Knowledge is power.
(Ipsa Scientia Potestas Est)
Sir Francis Bacon (1561 - 1626), Meditations

From the book “Integration-Ready Architecture and Design” on Software and Knowledge Engineering

An Introduction to Knowledge Technologies

By Jeff Zhuk

Ontology is a controlled, hierarchical vocabulary for describing a knowledge system or knowledge-handling methods.

This chapter is an introduction to a development paradigm in which software and knowledge engineering are integrated. As always happens on the other side of an economic crisis, a new set of skills will be required. A growing number of developers will actively use the knowledge technologies reviewed in this chapter.

The chapter starts by talking about fundamental standards that currently bridge ontology and engineering: the Resource Description Framework (RDF), the Semantic Web language DAML+OIL (DARPA Agent Markup Language + Ontology Inference Layer), Topic Maps concepts, and their XML Topic Maps (XTM) standard knowledge exchange format.

We’ll continue with a brief overview of data mining methods with coming Java support and eventually discuss the challenging topic of *generic* knowledge, not just knowledge of a specific business domain, expressed in natural language. The final part of the chapter describes OpenCyc, probably the most exciting knowledge instrument today, and provides examples of using the CycL language and OpenCyc engine in distributed knowledge systems.

I hope this chapter does not take you, my reader, by surprise. Integration-ready systems and collaborative engineering *need* and *help create* knowledge technologies, which creates a very healthy cycle.

A customer with a computer and computer skills is still the main target for computerized services today. Even when searching Goggle.com for a specific topic, you need to know the specific terms of the industry this topic belongs to. This requirement prevents or hinders information exchange between different knowledge domains. The computer illiterate part of the population is almost completely excluded from the computerized service client base. There also are people with disabilities who are prevented from using computers in a general manner.

In addition, there is a “gray area” of the population who have limited computer skills but no desire to use these skills. These individuals have learned from their experience that computers are too stupid and cannot serve them well in their specific fields today. Service providers have a great reason to employ knowledge technologies and drastically increase clientele for their services.

Knowledge technologies help to create a bridge from natural language to a specific service request. For example, the Semantic Web is the representation of data on the World Wide Web based on the RDF. Another direction where knowledge technologies can be helpful is the area of speech recognition systems (SRSs). SRSs are extremely narrow in their business domains today. Current SRSs lack general knowledge representation; they direct customers into the “select one of the options” routine.

There are many methods for representing knowledge, including written documents, text files, and databases. Below, I review a few technologies used in this vast area: The Semantic Web (an umbrella for many other technologies), XML, RDF, Topic Maps, frames and slots methods, the CycL language, and others.

The Semantic Web is a vision for the future of the Web, in which information is given explicit meaning, making it easier for machines to automatically process and integrate information available on the Web. The Semantic Web will build on XML's ability to define customized tagging schemes and RDF's flexible approach to representing data. The next element required for the Semantic Web is a Web ontology language, which can formally describe the semantics of classes and properties, used in web documents. In order for machines to perform useful reasoning tasks on these documents, the language must go beyond the basic semantics of RDF Schema. [1]

Ontology

Knowledge-handling methods and terms are often called *ontology*. Ontology formally defines a common set of terms that are used to describe and represent a domain of knowledge. Automated tools to power advanced services related to knowledge management can use ontology (knowledge-handling methods and terms). Ontology is critical for applications that want to search across or merge information from diverse communities.

Ontology can provide terminology for describing content with *rules or assertions and inferences* that define *terms using other terms*. Good search engines include some ontology definitions provided for specific business areas. We can call them *specific ontologies*.

For example, a specific ontology can be created to define group memberships. This ontology might include terms such as *user*, *group*, *member*, and *role*. This ontology could also include definitions such as *groups have members*, and *every group member has a role*.

A search system that uses such ontology would take initial key data entered by a user and look for additional data required by the rules. Such a system can obtain search results superior to conventional search systems. Of course, this superiority relies on additional data provided with content annotations. Content providers must be in the game.

It is important that ontologies are publicly available and different data sources can commit to the same ontology for shared meaning. In addition, ontologies should be able to extend other ontologies in order to provide additional definitions.

XML document type definitions (DTDs) and XML Schemas are sufficient for exchanging data between parties who have detailed agreements, specifications, and an existing and stable vocabulary. At the same time, they have no semantic mechanisms to understand changing or new XML vocabularies.

