KNOWLEDGE-BASED DEVELOPMENT

PHILOSOPHY AND THEORETICAL FOUNDATION OF GENEXUS

By Breogán Gonda and Nicolás Jodal

May 2007

SUMMARY: THIS IS A SIMPLE, SYSTEMATIC OUTLINE OF THE PHILOSOPHY AND THEORETICAL FOUNDATION OF GENEXUS. EVEN THOUGH MOST OF THESE ELEMENTS ARE WELL-KNOWN IN THE GENEXUS COMMUNITY, THEY ARE ONLY PARTIALLY DOCUMENTED.

INTRODUCTION

The purpose of this paper is to help better understand the basic ideas and theoretical foundation that made GeneXus possible, as well as its essence and future developments.

The first objective of GeneXus was to create a robust data model. Only after this model had been carefully built was it possible to consider expanding it and adding declarative elements such as rules, formulas, screens, etc., trying to describe users' views of data and thus include all the knowledge in business systems.

Why try to encompass all the knowledge in business systems? Because it is a necessary condition to design, generate and maintain any company's business computer systems in a completely automated fashion. It seemed obvious that the IT community would eventually choose this path and that we should take the lead as soon as possible.

But, how could we encompass all the knowledge in business systems? How could we include the many different cases of business systems?

The first GeneXus version (100% declarative) was aimed at generating and maintaining 70% of the programs. These programs were handled automatically by the system, while the remaining 30% had to be manually programmed and maintained.

Prospects understood that automatically solving 70% of their problem was very good, but they doubted that automatic maintenance would be possible. The first ones to adopt GeneXus did so only expecting to increase development productivity. After a few months, those customers were able to appreciate a large increase in development productivity, but much more so the advantages of automatic maintenance.

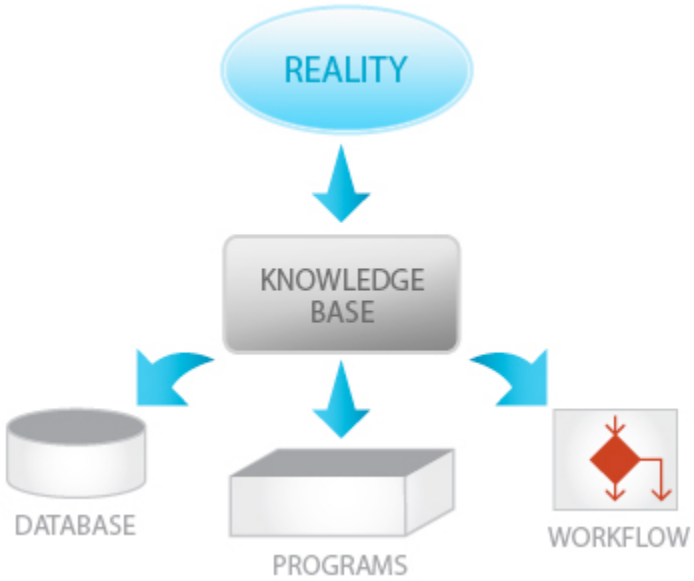
One year after launch, the GeneXus project started facing great pressure from customers. The success of automatic maintenance and its advantages drove them to demand "automatic maintenance of everything," which required "automatic generation of everything." This was unprecedented in the global market, and many years later, GeneXus still is the only product in the world to solve these problems.

So, what is GeneXus' objective today? **The objective of GeneXus is to achieve very good automated management of business systems knowledge**.

That's the essence of GeneXus. Once this objective is achieved, a wide set of sub products are possible, such as:

- Automatic design, generation and maintenance of the database and application programs needed by a company.
- Generation, based on the same knowledge, for multiple platforms.
- Integration of knowledge from different sources to address complex needs at lower costs in terms of time and money.

Generation for future technologies when they become available, from the knowledge currently stored by GeneXus customers in their knowledge bases. GeneXus works with pure knowledge, which remains relevant regardless of the then current technologies.



The entire development has followed certain guidelines, and this document aims at providing a simple, conceptual explanation of these guidelines.

NEW PARADIGM: DESCRIBING INSTEAD OF PROGRAMMING

How do we intend to describe reality?

