# The Know-It-All Project:

# A Case Study in Framework Development and Evolution

**Greg Butler**
**Ling Chen**
**Xuede Chen**
**Ashraf Gaffar**
**Jinmiao Li**
**Lugang Xu**

*Concordia University*

**ABSTRACT**

The Know-It-All Project is investigating methodologies for the development, application, and evolution of frameworks. A concrete framework for database management systems is being developed as a case study for the methodology research. The methodology revolves around a set of models for the domain, the functionality, the architecture, the design, and the code. These models reflect the common and variable features of the domain. There is a set of alignment maps between models that provides traceability while preserving commonality and variability.

From a database perspective, our aim is to provide support for all data models, integrated and heterogeneous databases, and eventually support for incomplete and uncertain data. The current prototype covers the relational data model. Our short-term goal is to add deductive databases and graph databases, thus providing diagrammatic queries to support applications in bioinformatics.

## 1 Introduction

Despite the dramatic improvement on design techniques and tools supports, it is still difficult and error-prone to design and implement complex software. Systematic reuse using domain engineering, product lines, or frameworks has been successful in delivering the economic benefits of reuse. An object-oriented framework is a concrete realization in source code of a domain-specific software architecture. While frameworks are often cited as an example of the reuse of requirements, architecture, design, and code, there is a prevailing conception that the only concrete model used for frameworks is the source code itself.

The Know-It-All project, which has been underway at Concordia University since 1997, has three sets of aims:

1. to research methodologies and models for framework development, application, and evolution;

2. to develop a framework for database management systems, supporting a variety of data models of data and knowledge, the integration of different paradigms, and heterogeneous databases; and

3. to apply the Know-It-All framework to advanced database applications for bioinformatics.

The research on software methodology in an academic setting needs a concrete case study in order to evaluate the methodology and models; the framework case study needs to be evaluated by creating real applications. For us the database framework is a case study to validate our methodology, while the applications to bioinformatics allow us to validate the database framework. A side-benefit of this is that, along the way, the research in software technology leads to a platform for research in database technology, which in turns leads to advances in bioinformatics and genomics.

Methodologies for the development of a framework have been suggested that use domain analysis, software evolution, and design patterns. However, identifying the required flexibility for the family of applications and designing mechanisms that provide this flexibility is the problem. Furthermore, the evolution of the framework must be considered, especially as all frameworks seem to mature from initial versions through to a stable platform.

Refactoring of source code is one approach suggested for the development and evolution of frameworks. Our work attempts to show how a framework can be described by a set of models, not just source code, and how refactoring as a concept can be broadened to apply to each of these models. Our view of refactoring includes the underlying motivation for the restructuring performed during the refactoring. This motivation could be expressed as an issue, rationale, or force (or sets of these). In overview, *cascaded refactoring* is the process of sequentially refactoring each model, one after the other, where the restructuring of the previous model provides the motivation for the refactoring of a model. Cascaded refactoring is the basis of our methodology.

## 2 Background

Frameworks [8)] offer a concrete realization of a product line. A framework is an architecture, plus an implementation, plus documentation that capture the intended use of the framework for building applications. A framework provides a highly effective mechanism for software reuse within an application domain. The framework captures the features that are common across the product line. In return for relinquishing some design authority, the developer can build a new application faster by hooking to the framework just the code that is unique to the new application.

A domain is an area of knowledge that is scoped to maximize the satisfaction of the requirements of its stakeholders; includes a set of concepts and terminology understood

by practitioners in that area; and includes the knowledge of how to build software systems in that area.

The basic properties of a framework are *modularity*, since abstract classes with stable interfaces, encapsulate and localize change in "hot spots", the points of planned variability; *reusability* of analysis, design, and code; *extensibility* by providing explicit hot spots or "hooks" for planned variability; and *inversion of control* since the framework calls the custom code.

## 2.1 Terminology

There is confusion with terminology since different communities using different words–– feature, hook, hot spot — to describe essentially the same concept: a variable aspect of the system.

A *feature* is any aspect of a system used to characterize it to a stakeholder. This includes both common and variable aspects, but the variable features are the most interesting.

A *hook* is a point in the framework that is meant to be adapted in some way, such as by filling in parameters or creating subclasses.

A *hot spot* is a variable aspect of a framework domain, whereas a fixed aspect is called a frozen spot.

## 2.2 Framework Development Approaches

We can classify the approaches to framework development as

- example-driven, bottom-up, incremental development;
- top-down domain engineering;
- hot spot generalization;
- use case-driven; and
- hybrid, which combines one or more of the above.

These classifications are not mutually exclusive.

The classic development approach in the object-oriented community [25)] example-driven, bottom-up, and incremental. A framework is refactored as new requirements are encountered as the applications are developed one by one.

Domain engineering [28)] often invests more effort up-front in the modeling of the domain and the specification of an architecture for the product line. Their understanding of the domain is more complete, and is captured in range of models, including the context,

scope, taxonomy, data dictionary, feature model, domain specific software architecture, descriptions of major functionality as use cases and algorithms, and exemplar systems in the domain.

Hot spot generalization [22,26] identifies the points of flexibility — the *hot spots* — required in the product line; classifies the type of flexibility required (using patterns or meta-patterns); and then assigns a subsystem with a facade class within the design to implement the hot spot.

The use case-driven approaches extend the OOSE methodology of Jacobson from simple applications to product families. They include Catalysis, RSEB [13], and Feature RSEB[11].