RDF and RDF Schema

RDF and RDF Schema [2] begin to approach this problem by allowing simple semantics to be associated with terms. With RDF Schema, one can define classes that may have multiple subclasses and superclasses; one can also define properties, which may have subproperties, domains, and ranges. In this sense, RDF Schema is a simple ontology language.

However, in order to achieve interoperation between numerous, autonomously developed and managed schemas, richer semantics are needed. There is a need for instruments capable of sharing the knowledge across the boundaries of notation, grammar, knowledge domains, and natural language.

In RDF, each schema has its own namespace identified by a Uniform Resource Identifier (URI). The URI identifies any content presented by text, an image, or a sound file. A typical example of a URI is *http://IPServe.com*.

Each term in the RDF Schema is identified by combining the schema's URI with the term's ID. Any resource that uses this URI references the term as defined in that schema. However, RDF is unclear on the definition of a term that has partial definitions in multiple schemas. The specification appears to assume that the definition is the union of all descriptions that use the same identifier, regardless of source.

DAML+OIL: A Semantic Markup Language for Web Resources

DAML [3] was created as part of a research program started in August 2000 by the Defense Advanced Research Projects Agency (DARPA), a U.S. governmental organization. OIL is an initiative funded by the European Union Programme for Information Society Technologies as part of its research projects.

The marriage of DAML and OIL produced a semantic markup language for Web resources. The language is based on RDF and RDF Schema. DAML+OIL extends RDF capabilities with richer modeling primitives.

A few words about RDF:

An RDF document is a collection of assertions that typically begins with the tag *<rdf:RDF* and several obligatory RDF declarations that refer document prefixes (e.g., *xsd:*) to existing specifications. Each topmost RDF element is the subject of a sentence. The next level of enclosed elements represent verb/object pairs for this sentence. For example:

```
<Class ID="GroupAccount">
  <subClassOf resource="#Account"/>
</Class>
```

This means that the *GroupAccount* class is a subclass of the *Account* class.

The DAML example provided below is in effect an RDF document that includes DAML extensions. DAML extensions are easily recognizable because they have *<daml:* prefixes. The example begins with an RDF start tag including several namespace declarations:

```
<rdf:RDF
        xmlns:rdf ="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
        xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
        xmlns:xsd ="http://www.w3.org/2000/10/XMLSchema#"
        xmlns:daml="http://www.w3.org/2001/10/daml+oil#"
        xmlns:dex ="http://www.w3.org/TR/2001/NOTE-daml+oil walkthru-20011218/daml+oil-
ex#"
        xmlns:exd ="http://www.w3.org/TR/2001/NOTE-daml+oil-walkthru-20011218/daml+oil-ex-
dt#"
        xmlns     ="http://www.w3.org/TR/2001/NOTE-daml+oil-walkthru-20011218/daml+oil-ex#"
>
```

XML namespace declarations (the *xmlns* above) relate prefixes to their specifications. Therefore, in this document, the *rdf:* prefix should be understood as a reference to *http://www.w3.org/1999/02/22-rdf-syntax-ns#*. This is a conventional RDF declaration appearing at the beginning of almost every RDF document.

The second and third declarations make similar statements about *rdfs:* and *xsd:* prefixes that refer to the RDF Schema and XML Schema datatype namespaces.

The following declarations provide references for *daml:*, *dex:*, and *exd:* prefixes. These again are conventional DAML+OIL declarations.

The final declaration states that unprefixed elements refer to *http://www.w3.org/TR/2001/NOTE-daml+oil-walkthru-20011218/daml+oil-ex#* that is, the location of this document itself.

After these initial declarations, we can indicate that this RDF document *is* an ontology.

```
<daml:Ontology rdf:about="Collaborative Engineering">
```

Before we can describe our topic, we need to define some basic types. DAML, like object-oriented languages, does this by giving a name for a class.

```
<daml:Class rdf:ID="Account">
```

This assertion tells us that there is a class named *Account*. It is possible for others to refer to the definition of *Account* that we give here.

```
 <rdfs:label>Account</rdfs:label>
 <rdfs:comment>
  This class of Accounts provides a base for User and Group Accounts.
  </rdfs:comment>
</daml:Class>
```

We introduced a label for graphical representations of RDF, as well as a comment, and we closed the class definition.