We intend to "describe**" instead of "**programming**." We intend to maximize declarative descriptions and minimize procedural specifications.**

Why describing **instead of** programming**?**

This objective constitutes a paradigm change and implies a culture shock. The problem with paradigm changes is their slow adoption, but if they are correct (and this is undoubtedly so!) at some point events occur that bring about their fast, widespread adoption.

What are those events? Today, most systems are manually developed and maintained, but how has the market evolved in the last 40 years (data up to 2006)?

> **System complexity has increased by 2000%**
>
> **Programming language productivity has increased by 150%**

This situation is unsustainable for businesses; costs are skyrocketing.

We can, nevertheless, group these costs around two dimensions: **Time** and **Money**

Thanks to the availability of fast, cheap communications, companies have discovered that they can act on the **Money** dimension by hiring system development and maintenance in low-wage countries, and they have done it extensively in recent years.

Actually, the cost problem hasn't been solved; instead, the unit of measurement has been changed. The number of work hours is basically the same (it's the same workload) but the price per hour is much lower. If we measure it in money, there was a dramatic cost reduction.

What about the **Time** dimension? The above doesn't apply to time.

Business systems have to address new needs: new devices, new users, new usage methods, new integration possibilities. That's why they grow increasingly complex and companies are subject to an ever-critical time-to-market that they can't change. As a result, more companies are being adversely affected by long development and maintenance times of new and current solutions. Today, and especially in the near future, it's impossible to do good business unless you rely on adequate systems.

This situation can't go on like this for long. Slowly at first and currently at a faster pace, more examples emerge:

Companies in most developed countries are increasingly resorting to outsourcing system development –generally accompanied by offshoring to low-wage countries– but at the same time, a growing number of companies are taking the opposite direction because of unsatisfactory outsourcing experiences.

More and more companies find it very difficult to fire their own technicians and transfer their jobs to foreign workers in distant countries. In order to avoid that, they need their in-house costs to be similar to those of outsourcing. Only the adoption of high-level technologies will make it possible.

A dramatic increase in productivity is necessary, but the productivity of programming languages has long reached a plateau.

Can this problem be solved through better programming languages? The facts show it cannot. The problem doesn't lie in programming languages but in programming itself.

So, how can we achieve the necessary increase in productivity? With knowledge-based, not programming-based, development: the solution is to **describe instead of programming**!

At Artech we're sure of that and we're getting ready for the market explosion of knowledge-based development that will take place in the coming years.

DATA MODEL

Historically, the IT community started working with one rigid physical model. Later on it added other models in an attempt to obtain more flexibility and some degree of description permanence. The most outstanding work on the subject is the 1978 report of the **ANSI SPARC [1]** group, which introduces three models:

External Model: where external views are represented.

Logical or Conceptual Model: it's another model –usually an E-R model– to be obtained by abstraction from reality.

Physical Model: basically, it was the database schema.

The idea was to develop all three models in parallel so that programs only interacted with the External Model and, from the Logical Model, map those programs and the database (Physical Model). In this way, programs would be independent from the database.

The report was praised at the time and later almost forgotten. In the early 80s, only one manufacturer tried to implement this three-model schema: Cincom Systems, for their product **SUPRA** and for different languages accessing **SUPRA** databases.

Beyond all this, the ANSI SPARC report is still valid. The main difference today is the available technology resulting from the significant technological changes that have occurred in recent years.

In view of current technology, we conclude that there may be several models, but **the most important model for users and developers is the External Model**: it gathers external knowledge and everything else –such as auxiliary models that may be helpful– must be automatically inferred from this External Model.

In this paper we focus on the **External Model**, which contains genuine knowledge and the essence and uniqueness of GeneXus (the **What**). In the face of new technology possibilities and new customer needs, the first thing to do is expand this External Model.

NECESSARY MODEL FEATURES

In the future, GeneXus object types will evolve, but following certain principles explained here.

However, there is one previous question: who has the knowledge in companies?

The only objective and detailed knowledge is that of users of all levels about views of the data they use, need or want. There isn't any objective knowledge about data itself.

