X Evolution2** です。

GENEXUS のトレンド

GeneXus は絶えず発展を続けており，４つの尺度を使って進化のトレンドを表すことができます。つまり，**完全性**（**COMPLETENESS**），**生産性**（**PRODUCTIVITY**），**普遍性**（**UNIVERSALITY**），そして，**使いやすさ**（**USABILITY**）です。

完全性（COMPLETENESS）： GeneXus によって自動的に生成され保守されるユーザシステムのコードの割合を測定します。
1992 年以来，**GeneXus** の完全性は常に 100%です。これは変更を自動的に反映させることで，開発と保守に関わるコストを劇的に減らせる，という基本的な ARTech 社の約束です。

普遍性（UNIVERSALITY）：利用可能な最も重要なプラットホームのカバー範囲を測定します。
現在，**GeneXus** は総ての利用可能なプラットホーム（既に，広く利用され，イントール数が増加しているものを基準とした）をサポートしています。
この様に，お客様は **GeneXus** で開発されたアプリケーションを，サポート対象である別のプラットホームに安価に移植できるので，**GeneXus** はお客様に素晴らしい選択の自由を提供していることになります。

生産性（PRODUCTIVITY）：GeneXus で開発した場合の生産性の向上を，マニュアルでプログラムを作成した場合と比べて測定します。
長年に亘り，**GeneXus** は開発生産性向上の目標を，十分と考えられた 500%という値に置いていました。
今日，顧客は新しいディバイスと新たな必要性のために，ますます複雑なシステムを必要として

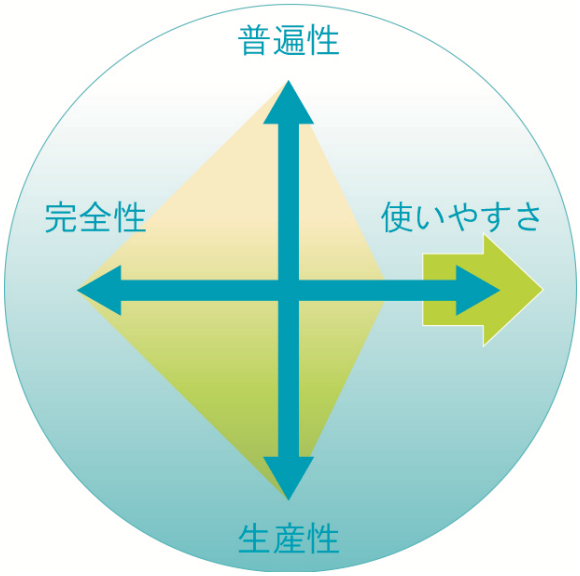
います。特に，一般的に言って，訓練することのできないエンドユーザに優しく使ってもらうためには，複雑な作りになったとしても，ニーズに合致したシステムのためには仕方ありません。その上，マーケットに出すまでの時間を短縮するために，より短時間でそれらのシステムを絶えず開発しなければなりません。
2004 年に，バージョン 9.0 が出るとこれらの目標を 1,000%に変更しました。そして，Rocha というコード名が付けられた新しいバージョン（訳者注：**GeneXus X**）では，2,000%に変更しました。

使いやすさ（USABILITY）：一般的な言葉として，使いやすさを測定します。技術系のユーザのみならず，非技術系のユーザにも手が届き，**GeneXus** を利用して戴ける様に努めます。

GeneXus で開発する人は，指定されたプラットホームに関する専門技術を持たなくても，そのプラットホーム上で稼働する素晴らしいアプリケーションを開発できるので，**GeneXus** には基本的なレベルでの重要な使いやすさ（Usability）が有るということは暗黙の内に了解されています。

しかしながら，我々が考えていることは，それほど技術的な前提条件を付けずに，より多くのユーザが GeneXus を利用できる様にする，という限界に挑戦することです。

Rocha バージョン（訳者注：**GeneXus X**）では使いやすさが大幅に改善されますし，この傾向は将来のバージョンでも続きます。



その次は？　我々のモデルは固定的なものでしょうか。

絶対に違います。つまり，原則とは永久的なもので，経験によって証明されるものです。しかし，最も簡単なやり方で，現実（業務）に現れる新たな事象を記述できる様に，常に厳しい目で **GeneXus** を眺め，機能を拡充していくべきだと考えています。

.

参考文献

[1] ANSI-SPARC: Final report of the ANSI/X3/SPARC DBS-SG relational database task group (portal ACM, citation id=984555.1108830)

[2] WARNIER-ORR: Warnier, J.D. Logical Construction of Programs. Van Nostrand Reinhold, N.Y., 1974; **Orr, K.T. *Structured Systems Analysis***. Englewood Cliffs, NJ, 1977: Yourdon Press; Kenneth T. Orr, Structured Systems Development, Prentice Hall PTR, Upper Saddle River, NJ, 1986.

[3] JACKSON: Jackson, M.A. *Principles of Program Design*. London: Academic Press, 1975.

[4] CODD: E. F. Codd, A relational model of data for large shared data banks, Communications of the ACM, v.13 n.6, p.377-387, June 1970.

[5] RULES: ある特定の会社のモデルに対応したルールということではなく，どんなアプリケーションからも独立していて，常に正しいとされる一般的なルールは存在します。例えば，

一貫性の必要性：詳細を見なくても，あらゆる科学において言える様に，一貫性がルールの主な源泉であると言えます。つまり，我々は現実（業務）は一貫していると仮定しています。ですから，現実（業務）のどの様な表現も一貫しているべきでしょう。

この原則に基づいて，我々のモデルを豊かにする多くの一般的なルールを推測できます。

一つの単純だけれども重要なケースは，参照整合性（**referential integrity**）のケースです。

（訳責：ジェネクサス・ジャパン株式会社　大脇文雄）

（謝辞：この論文の翻訳に際し、ご支援を頂戴した黒川利明氏、内山東平氏、四居雅章氏に深く感謝致します。）

GeneXus™

ジェネクサス・ジャパン株式会社
〒141-0031
東京都品川区西五反田 2 丁目 27 番 3 号　五反田フロント
Tel. (03)6303-9381　　Fax. (03)6303-9980