# 3 Our Methodology and Models

Our methodology for framework development and evolution is a hybrid approach. It combines the modeling aspects of the top-down domain engineering approaches, and the iterative, refactoring approaches of the bottom-up object-oriented community. The basic philosophies are

> *Framework development is framework evolution.*
> *Evolution = refactoring + extension.*

The methodology weaves together steps for partial domain engineering to better understand the domain and how to evolve the current working set of partial models, and steps of system refactoring and extension.

The methodology stresses *traceability* between the models. In this respect it follows FeatureRSEB [11] and FORM [16]. In the context of a framework or product line, the methodology must also stress the commonality-variability distinction. We define an *alignment* [1] of models to be a traceability mapping that is consistent with the mapping of common and variable features.

While the working set of partial models is incomplete, and hence some mappings for traceability or alignment will be partial maps, we still desire a *consistent, coherent, aligned set of models and maps*.

Our models are

- $M_f$, the feature model;
- $M_u$, the use case model;
- $M_a$, the architectural model;
- $M_d$, the design or class hierarchy;
- $M_s$, the source code;

and also a test plan and test cases as part of the documentation.

**Feature Model** A *feature* is any aspect of a software system that is used to characterize the system to the users, customers, or developers. A *feature model* is a collection of all the features and their relationships. For a product line or framework, a feature model classifies each feature as *mandatory, optional,* or *alternative* in order to express the commonality and variability across the applications in the product line. An application is specified by its *feature set*, which is the set of all features provided by the application. A feature set is a subset of the set of all features, and the feature set satisfies the constraints amongst features imposed by the model.

There are several views of structure for a feature model. The basic concept is a finite set *F* of the features. The usual relationship of interest is the *subfeature* relationship that defines a hierarchy of features. There is also a *dependency* relationship between features, and the actual dependency may be specified using a constraint. One can consider the subfeature relationship as a special kind of dependency.

Structure is also provided by two classifications. The usual classification into mandatory, optional, and alternative can be considered as a map *variability* from *F* to the set {*mandatory, optional, alternative}*. The FORM methodology adds a classification of features into capability features, domain technology features, operational environment features, and implementation features. This can be considered as a map *kind* from *F* to the set *{capability, technology, environment, implementation}*.

**Use Case Model** Use cases capture the functionality of a system, called the *target system*, as it is meant to behave in a given environment called the *host system*. A use case describes how a group of external entities, called *actors*, make use of the system under consideration. The use they make is modeled by the passing of signals or information between the actors and the system.

A *use case* is a description of a cohesive set of dialogues that the primary actor initiates with a system. The dialogues are *cohesive* in the sense that they are related to the same task, or form part of the same transaction. Cohesiveness is often determined by having a *goal* in common for the tasks, or by having a common *responsibility*.

The high-level view of use cases provides a *generalization* and *inclusion* relationship between use cases. Abstract use cases may occur in the model to express commonality amongst use cases, even if the abstract use case has no concrete scenario; its (leaf) children in the generalization hierarchy will have concrete scenarios. The inclusion relationship connects a task with a subtask represented as a use case.

Extension points and variation points in a use case model allow the variability in functionality to be depicted.

A use case encompasses a collection of scenarios, since there may be several ways in which an actor can (successfully or unsuccessfully) attempt a task. Potts [21)] defines a *scenario* as "a description of one or more end-to-end transactions involving the required system and its environment". Scenarios may be classified as follows. The *main scenario* describes the usual way in which the task is successfully performed. Typically, in the main scenario, the simplest sequence of interactions is described, and it is assumed that

all steps execute successfully. A *variant scenario* describes another way of using the system where it is assumed that all steps execute successfully. An *exceptional scenario* describes a scenario where exceptional or error conditions may arise. It may be possible to recover from the exceptions and therefore successfully complete the task — this is called a *recovery scenario* — or it may not be possible to recover — this is called a *failure scenario*.

A scenario may be described in several ways, from a simple narrative text description, to numbered steps indicating the subject-action-object triples, to basic Message Sequence Charts (MSCs) [24] or UML sequence and collaboration diagrams.

A use case may also be described in terms of *episodes*. Each episode represents a subtask, or the parts of the dialogues in the scenario that perform the subtask. A high-level Message Sequence Charts (MSCs) [24] can be used to depict a use case and its episodes utilizing operators for sequence, alternation, iteration, parallel execution, and exceptions.

Use cases may be classified into different kinds, such as business use case, requirements use case, analysis use case, system use case, and design use case. These reflect the perspective of the writer, as well as the level of detail and the kind of target system.

**Architectural Model** Software architectures provide an abstract description of the organizational and structural decisions that are evident in a software system. The development of an architecture requires the decomposition of system into subsystems, the distribution of control and responsibility, and the development of the components and their connections or means of communication. General principles of information hiding, such as the use of modules, layers, and abstract machines (API's) help manage the complexity of the task.

An architectural style [27] may be described in terms of

- a *vocabulary* of the basic design elements (components and connector types),
- a set of *configuration rules* which constrain how components and connectors may be configured,
- a *semantic interpretation* which defines when suitably configured designs have a well-defined meaning as an architecture, and
- *analyses* that may be performed on well-defined designs.

From the perspective of UML and RUP [17], the description comprises four diagrams plus the use cases/scenarios. The diagrams cover the design view, the process view, the implementation view, and the deployment view. The design view presents the subsystems, their interfaces, the dependency between subsystems, and nesting of subsystems.