That's why we **must favor the concrete over the abstract**. There will be many users with different profiles and roles within the company, and they deal better with concrete elements of their surroundings than with abstract concepts. That's why the representation of knowledge should be simple, detailed and objective.

It seems appropriate to rely on an External Model that gathers these views. An External Model can't have any physical or internal elements such as files, tables, entities, entity relationships, indexes or anything else that may be automatically inferred from it.

What elements are desirable and what elements are unacceptable in GeneXus?

We're trying to create a model with the following features:

Knowledge that can be automatically managed. A model that is automatically manageable by computer software. That is to say, our model's knowledge should be automatically understandable and operable.

Consistency. A model that is always consistent.

Orthogonality. The objects that constitute a model have to be independent from each other. Adding, modifying or deleting an object won't require changing any other object.

Automatic design, generation and maintenance of the database and programs. Automatic design, generation and maintenance of the database and programs are essential tasks that computer software can do automatically from a model.

Scalability. The previous items describe the qualitative features that our model should have. But since we intend to solve big problems, we need the model's storage, access and inference mechanisms to work efficiently regardless of the problem size.

REFERENCE FRAMEWORK

How can we accurately represent knowledge? By creating a set of predefined object types that enable representation of reality and are automatically understandable and operable.

The path usually followed by the industry is to start from an E-R or similar data model. But, how can we obtain an accurate data model? A corporate data model needs hundreds or even thousands of tables. Traditionally, data models were designed through an abstraction process, which implies communicating with several end users who have the necessary knowledge about the company, and identifying their relevant objects and relationships in order to include the corresponding entities and relationships in the model.

This abstraction process was highly subjective: who in the company had a deep knowledge of the data? Who really knew the relevant objects and relationships that were the data input for the model? The answer is clear and conclusive: nobody. So it was necessary to resort to the knowledge of different people, and each one added their own subjectivity.

Where was the objective, detailed knowledge necessary to build our model? This knowledge about data certainly didn't exist in the company. However, all users of any level knew very well the data views that they used, needed, or wanted: **Nobody knows the data but everybody usually works with views of this data**.

These data views were chosen as basic input for the model but one question remained: how to describe them objectively.

The answer was to create a solid reference framework to base the other descriptions.

This reference framework is constituted by **Attributes**.

Basic Semantic Element: Attribute

Each attribute has a name, a feature set and a meaning.

In order to obtain a model of great expressive power, we need to represent semantic elements besides the syntactic elements which are well-known to the IT community. But it's difficult to work automatically with semantics.

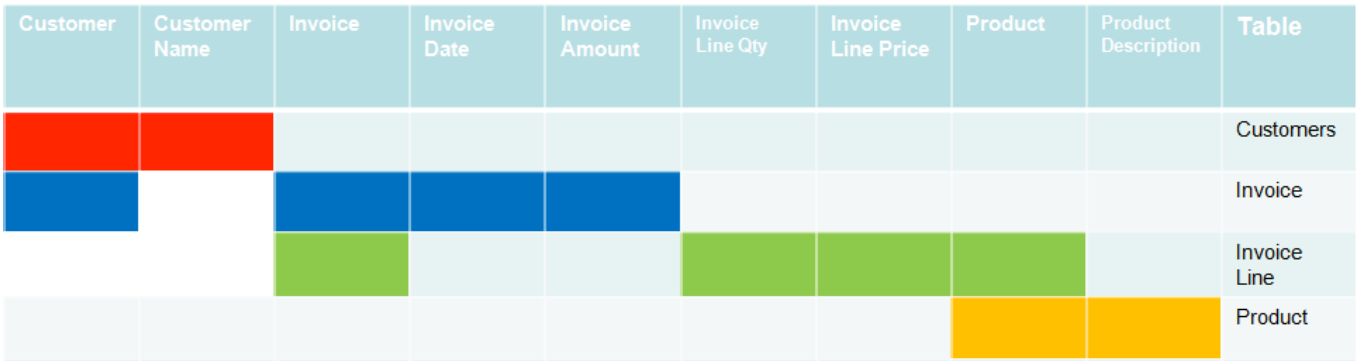
To automatically manage a semantic element, it has to be given a syntactic representation.

Only one semantic element was chosen to be represented in our model: Attribute **meaning**.