There are two types of accounts, Group and User.

```
<daml:Class rdf:ID="GroupAccount">
  <rdfs:subClassOf rdf:resource="#Account"/>
</daml:Class>
```

The *subClassOf* element indicates that its subject – *GroupAccount* – is a subclass of its object – the resource identified by *#Account*.

```
<daml:Class rdf:ID="UserAccount">
  <rdfs:subClassOf rdf:resource="#Account"/>
  <daml:disjointWith rdf:resource="#GroupAccount"/>
</daml:Class>
```

The *disjointWith* element is a DAML extension of *rdfs*. This element tells us that no object can be both a *UserAccount* and *GroupAccount* in this ontology.

We can define DAML+OIL properties that relate objects to other objects or those that relate objects to datatype values.

We define the *hasGroupMembership* relation that will be used to connect a User to Group accounts.

```
<daml:ObjectProperty rdf:ID="hasGroupMembership">
```

Then we say that *hasGroupMembership* is a property that applies to *UserAccount*.

```
  <rdfs:domain rdf:resource="#UserAccount"/>
```

Like the domain, we also declare the range of the *hasGroupMembership* relation. Below, we define that the value of the *hasGroupMembership* property can only be *GroupAccount*.

```
 <rdfs:range rdf:resource="#GroupAccount"/>
```

We then close the *ObjectProperty* tag.

```
</daml:ObjectProperty>
```

Above, we effectively declared that every user could have memberships in one or more groups.

In a similar way, we can define *DatatypeProperty*.

The more sophisticated example below defines some restrictions on *GroupAccount*. For objects that have the type *GroupAccount* (subclasses), we provide property constraints. We not only specify a maximum, minimum, or precise number of values for that property, but also enforce the type that these property values must have:

```
<daml:Class rdf:about="#GroupAccount">
 <rdfs:comment>
```

Only one administrator is allowed in the group, and it must be a group member.

```
 </rdfs:comment>
 <rdfs:subClassOf>
  <daml:Restriction daml:maxCardinalityQ="1">
   <daml:onProperty rdf:resource="#hasAdministrator"/>
   <daml:hasClassQ rdf:resource="#GroupMember"/>
  </daml:Restriction>
 </rdfs:subClassOf>
```

```
</daml:Class>
```

This states that a *GroupAccount* may have at most one administrator that is a *GroupMember*.

After we define some basic types, we can create objects of these types.

```
<UserAccount rdf:ID="Alex.Nozik">
  <loginName>alex.nozik</loginName>
</UserAccount>
```

Finally, we end the document with the *rdf:RDF* closing element.

```
</rdf:RDF>
```

We can see that the RDF-based approach is very scalable and can be applied to many knowledge areas. At the same time, however, it is not rich enough to deal with natural language flexibility.

Topic Maps

Topic Maps is the ISO 13250 standard that defines a model and interchange syntax for knowledge representation with topics, occurrences of topics, and relationships – "associations" – between topics. Topic Maps can be compared to the "GPS (Global Positioning System) of the information universe," a base technology for knowledge representation and knowledge management.

Topic Maps modeling started in 1991 with the initial goal of merging information. The idea was to find a formal way for capturing information models. The scope was later broadened to multiple applications providing access to information based on a model of the knowledge it contains.

The key concepts of Topic Maps modeling are:

- Topic (and topic type)
- Occurrence of the topic (and occurrence role)
- Association of the topic (and association type)
- Scope of the topic

A topic can be any "thing" whatsoever – a person, an entity, a concept, really anything – regardless of whether it exists or not.

The term *topic* refers to the element in the Topic Map document. Topic types represent a typical class-instance relationship. Topic types are themselves defined as topics. For example, in software documentation, they might be functions, variables, objects, or methods.

Topics have three kinds of characteristics: names, occurrences, and roles in associations. There are base names (required), display names (optional), and sort names (optional). A topic may be linked to one or more information resources, called *occurrences* of the topic. An article about the topic and a picture related to the topic are examples of occurrences.

Occurrences may be of different types – for example, "file," "monograph," "article," "illustration," – generally supported in the standard by the concept of the *occurrence role*.

A topic *association* asserts a relationship between two or more topics. For example:

"The Sun Educational Services (SES) headquarters are *located in* Broomfield, Colorado."

"Java Distributed Computing (book) was *written by* Jim Farley."