It is possible to take a use case view of a subsystem, where the subsystem is regarded as the target system with the host system consisting of all other subsystems in the

architecture. Then a subsystem's services are described in the use case model for the subsystem, and accessed through its interfaces.

It is also possible to take a class view of a subsystem, where one identifies a subsystem with a facade class. One identifies the subsystem interfaces with the class methods.

At present we model architecture using layers, subsystems, and interfaces using the UML notation. So far that has been adequate. However, in the future, we plan to investigate the use of the Siemans approach [14)] to architectural modeling.

**Design Model** The design is modeled in the classic object-oriented style separating static structure and dynamic behavior with the class diagrams, interaction diagrams, and state diagrams of UML. Design patterns usage is encouraged.

**Trace Maps** The alignment maps used are

- $T_{fu}$, the trace map from the capability features to the use case model;
- $T_{fa}$, the trace map from the operational environment features to the architectural model;
- $T_{fd}$, the trace map from the domain technology features to the design;
- $T_{fi}$, the trace map from the implementation features to the source code;
- $T_{ua}$, the trace map from the use cases to the architecture;
- $T_{ud}$, the trace map from the use cases to the design;
- $T_{ad}$, the trace map from the architecture to the design; and
- $T_{di}$, the trace map from the design to the source code.

## 3.1 Framework Evolution by Cascaded Refactoring

Evolution is naturally viewed as refactoring followed by extension in those bottom-up methodologies for framework development that are centered on refactoring. The new structure of the system and the refactorings to effect the re-structuring are chosen with the required extension in mind. However, there is little literature on this connection between refactoring and extension. We also say very little on this topic. A useful reference on extension of object-oriented systems is Mätzel and Bischofberger [19)].

Refactoring [9)] is a behavior-preserving program transformation that automatically updates an application's design and underlying source code. Primitive refactorings perform simple edits such as adding new classes, creating instance variables, and moving instance variables up the class hierarchy. Compositions of refactorings can create abstract classes, capture aggregation and components, and even install design patterns [29)].

Our methodology extends the notion of refactoring that has been applied to source code, and to design in the form of class diagrams, to other models of software systems. The methodology relates the set of refactorings across the set of models, through change impact analysis using the trace maps. This process, called *cascaded refactoring*, is a series of refactorings of the models. The impact of the refactorings for a model $M_x$ is translated via the trace maps that have domain $M_x$ to determine constraints on the refactorings of models $M_y$. The methodology is unique in several ways:

- to view refactoring as an issue-driven activity;
- to document the rationale of an application of a refactoring as a triple: intent of restructuring, choice of refactoring(s), and impact of the restructuring; and
- the notion of cascaded refactoring, where the restructuring of one model determines constraints on the restructuring of other models (via the traceability and alignment maps).

We still need to investigate how the choice of refactorings is influenced by the desired extensions of the framework. The contributions [6)] we have made to date are

- refactoring of feature models;
- refactoring of use case models;
- preliminary work on refactoring of architectures; and
- clarify the "behavior" that is preserved by these refactorings.

### 3.2 Documenting Frameworks

The application of a framework to create a product line is hampered by three issues:

1. the learning curve for a framework is too steep;
2. the framework itself must be stable before application development can be undertaken on a large scale; and
3. there is a lack of experience across the maturity lifecycle of a framework — customizing a white-box framework is very different from applying a generator to build an application.

Our research [4,5)] has clarified many of the issues, and recommended a set of documentation to assist application developers.

- An overview of the framework setting a context for the domain and the variability in the framework.
- A graded set of example applications illustrating the customization of features.
- A cookbook of recipes of how to customize each feature, organized in a spiral fashion from easiest/most common to most advanced/least common, and using the examples to keep the presentation concrete.

We view documentation as a key step in framework development, since the ability to write clear documentation that explains how application developers should use the framework means that the concepts of the design are clear and that the steps required for customization have been clearly thought out. Hence, documentation verifies that the framework is easy to use, and this is the overriding goal of framework design.

# 4 The Case Study for Domain of Databases

Know-It-All is an object-oriented framework for database management systems. It is written in C++, with some Java for user interfaces, and XML for communication of data between the C++ framework and the Java tools. The user interfaces will provide a full range of query mechanisms, from icons for canned queries, to forms, to textual queries in set comprehension languages, and diagrammatic queries.

| Dimension | Features |
|---|---|
| Data Model | none, flat file, relation, deductive, graph, object, statistical, time-sequence, string, text, structured text, multimedia, spatial, GIS, constraint |
| Functionality | browse, query, bulk load, update, transaction, trigger, stored procedure, integrity constraint, workflow |
| DBA Functions | configure, monitor, tune, evolve, auto-configure, auto-tune |
| Feature | concurrency, recovery, security, client/server, distributed, heterogeneous |
| Performance | data clustering, compression, encryption, index, query optimization, view materialization, caching |
| Scale | #user, #transactions, #databases, #processors |

Table 1: Some Dimensions of the Database Domain

Our domain analysis considered common database textbooks and research literature, commercial and public-domain systems, and research prototypes. The domain is huge, as indicated by a summary in Table 1, however, our initial scope can be seen in the top-level feature model Figure 1.

The use case model Figure 2 not only shows the major functionality of the system, but also the context of the system and the roles and interaction between users, application developers and database administrator.