So, the second basic element is Meaning, and in our model it's defined as follows: **meaning (attr) = name (attr).**

Attribute names become essential to the model → Clear rules are needed to assign them so that consistency is ensured throughout the model.

To that end, we adopted the **URA (Universal Relational Assumption)**. It means that an attribute will have the same name everywhere and there won't be two different attributes (with different meanings) that share the same name. It's important that names are self-explanatory and easy to understand by other users.

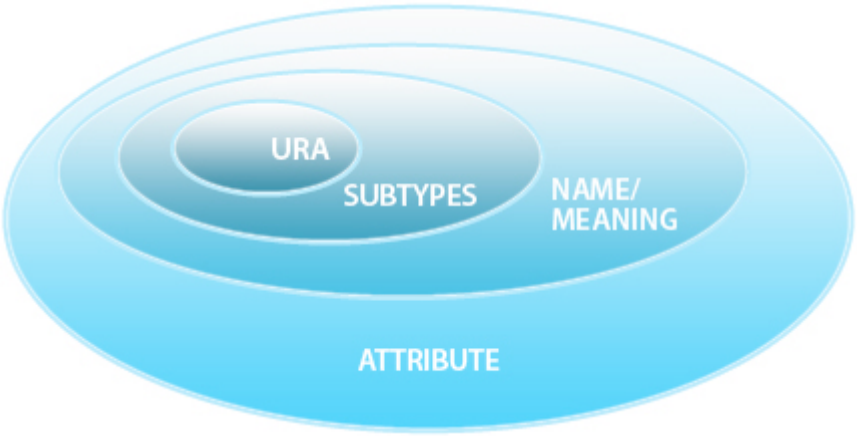


So, the third basic element is the URA (Universal Relational Assumption).

Because reality has shown that the URA isn't enough, we've created **attribute subtypes**, and later on, **attribute group subtypes**.

The fourth element is constituted by subtypes.

IN SUM: Attributes as described here (considering **meaning**, **URA**, and **subtypes**), are the basic semantic element of the GeneXus theory. **Attributes** constitute the reference framework on which we build the GeneXus models.



DESCRIPTION OF REALITY

After introducing the reference framework, we'll describe reality through a set of object type instances. They will be defined on the basis of their attributes, which will be used to gather characteristics of the reality to be represented.

TRANSACTIONS

Every user has one or more views of the data he/she uses daily.

Among these views, we can make a first group with those used to manage data (with restricted access to add, modify, delete and view it). These users' views are called **Transactions** and are the first GeneXus object type.

Each transaction has a set of elements:

Transaction data structure: Data is presented within a structure and we have to accurately gather these structures. Below is the structure of a well-known transaction, a regular invoice.

[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

This representation has three attribute groups: A prolog or header that appears once at the top, a body that appears several times, and an epilogue or footer that appears only once. In this header, next to the ***Invoice_Code*** attribute is an asterisk (*), which means that for each **Invoice** (each occurrence of the header and footer) there is only one ***Invoice_Code***.

Next, there is a set of attributes between brackets, meaning that this attribute set can occur several times within each ***Invoice***; this repetitive group is usually called "body". The body should have an identifier which clearly marks each ***Invoice*** line. In this case it's ***Product_Code*** and this condition is indicated with an asterisk (*) on the right side. After the repetitive group there's a footer.

This Data Structure Diagram and the one currently used by GeneXus, which has more graphic features, are based on the diagrams introduced by Warnier-Orr's **[2]** and Jackson's **[3]** works.

Rules: Most likely, there will be a set of rules for the transaction, as shown below:

There won't be two ***Invoices*** with the same ***Invoice_Code***.

The ***Invoice_Code*** will be correlatively assigned.

The ***Invoice_Date*** will default to the date of issuance.

The ***Invoice_Date*** can't be earlier than the ***Invoice*** date of issuance.

An ***Invoice*** will be rejected if any ***Product_Stock*** is negative.

An ***Invoice*** will be rejected if the ***Customer_Debit_Balance*** exceeds the customer's credit limit.

When the acceptance of an ***Invoice*** determines that the ***Product_Stock*** < ***Product_Reorder_Point*** for any product, the ***Provisioning (Product)*** routine should be activated.