"Alexander Pushkin was *born in* Russia."

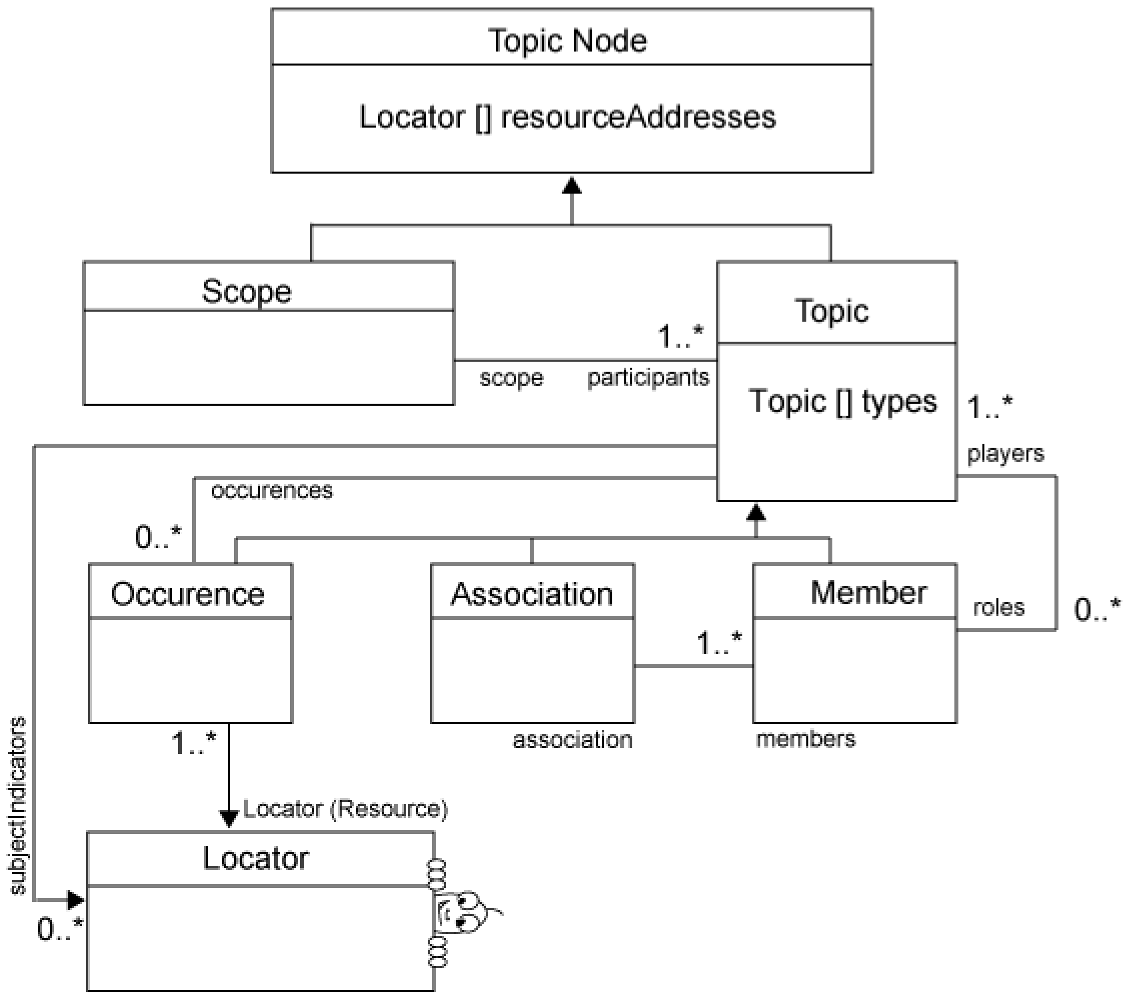
Association types, such as *located in*, *written by,* and *born in*, define relationships between topics. Topics can play different, or the same, roles in these relationships. For example, A and B have the same role in the association “A *works with* B.” However, they play different roles in the “A was *rescued by* B” association.

Scope is another characteristic of topics that limits their applicability. The same topic can be considered under different circumstances, or in a different scope. For example, when we refer to Washington, we always provide a scope for this topic. It can be a president of the U.S., a state, or Washington, D.C.

We have now considered the main characteristics assigned to topics: names, occurrences or resources, and association roles.