Know-It-All is designed with scientific databases in mind, and does not provide for transactions. Instead, it provides a data feed mechanism for bulk or incremental data loads. The prime concern is querying of existing data. The framework provides a generic infrastructure for database management systems and allows them to support a range of data models (relational, object, object-relational, etc) where the data model itself, and its constituents for query language, query optimizing, indexing, and storage have clearly defined roles. A database in Know-It-All is seen as a series of layers, each of which provides the same interface. The usual breakdown of responsibilities into physical, logical, conceptual, and view layers is followed by Know-It-All. However, a database, as seen by the end-user, allows views of views, and mappings of object conceptual models to relational conceptual models. Eventually, Know-It-All will incorporate composite databases (such as integrated or heterogeneous databases) and make no distinction between simple and composite databases.

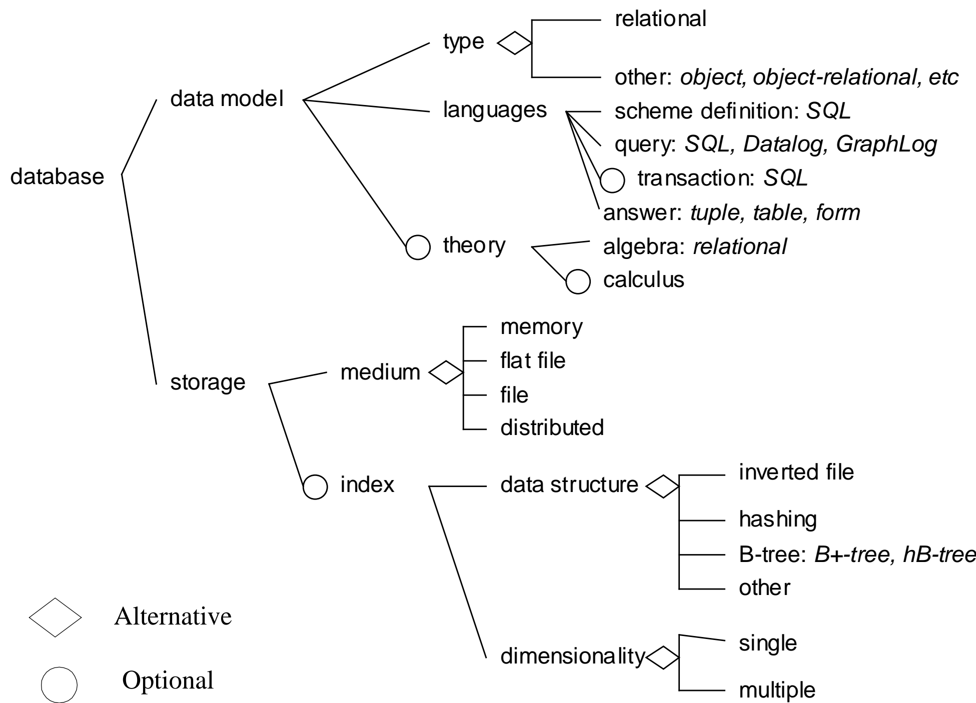


Figure 1: Top-level Feature Model of the Know-It-All Framework

Each layer in Know-It-All is basically a translator between its client layer and its supplier layer(s), as shown in Figure 3. A layer provides a mechanism to decompose or translate queries, and a mechanism to reconstruct answers (for example, an execution plan for relational algebra expressions). The translation is done with the aid of the schema, and produces both the translated query, and the mechanism to reconstruct answers. The layer architecture is adapted from one [20)] for heterogeneous databases, while the reconstruction mechanism is designed as an iterator (which is a tree of iterators) that returns the next result [10)]

Our prototype implements the relational data model. For support of diagrammatic queries, it is a priority to support deductive and graph databases. For general needs in genomics, we need to support object databases. This will allow us to support spatial, temporal, and image databases. There are now also algebras for object-relational databases, so they will also be supported.

## 4.1 Subproject: Query Optimization Framework

The Know-It-All framework contains a subframework for query optimization, and a subframework for indexing techniques. The optimization framework is based on the broadly applicable OPT++ [15)]. Figure 4 shows the overall class diagram. In this figure, components of this framework are represented as packages and only the top-level classes in each component are shown.