Formulas: Most likely, there will be a set of formulas for the transaction, as shown below:

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***)

Display elements: An object has to be associated with display elements such as screen formats for Windows or Web, graphic and dialog box styles, etc.

Transactions are declarative, and a great part of the desired description of reality is obtained by capturing their knowledge and systematizing it in a Knowledge Base.

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

A quick look at this object may indicate that our model contravenes Codd's Relational Model. New elements such as formulas aren't included in Codd's Relational Model, and repetitive groups are explicitly left out.

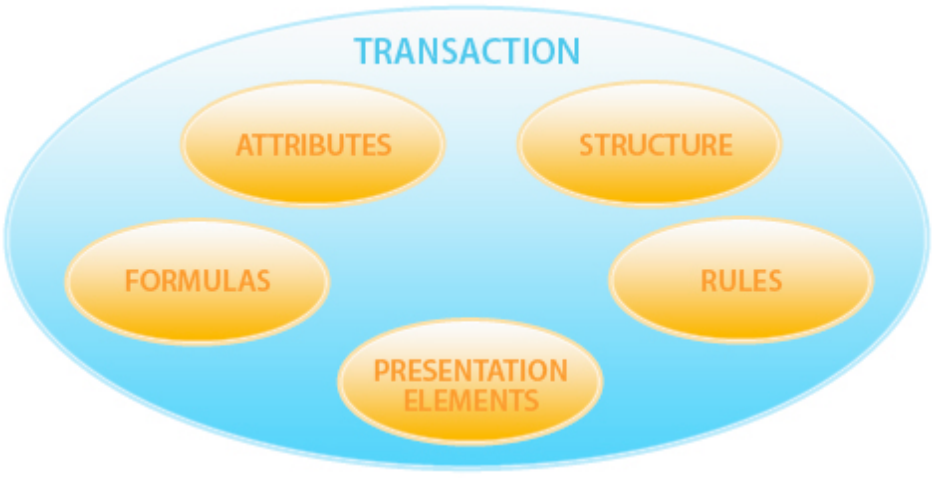
The Relational Model and our model have different objectives. It should be made clear that Codd never said that reality is formed by a set of relationships, which would be a big mistake. He said that the relational model is very good for storing and operating data.

The Relational Model is aimed at accurately representing data in a database by satisfying some desirable conditions: eliminating redundancy, introducing a small set of rules so as to avoid the greatest sources of data inconsistency, and including a set of operators to manage data appropriately.

The External Model will be used to obtain and store knowledge. This model aims at representing reality in the most direct and objective way possible. For this reason, we gather user views and store them in the model. Then, all the knowledge contained in them is captured and systematized in order to maximize the inference capabilities.

Within this context, the Relational Model (or rather, a superset of this model) is used to represent and/or manage data. It corresponds to the so-called Internal or Physical Model referred to in the ANSI SPARC report mentioned above.

In our theory, the Relational Model is extensively used.



PROCEDURES

Transactions, as we've seen, allow us to describe user data views declaratively and store the necessary knowledge to automatically design, generate and maintain the database and programs that users need to manage their own data views.

But this is not enough, and batch or similar processes are necessary. These processes can't be inferred from the knowledge provided by data views.

In the first GeneXus version released, transactions solved approximately 70% of the problem by generating the corresponding programs and 30% of the programs were missing.

To meet 100% of customer needs, we decided to use an object that allowed procedural descriptions: a high-level procedural language.

What features should this language have? The best procedural languages in the market were considered and their features analyzed. Interesting constructions were found, such as the well-known **For Each** ... **End For** that allows working with record sets in a simple, high-level way.

However, all languages studied have a serious drawback in that they need references to low-level physical elements such as tables; therefore, when writing a program the developer should know from which table to take each attribute.

For GeneXus, this is unacceptable because **if an attribute is changed to another table, the description would no longer be valid!**

In other words, the External Model doesn't include tables, so any specification that contains them may not be orthogonal. That's why we wouldn't be able to ensure automatic application maintenance.

The solution was to design a language whose syntax is inspired in the most advanced constructions of the best languages, and whose arguments are always part of the External Model.