Topic Map representation is defined by the XTM specifications [4].

Is there any code around that supports some concepts of Topic Maps? There are several companies and open source projects [5] working in this direction. Every project has its own model that maps Topic Map concepts to its software and supports these concepts. All models are different but still have many common features. I have tried to summarize these models in a simplified version that reflects the mainstream approach to Topic Map object-oriented modeling. Fig. 5.1 displays this “averaged” version of an object model diagram that represents Topic Map concepts in most current projects.



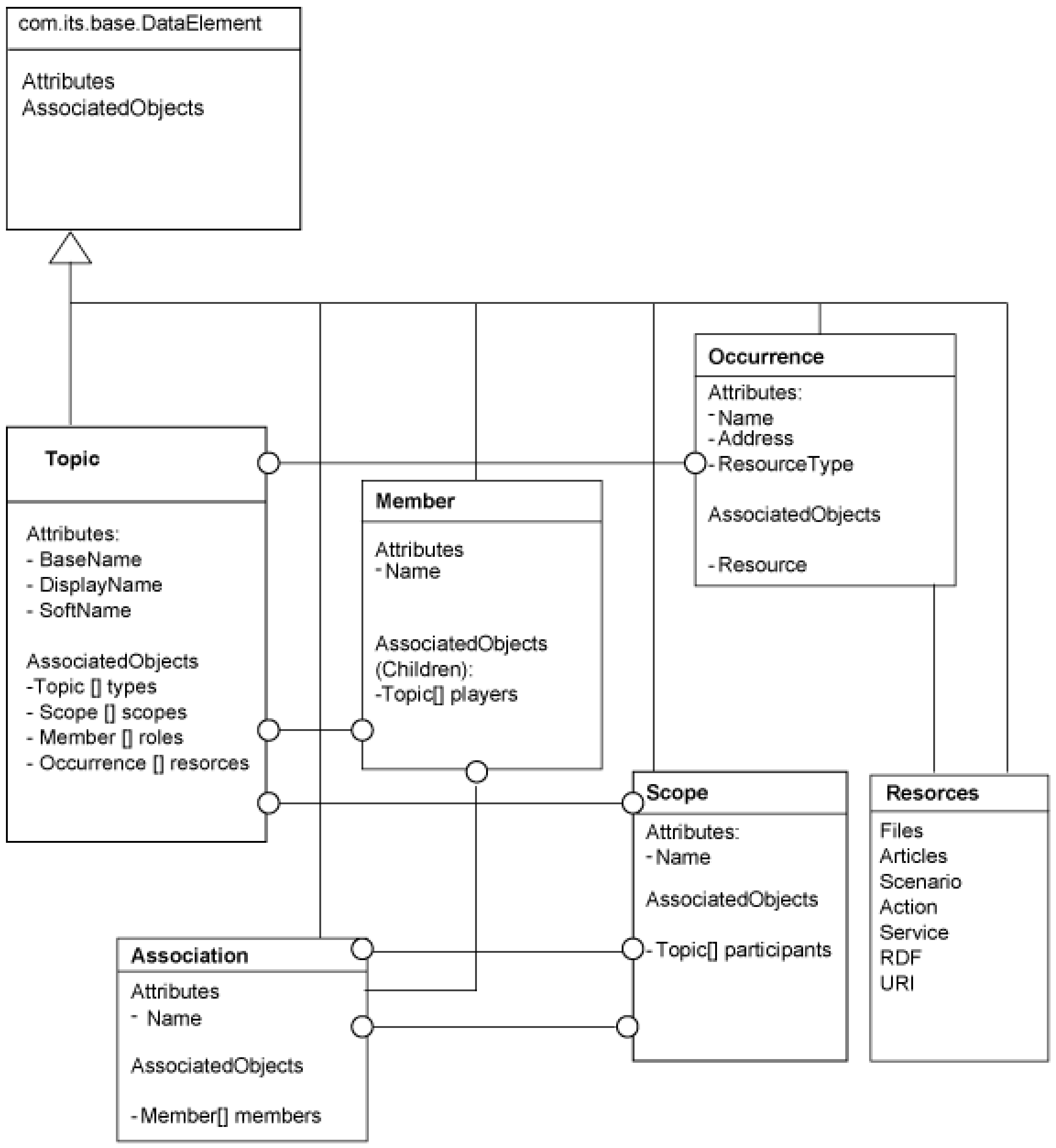
[Fig.5-1]

One of the Java implementations of Topic Maps can be found in the Topic Maps 4 Java open source project [5]. An early version of this project released around 100 classes that support handling Topic Maps.