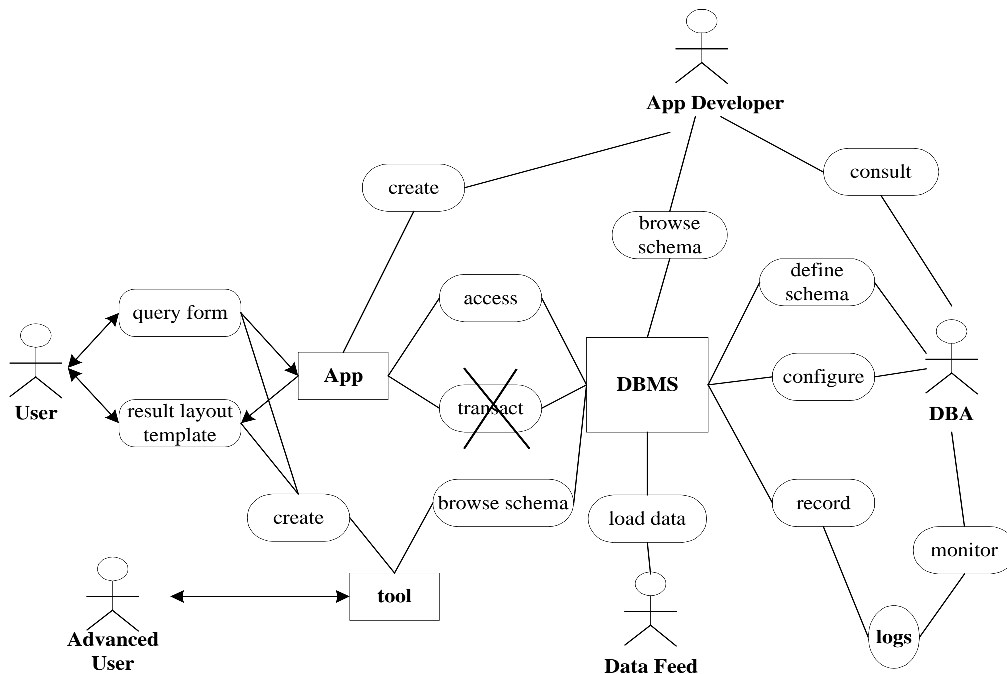


Figure 2: Use Case Model of the Know-It-All Framework

The Search Strategy component encapsulates the system control flow. It implements different search approaches that are used to perform the search in the search space. At any instance of time, only one search approach dominates. It also encapsulates the cost model and implements the cost estimation framework. It contains the logical query plans and physical query plans that have been built, and it performs the cost evaluation on these plans and prunes out the sub-optimal ones. The Search Strategy component maintains an aggregation reference to the Algebra component because each logical query plan is represented by applying an operator (in the Algebra component) to its root node, and each physical query plan is represented by applying an algorithm (in the Algebra component) to its root node. The major classes in the Search Strategy component

include: the QUERYOPTIMIZERFACADE, the SEARCHSTRATEGY, the SEARCHTREE, the OPERATORTREE, and the ALGORITHMTREE. The class QUERYOPTIMIZERFACADE is the facade of this system. It simplifies the use of the system. The SEARCHSTRATEGY is an abstract class and provides interfaces for all search strategy approaches that are used in query optimization. The SEARCHTREE represents the search tree that is used to explore the optimal plan. It implements the search strategy interfaces. The OPERATORTREE represents a logical query plan of a query. It is an algebraic expression that represents the particular operations to be performed on data and the necessary constraints regarding order of operations. The ALGORITHMTREE represents a physical query plan of a query. It is a tree of algorithms that specifies the particular order of operations and the algorithm used to implement each operation.

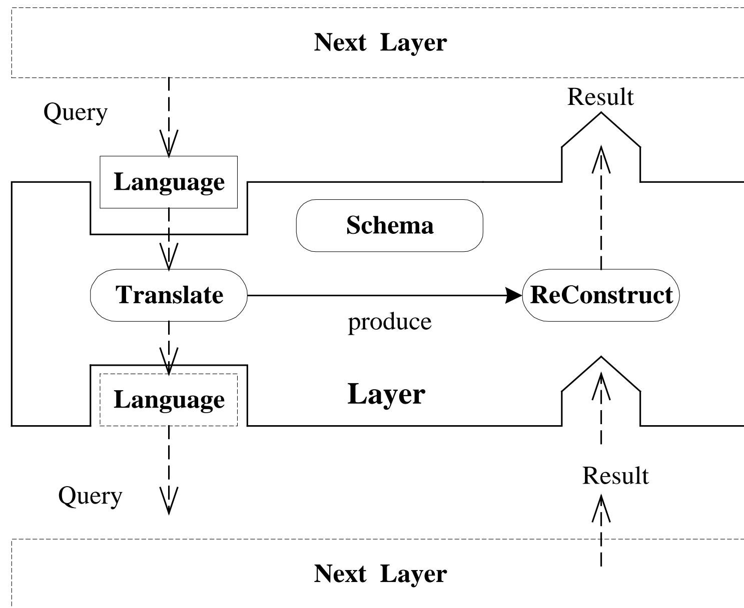


Figure 3: Basic Building Block

The Algebra component defines the logical operators (operations on the database) and the physical operators (execution algorithms for these logical operators) in the database. There are two major classes in the Algebra component: the DBOPERATOR and the DBALGORITHM. The former defines the interfaces for all possible logical operators in the database system. The latter defines the interfaces for all possible execution algorithms for these logical operators. They are all abstract classes and depend on the optimizer-implementor to define the actual database algebra. The definitions of the interfaces for the logical and physical operators should consider easy extension of possible operations that are performed on them.

The Search Space component defines what the search space is. It defines the operations on the Algebra. The effects of executing the operations in the Search Space component are:

- A tree is transformed to another tree. The new tree becomes bigger as a result of expansion or these two trees are equivalent as a result of some equivalent transformations.

- A logical query plan is converted to a physical query plan.

Conceptually, there are two search spaces. One is the logical search space. It is a space of logical query plans and decides how one logical query plan derives from another. The other is the physical search space. It is a space of physical query plans and decides how one physical query plan derives from another. The conversion from the logical query plan to the physical query plan can be deemed as an operation on the logical search space. There are two major classes in the Search Space component. One is the OPERATORTREEVISITOR and the other is the Generator. The OPERATORTREEVISITOR is an abstract class and defines all operations performed on the Algebra component. It serves as a bridge between the Algebra component and the Search Space component. The Generator is not a class defined in this framework. In fact, there are three generator classes, each is defined for one of the following operations: tree expansion; tree transformation; or conversion of a logical query plan to a physical query plan.