For example, instead of tables, we'll talk about attribute sets necessary to perform an action. The system, at automatic generation time, determines which tables are involved and, if applicable, which indexes are used and how to do it. With all this, it generates the program.

Language syntax won't be dealt with in detail here. It's a procedural language with instructions to manage data that remains valid after structural changes in the database, and that has the logical and arithmetical instructions common in a high-level procedural language.

In this way, all cases found have been solved, following GeneXus' guidelines on consistency and orthogonality.

What are the strengths and weaknesses of this object?

> **Great strength**: Its procedural capabilities complement the declarative capabilities of Transactions, and (as we'll see in other GeneXus objects) they ensure the solution of any business system problem.

Weakness: The task of writing procedural descriptions is less productive than writing declarative descriptions, as in Transactions.

To resolve this situation, elements have been and will be added in the near future to lessen the need for procedural descriptions.

The most important object types are **Transactions** and **Procedures**. They allow a reasonable description of reality even though, with time, a significant set of new objects has been added: some of them to complete our description (**GXflow** for describing business operations workflow and generating the necessary code to use the specialized management server); others to enable more user friendly dialog boxes as new architectures supported them (**Work Panels**, **Web Panels**, **GXportal**); others to ease knowledge reuse in order to significantly increase productivity (**Business Components**, **Data Providers**, **Mini-Procs**, **Patterns**); and others to take Database (**GXquery**) or Data Warehouse (**GXplorer**) querying to end users. These objects can be combined to meet the most complex needs.

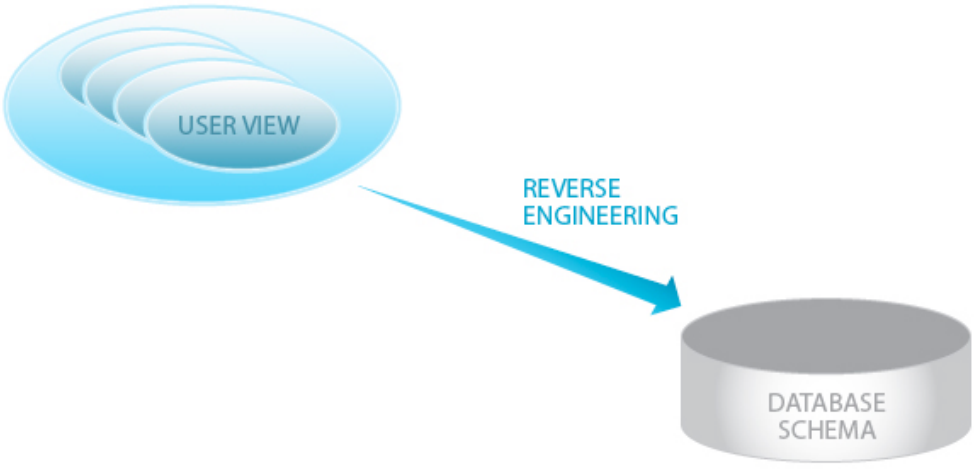
Is GeneXus' object type set complete? It can't be demonstrated in theory. In practice, it has proven so. More than 40,000 GeneXus developers worldwide exclusively use GeneXus to develop mission-critical systems, which accurately represents the IT community in general and reality itself. Also, this shows that it completely meets the needs of today's business systems.

But it isn't a closed set. In the future, new needs will probably arise that will lead us to introduce new object types in order to meet them.

THE RELATIONAL MODEL'S IMPORTANCE IN BUILDING THE KNOWLEDGE BASE

It's reasonable to believe that there's only one relational model for a given set of data views. We won't prove it here, but if this statement is true, we can find a reverse engineering process that uses the set of data views to deliver this minimum, relational database schema.

Achieving this reverse engineering process was the first research breakthrough of the GeneXus project. Then, building the necessary Knowledge Base over it has been a successful, ongoing work.



EXTERNAL MODEL AND KNOWLEDGE BASE

Initially, **Knowledge Bases** are associated to a set of inference mechanisms and contain general rules **[5]** which are independent from any particular application. When describing the object user's reality, descriptions are stored in the **External Model**. The system automatically captures all the knowledge contained in the **External Model** and systematizes it, adding it to the **Knowledge Base**. Additionally, over the previous knowledge, the system logically infers a set of results that help make future inferences more efficient.