After reviewing different models, I suggest that all Topic Map objects behave in a very similar manner and can be represented with a typed collection of objects. The *com.its.base.DataElement* class is the base for this collection, and the *com.its.base.TMService* class is able to handle the whole collection.

Object-oriented programming (OOP) enforces strong typing in OOP languages. Some developers tend to create a new class for any new data structure. This expensive practice works OK on corporate workstations; however, it quickly fails under small-device constraints. Server-side development has been steadily growing during the past decade. I expect this decade will show increasing demand for client applications for numerous devices. The typed-object approach leads to an economic programming model. This does not mean that we must deviate from OOP, we just want to be more selective in creating new classes. Object behavior is an important criterion in this choice.

The *com.its.base.DataElement* class will be considered with several different services later in this book. This class has built-in properties to provide security access and data evaluation, and generic support for business attributes and associated objects. Fig. 5.2 displays an object model diagram that represents Topic Maps with the typed-object approach.



[Fig.5-2]

Products that support XTM deliver (and are capable of reading) files with exactly the same standard format, regardless of the chosen object model and implementation details. Here is an extract of an XTM file.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<topicMap id="global knowledge container">
 <topic id=" Services">
```

```
  <baseName>Training</baseName>
  <occurrence>
   <topicRef xlink:href="http://JavaSchool.com"/>
  </occurrence>
 </topic>
...
</topicMap>
```

Knowledge management is different from information management because knowledge assumes more than just having information about a subject. Topic Maps may be considered the standard for knowledge classification, codification, and formalization.

Topic Maps and the associated syntax of XTM can represent both human knowledge and the structural relations within elements of that knowledge. Topic Maps are capable of providing the interchange of such information across the boundaries of knowledge domains. To accomplish such interchange, this technology relies on the availability of a set of rules capable of expressing the reasoning needed for knowledge classification.

Do we become a bit smarter by getting more data? Yes and no. We would appreciate information much more if it helped us predict market behavior, prevent fraud, or explain the cause of cancer or the reason for the disease called *aging*.

About 20 years ago, I had the privilege of working with a very talented, world-famous gerontologist, Dr. Tamara Dubina. At the time, she was researching a new concept of biological age. Her work related health indicators to this new concept, thus giving different perspectives on people's aging process.

(I helped with the math model and programming, using nonlinear regression methods.) [6]

Tamara collected a tremendous amount of data during this research. There was a common feeling that the volume of information did not make retrieving knowledge easier. Creating a model that could describe this data was not a trivial task. At the same time, the data were priceless for testing the model. That was a typical data mining process: analysis – model – test.