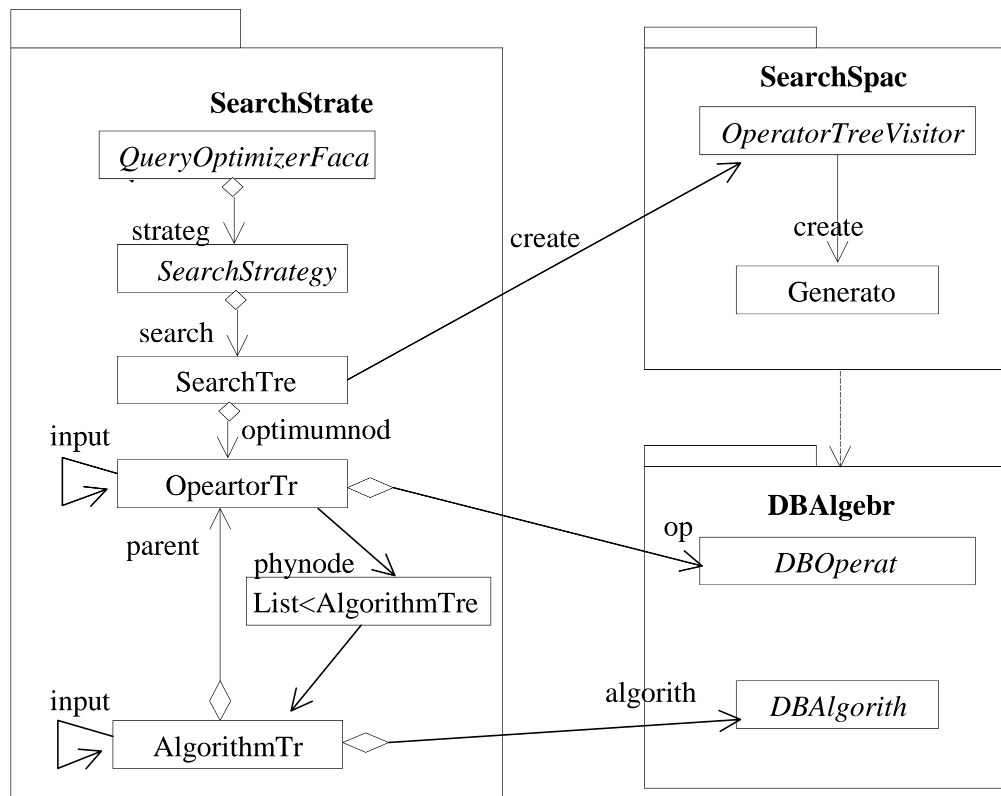


Figure 4: Overall Class Diagram

## 4.2 Subproject: Tree Index Framework

The indexing subframework is based on GiST [12)], which covers tree-based indexes, including multi-dimensional trees and similarity-based retrieval. We have re-designed GiST to follow the design of STL containers in C++, to follow good object-oriented design practices, and use design patterns. In the future, the indexing subframework will be extended to cover inverted files and hashing techniques.

## 4.3 Subproject: Diagrammatic Query Interface

Data management, access, and mining are at the heart of bioinformatics. While relational databases are the accepted standard within industry, there has been considerable research into deductive databases and graph databases to extend the capabilities of relational databases. Deductive databases allow a view, called the intensional database, to be defined using logical rules, and allow logical queries against the view. Since the rules allow recursive definitions, the resulting expressive power of the query language is greater than ordinary relational databases. Graph query languages are even more expressive, while having the very important property of a visual representation.