GeneXus works permanently over the Knowledge Base. All the knowledge in the Knowledge Base is equivalent to that of the External Model (a subset of the Knowledge Base), as it consists of the External Model, rules and inference mechanisms which are independent from this External Model, and a set of other elements automatically inferred from it.

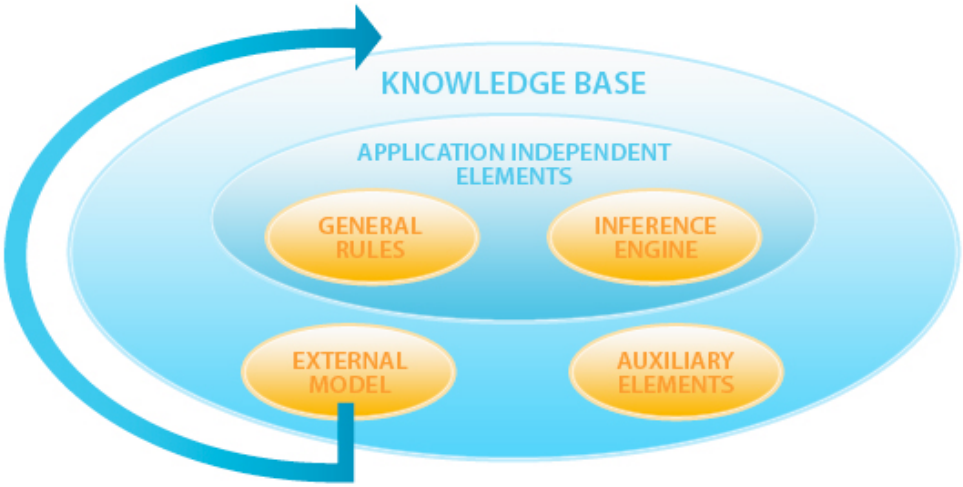
The developer can change the External Model by changing objects of the user's reality. Those changes will automatically propagate to all the elements that need it, such as other elements of the Knowledge Base, database and application programs. Likewise, the developer can't directly change any element that doesn't belong to the External Model.

This work focuses exclusively on the **External Model** (the **What**). It refrains from dealing with the **Knowledge Base** that contains and maintains it, and that constitutes the **How**, a complex set of mathematical and logical tools which are the basis of the inference processes and intermediate results used to increase the efficiency of these inferences. Knowledge about the How is not necessary to successfully use GeneXus.

What's more important: the What or the How? Neither works without the other, but if the What is not correct, everything is wrong.

Something very important can be inferred from the above, which goes beyond the current implementation of GeneXus: **ALL** the knowledge is contained in the External Model and because of that, we could support the **Knowledge Base** in a completely different way tomorrow, and our customers' knowledge would still be reusable without any problems at all.

In sum: The Knowledge Base and associated inference mechanisms constitute a big "inference machine." A good example of this is that given a data view we can automatically infer the necessary program to manage it.



D-DAY AND PROTOTYPING

Prototyping is a great resource and if used properly, it avoids an exaggerated dependency on the final test, or even deployment (D-Day, where everything happens and Murphy's Law especially applies).

Because of GeneXus' features, everything can be tested at any time and every object can be prototyped at the most convenient time. This resource is very useful and a direct consequence of the GeneXus theory. In particular, it's a consequence of the automatic propagation of changes, an important and exclusive GeneXus feature

GENEXUS BASICS OVERVIEW

Automatic knowledge management. It's based on a model of the object reality, an **External Model** that can be automatically managed by computer software. In particular, one that provides:

Consistency: This model is always consistent.

Orthogonality: External Model objects are independent from each other. Adding, modifying or deleting an object won't require changing any other object.

Automatic generation and maintenance of the database and programs.

External Model: The essential knowledge corresponds to an External Model that can't contain any physical or internal elements such as files, tables, entities, entity relationships, indexes or anything else that may be automatically inferred from that External Model.

Integration: GeneXus has a set of proprietary objects that are conceptually integrated, and in the Rocha version it will provide new levels of integration with other modules or products developed by Artech or third parties.