Data Mining Process and Methods

Statistics help with data analysis and give us a more focused view on the past*.* *Data mining* methods look for patterns hidden in multiple data records and help build a model that can actually provide some insight into the future. For example, MatchLogics, Inc., one of the early Internet successes, used data mining to understand Internet users and offer them the right products.

The data mining process consists of several steps.

1. Collecting data.
2. Sorting and filtering data for modeling.
3. Building a model.
4. Testing the model on another set of data. Testing the model on the same data (which we used to create the model) proves nothing.
5. Tuning/fixing/redoing the model, based on the test results; then return to step 4 or (if the test shows great results) move on to step 6.
6. Applying the model to real data and looking into the future.

Some models change rarely, some often. Most day traders, for example, may not have enough time to create a working model.

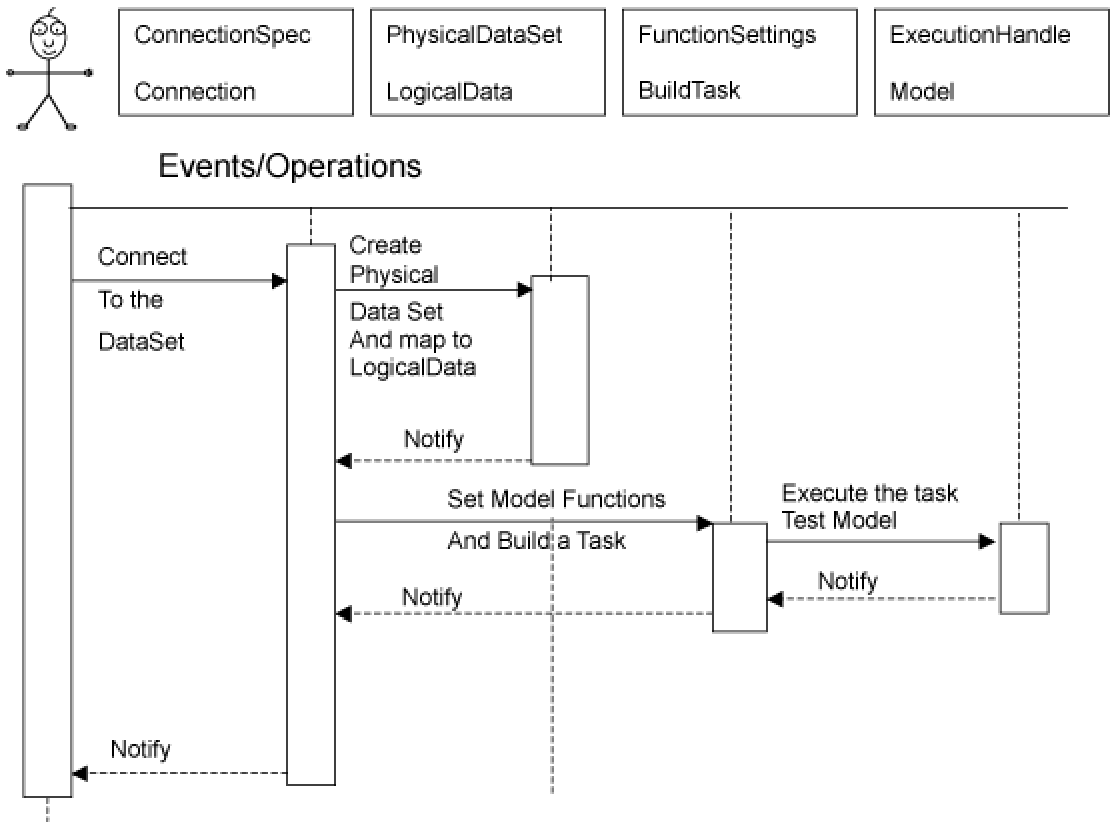
One of the specifications provided in the data mining area belongs to the Java community. Java Specification Request (JSR) 73 [7] identifies the main specification objectives:

1. Provide access to data mining systems in a vendor-neutral manner.
2. Make it accessible to non–data mining experts.
3. Provide a set of functions and algorithms.
4. 4. Target the J2EE platform with consistent interface to JCX – Connector Architecture (JSR 16); JMI – Metadata Interface (JSR 40); and JOLAP – Online Analytical Processing (JSR 69).
5. Provide compatibility with existing data mining standards: CWM DM – Common Warehouse Metadata; PMML – Predictive Model Markup Language; SQL/MM for DM – ISO SQL (structured query language) standard.

An application programming interface (API) with supported Java class implementations will glue client applications to the data mining engine and metadata repository.

The main data mining methods start with data classification and approximation functions and continue with association rule discovery and attribute importance evaluation. Data mining software captures high-level specifications for model building. The software must be able to specify the algorithm of data approximation (or regression) using association rules, a decision tree, or another specific model. The target client can be an expert or a novice user.

How does software help produce a model from mostly numeric data? There are several typical operations: Connect to a data set. Map physical data to logical data (a set of logical attributes used as input to model building). Set functions for a future model. Build, test, and apply the model. Fig. 5.3 shows the sequence diagram, and Fig. 5.4 provides an example with several lines of code using *javax.datamining package* classes.



[Fig.5-3]

```
import javax.datamining.*;
```

```
// connect to the DataSet1
 ConnectionSpec connectionSpec =
connectionFactory.createConnectionSpec(?DataSet1", ?Mark",
?pswd");

Connection dataSetConnection =
connectionFactory.getConnection (connectionSpec);

// Create PhysicalDataSet
PhysicalDataSet dataSet = new PhysicalDataSet
(?http://JavaSchool.com?);
dataSet.getMetadata();


// Create LogicalData and map to Physical data
LogicalData logicalData = new LogicalData(dataSet);