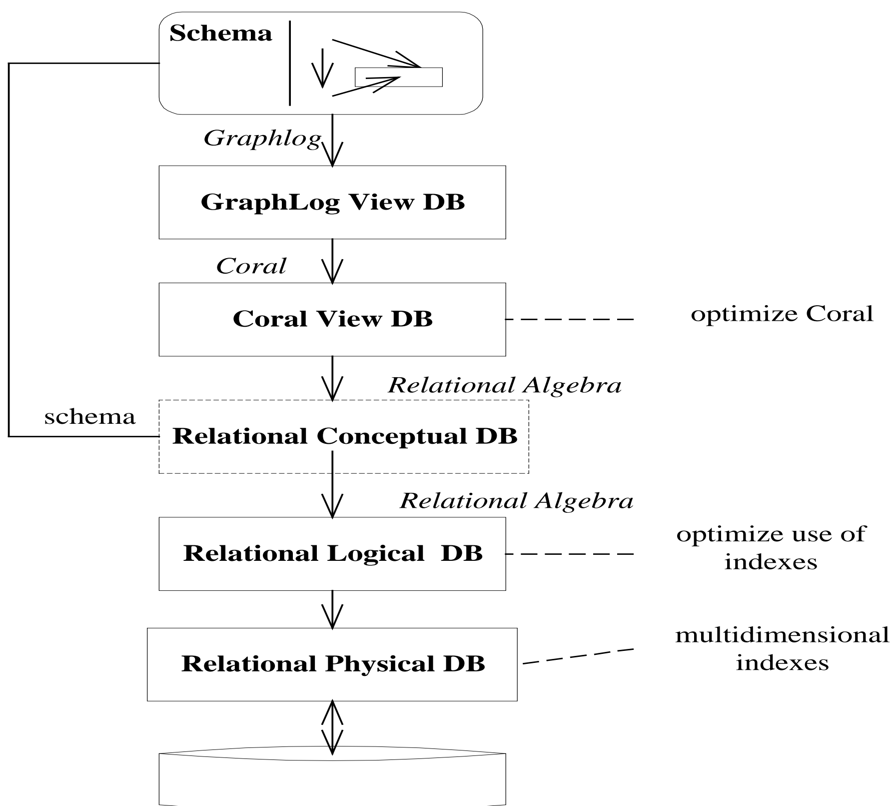


Figure 5: Hy+ and GraphLog in the Know-It-All Framework

The group of Alberto Mendelzon at the University of Toronto developed the GraphLog graph query language based on hygraphs, and a visual interface, called Hy+, for expressing queries and browsing their results [7]. GraphLog is a graph query language extending Datalog and negation. The language has recursion, usually as transitive closure, and has path expressions. Path expressions are similar to regular expressions. A path expression can refer to a primitive relation, or construct more complex path expressions using the operators of negation, inverse, concatenation, alternation, kleene closure (*), or transitive closure (+).GraphLog is more expressive than Datalog (and SQL). Hy+ is a visual Smalltalk environment for composing GraphLog queries and viewing the results of the query. In the diagrams, nodes represent entities, while labels add identifiers and attributes to nodes. Edges represent relations and edge labels are path expressions. Blobs are nodes containing other nodes and represent a relation on containing and contained nodes. The underlying data structure of GraphLog and Hy+ is a *hygraph.*

The query interface for Hy+ imports the database schema, together with iconic descriptions for each class of entities and relations. The icons for relations are various forms of lines and arrows. A query, or a view definition, is constructed by selecting icons and composing a *hygraph*. A hygraph is basically a graph augmented with *blobs*. A blob relates a containing node with a set of contained nodes, and can be viewed as providing an aggregation or subset relation useful for abstraction of complex graphs.

The interface provides a window for the definition of views. A view may define a *property* of an entity, as a relation between the entity class and a “ground” symbol. A view may define a *relation* in terms of pre-existing concepts in the database and the views. A view may also define a *blob* in terms of pre-existing concepts in the database and the views. The icons for the defined property and relation arrows, and for blobs, appear in the schema window along with the icons for the database entities and relations. Closure of a relation is depicted as a dashed arrow of the same color as the underlying relation. A negation of a relation has a cross through the arrow.

The interface provides a window for the definition of queries. While composing a query, it is possible to have open the windows containing the definitions of the views that are used by the query. This, together with the schema window, guides the user to make well-formed queries.

The translation of GraphLog diagrams into logic programming notations such as Coral[23] or Datalog is straightforward. The hard part is ensuring efficient processing of the translated query. There has been no experimentation with Coral optimization strategies, and one really needs multidimensional indexes of the underlying relations to support efficient processing of the deductive programs.

Hygraphs are also the means for presentation of results of queries. The presentation is controlled initially by specifying a layout algorithm for the hygraph in a *layoutGraphLog* window, and by specifying which entities or relations to hide in a *hideGraphLog* window. As before, the use of icons and colors aids comprehension of the hygraphs. Hy+ also provided the ability to zoom in for more detail. The window for the presentation of results displays the hygraphs of the results, and provides tools for abstraction by collapsing blobs. It is very natural in Hy+ to refine queries by simply regarding the current query as a view definition, and then composing a query (a refinement) against the

database and using the new view. Of course, there is the potential to use the set of results as a materialized view.

While the initial Java version of the Hy+ interface will be implemented using Coral, we will eventually incorporate it as part of our framework. The first step is to incorporate deductive databases, in the form of Coral, into the Know-It-All framework for databases. This will be done by regarding the deductive database as a VIEWDB subclass defining an intensional database view of a relational database. See Figure 5. The second step is to incorporate graph databases directly into the Know-It-All framework, also as a VIEWDB subclass of either relational, object, or object-relational databases. Then we will incorporate recent techniques for the processing of path queries.

# 5 Applications to Bioinformatics

This is not the place for a long expose on genomics and bioinformatics, so we content ourselves with directing the reader to some of our other publications. An introduction to bioinformatics and its need for information technology can be found together our perspective on future directions [3)] for the technology. More specific to database technology is our paper [2)] on pathway databases, while the book [18)] covers the major database systems in bioinformatics.

# 6 Conclusion

The Know-It-All project is investigating methodologies for the development, application, and evolution of object-oriented frameworks. Although the research is being done in an academic setting, we are attempting to provide solid evidence of advances in modeling and methodology through the development of a case study framework for the domain of databases, and to apply this framework to real scientific applications.

The essence of the methodology — cascaded refactoring — has been formulated. It is an issue-driven process of restructuring and extension of a coherent set of (partial) models of the domain and the framework. The models underlay a use-case driven view of system development (as in RSEB), augmented by a layered feature model (as in FORM) to better model commonality and variability.

Many details of the methodology are still being investigated. This includes the choice of architectural model or models, and precise definitions of the trace maps for each pair of models. However, most of the unknowns about the methodology lie in cataloguing refactorings for the various models, and relating desired extensions of the framework to the choice of refactorings.

A subproject to develop modeling and refactoring tools for the use case models is underway, and we hope to initiate a similar subproject for the architectural models soon. These should throw much light on the gaps in our methodology.

The Know-It-All framework case study has completed a partial domain analysis and prototyped its architectural design in C++ for the relational data model. The query

optimization subframework has been completed, resulting in an implementation, models, and documentation on its use. The tree index framework will be completed in 2001.

The GraphLog interface, as a standalone system, will be finished in the near future, and over the longer term both Coral and GraphLog will be incorporated as data models in the Know-It-All framework.