GENEXUS TRENDS

GeneXus is constantly evolving and its evolutionary trends can be represented with four dimensions: **COMPLETENESS, PRODUCTIVITY, UNIVERSALITY** and **USABILITY**.

COMPLETENESS: It measures the percentage of user system code that is automatically generated and maintained by GeneXus.

Since 1992, GeneXus' completeness has always been of 100%. This is a fundamental commitment because it involves automatic propagation of changes and, therefore, a dramatic reduction of development and maintenance costs.

UNIVERSALITY: It measures the coverage of the most important platforms available. Currently, GeneXus supports all the active platforms, which are those with a large and growing installed base.

In this way, GeneXus offers customers a great freedom of choice because they can move their GeneXus-developed applications to another supported platform at a low cost.

PRODUCTIVITY: It measures GeneXus development productivity increase compared to manual programming.

For many years, GeneXus aimed at increasing development productivity by 500%, which was enough.

Today, customers need increasingly complex systems due to new devices and new needs, especially for systems that meet them regardless of complexity, while remaining friendly to users that generally can't be trained.

What's more, those systems must be developed in less time because of an ever-shortening time-to-market.

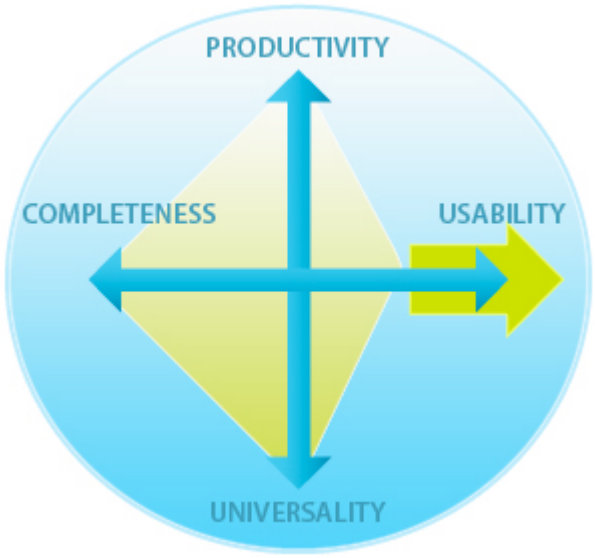
In 2004, these objectives were changed to 1000% for version 9 and to 2000% for the new version code-named Rocha.

USABILITY: It measures ease of use in general terms. This involves technical users, but also strives to reach more non-technical users and enable them to take advantage of GeneXus.

Using GeneXus implies a great increase in usability, as a GeneXus developer can develop outstanding applications for a given platform without expert skills in that platform, which represents a significant basic level of usability.

However, the idea is to push the envelope and enable many more users, with less technical prerequisites, to take advantage of GeneXus.

The Rocha version will greatly improve usability and this trend will continue in future versions.



WHAT'S NEXT? IS OUR MODEL WRITTEN IN STONE?

Certainly not; principles are permanent and proven by experience, but we should always look at GeneXus with a critical eye and expand it, so that it lets us describe new cases emerging from reality in the simplest way possible

.

BIBLIOGRAPHY AND REFERENCES

[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:

Regardless of the particular rules associated to one company's model, there are general rules which are always true and are independent from any application. For example:

Need for Consistency: Without going into further detail, it can be said that as in every science, consistency is the main source of rules: we assume that reality is consistent; therefore, any representation of reality should also be consistent.

On the basis of this principle, we can infer many general rules that will enrich our model.

One simple but important case is that of **referential integrity**:

1. Given X, a simple or compound attribute which is key of table T1, if X also appears in table T2, given a row of T2 there must be a row of T1 where T1.X = T2.X
2. Given X, a simple or compound attribute which is key of table T1, and Y, a simple or compound attribute such that Y = subtype(X), if Y appears in table T2, given a row of T2 there must be a row of T1 where T1.X = T2.Y
3. Given Y, a simple or compound attribute that appears in table T2 such that there isn't a table T1 where Y is key nor X (such that Y = subtype(X)) is key, Y can only appear in table T2.

CONTENTS