Potential future work on the Know-It-All framework includes a subframework for physical storage, a subframework for hash indexes, and a subframework for inverted file indexes. The range of data models included in the framework will also be extended.

At present our bioinformatics platform is using the mySQL database for its data repository. In the short term, we plan to create a mySQL-compatible instance of the Know-It-All framework with the relational data model, and to use this as a mirror of the data repository. This will allow us to use the enhanced querying available in the framework, and allow us to delay the implementation of transaction management in Know-It-All.

## Bibliography

1. G. Butler. Developing frameworks by aligning requirements, design, and code. In Proceedings of 9th Workshop on Software Reuse (WISR-9), 1999.

2. G. Butler, Database technology for pathways. In Workshop on Computation of Biochemical Pathways and Genetic Networks, E. Bornberg-Bauer, A. de Beuckelaer, U. Kummer, U. Rost (eds), Logos Verlag, Berlin, 1999, ISBN 3-89722-093-8, pp. 89-95.

3. G. Butler, E. Bornberg-Bauer, G. Grahne, F. Kurfess, C. Lam, J. Paquet, I. Rojas, R. Shinghal, L. Tao, A. Tsang. The BioIT projects: Internet, database and software technology applied to bioinformatics In *SSGRR*'2000, Suola Superiore G. Reiss Romoli SpA, Coppoto, Italy. URL http://www.ssgrr.it/en/ssgrr2000/proceedings.htm

4. G. Butler and P. Dénommée. Documenting frameworks. In [8], pp.495-504.

5. G. Butler, P. Grogono and F. Khendek, A reuse case perspective on documenting frameworks. Proceedings of Asia-Pacific Software Engineering Conference, 1998, pp. 94-101.

6. G. Butler and L. Xu. Cascaded refactoring for framework evolution. In Proceedings of 2001 Symposium on Software Reusability, ACM Press, 2001, pp. 51-57.

7. M.P. Consens, F.Ch. Eigler, M.Z. Hasan, A.O. Mendelzon, E.G. Noik, A.G. Ryman, and D. Vista. Architecture and applications of the Hy+ visualization system. *IBM Systems J.* 33:3 (1994), pp. 458-476.

8. M. E. Fayad, D. C. Schmidt, and R. E. Johnson, editors. *Building Application Frameworks: Object-Oriented Foundations of Framework Design*. Wiley, 1999.

9. M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

10. G. Graefe. Iterators, schedulers, and distributed-memory parallelism. *Software - Practice and Experience* 26(4) (1996) 427-452.

11. M. L. Griss, J. Favaro, and M. d'Alessandro. Integrating feature modeling with the RSEB. In *Proceedings Fifth International Conference on Software Reuse*, pages 76-85. IEEE Computer Society, 1998.

12. J. M. Hellerstein, J.F. Naughton, A. Pfeffer. Generalized search trees for database systems. In *VLDB'1995* 1995, pp. 562-573.

13. I. Jacobson, M. Griss, and P. Jonsson. *Software Reuse: Architecture, Process and Organization for Business Success*. Addison-Wesley, 1997.

14. C. Hofmeister, R. Nord, D. Soni. *Applied Software Architecture*. Addison-Wesley, 1999. ISBN: 0201325713

15. N. Kabra and D. J. DeWitt. OPT++: An object-oriented implementation for extensible database query optimization. *VLDB Journal* 8,1 (1999) 55-78.

16. K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh. FORM: A feature-oriented reuse method with domain-specific reference architectures. *Annals of Software Engineering*, 5:143-168, 1998.

17. P. Kruchten. The 4+1 view model of architecture. *IEEE Software*, pages 42-50, November 1995.

18. S. I. Letovsky. *Bioinformatics: Databases and Systems*. Kluwer Academic Publishers, Boston, 1999.

19. K.-U. Mätzel and W. Bischofberger. Designing object systems for evolution. *Theory and Practice of Object Systems*, 3(4), 1997.

20. L. M. Mackinnon, D.H. Marwick and M.H. Williams. A Model for Query Decomposition and Answer Construction in Heterogeneous Distributed Database Systems, *Journal of Intelligent Information Systems*, 11, 69-87, 1998.

21. C. Potts, K. Takahashi, and A. Anton. Inquiry-based requirements analysis. *IEEE Software*, pages 21--32, 1994.

22. W. Pree. *Design Patterns for Object-Oriented Software Development*. Addison-Wesley, 1995.

23. R. Ramakrishnan, D. Srivastava, S. Sudarshan, P. Seshadri. The CORAL deductive system. *VLDB Journal* 3,2 (1994) 161-210.

24. B. Regnell, M. Andersson, and J. Bergstrand. A hierarchical use case model with graphical representation. In *Proceedings of Second International Symposium on Engineering of Computer-Based Systems*, pages 270-277. IEEE Computer Society Press, 1996.

25. D. Roberts and R. Johnson. Patterns for evolving frameworks. In R. C. Martin, D. Riehle, and F. Buschmann, editors, *Pattern Languages of Program Design 3*, pages 471-486. Addison-Wesley, 1997.

26. H. A. Schmid. Systematic framework design by generalization. *Communications of the ACM*, 40(10): 48-51, 1997.

27. M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

28. STARS. Organization domain modeling (ODM) guidebook —– version 2.0. Technical report, Lockheed Martin Tactical Defense Systems, 1996.

29. L. Tokuda. Evolving object-oriented architectures with refactorings. In *Proceedings of ASE-99: The 14th Conference on Automated Software Engineering*. IEEE CS Press, October 1999.