// Set Model Functions
FunctionSettings modelSetting =
new ClassificationSettings (logicalData, ?servicePrice?);


// Build the task, execute the task, test model
BuildTask testModelTask = new BuildTask(dataSet,
modelSettings,?trainingModel?);
dataSetConnection.addObject (?testModel?, buildTask);
dataSetConnection.save();


ExecutionHandle testing = dataSetConnection.execute(testModelTask);
testing.waitForCompletion ();

Model model = (Model) dataSetConnection.getObject (?testedModel?);
```

[Fig.5-4]

Be aware that the *javax.datamining* package is not released at this time, and the final release version may have some syntax differences.

Data mining methods expressed in object-oriented language help us understand numbers, build models, and predict future numbers.

Probably the most difficult task is to understand people and natural language, and to retrieve knowledge from textual and spoken information. This is different from a search for textual information. I will review the method of frames and slots that is currently used in the Dialog Manager (Speech Recognition) product to parse natural language. Then, I would like to introduce you to the CycL language, which from my point of view, is the most powerful instrument created for building a bridge between computers and natural language.

Following are a few techniques that are currently dealing with natural language.

Frames and Slots

Frames and slots are very convenient for representing *domain* information. A frame has a name and a set of slots. Each slot is a concept hierarchy with the slot name as the root. For example, CU Communicator with its Dialog Manager [8] engine is a product based on the frames-and-slots method that offers a library of functions (parsers) for manipulating frames. Information is extracted from parses into frames and is stored in frames directly by the Dialog Manager. Here is an example:

```
Frame: System_Groups
[Group1]
[Roles]
[Role1]
[RoleN]
[Available_Services]
[Service1]
[ServiceN]
;
```

The application developer creates a task file, which is similar to a frames file for the parser. The task file contains:

- The definition of the system ontology
- Templates for prompting for information
- Templates for verifying information
- Templates for generating SQL queries

Dialog Manager offers a set of standard canonical functions for dates, times, and numbers.

When Dialog Manager receives a parse, it calls the function *extract_values()* to extract information from the parse into frames. The *extract_values()* function scans the parse for any token names that are in the canonical function table. When it finds a token in the table, it calls the function associated with the token, passing it to a pointer to the input string, starting at the token. The function rewrites the input string starting at that point.

After information is extracted and merged into the context, the function *action_switch()* is called to determine the next system action. This function examines the context and takes an action based on a prespecified order of priorities.

The frames-and-slots method, as well as the products (like Dialog Manager) that are based on this method, have their advantages and limitations. The method is relatively simple and works OK in a single domain, but it is hard to extend across domain borders to the level of generic knowledge.

Systems based on this method currently have a good business standing, as they are early in the speech recognition market. The proprietary technology (with no mainstream standards) used in most frames-based systems locks current clients into a single-vendor schema. It also limits knowledge-base development by current client business domains with very modest growth of generic knowledge data. This may discourage many customers as competition from VoiceXML-based systems (e.g., BeVocal, Nuance, SpeechWorks, Tellme) grows and XTM-based data become available to the public.

The CycL Language

The CycL language has been in development by Cycorp for almost two decades, but only recently has it been made available to the public in the OpenCyc project [9]. I would like to thank the ontology experts from Cycorp who helped me by providing their insight into this new and exciting technology: Jen Headley, Doug Lenat, John De Oliveira, Steve Reed, Keith Goolsbey, Jon Curtis, Michael Witbrock, Roland Reagan, Kevin Knight, Douglas Foxvog, and Tony Brusseau.

The advantage that the Cyc method has over methods considered earlier is that Cyc has a language that is capable of expressing knowledge: CycL. In CycL, the meanings of statements and inferential connections between statements are encoded in a way that is accessible to a machine.

Presently, natural languages are virtually meaningless to machines. I can say, “All system users have at least one login. All system users are people. Jeff is a system user.” From these sentences, a *person* can infer that Jeff has a login, but a *machine* cannot, at least not until a machine can understand English sentences using some common sense.

In the formal language Cyc uses, inference is reduced to a matter of symbol manipulation, something that a machine can do. When an argument is written in CycL, its meaning is encoded in the shape, or symbolic structure, of the assertions it contains. Determining whether or not an argument is valid can be achieved by checking for certain simple physical patterns in the CycL sentence representing its premises and conclusions.

One issue in the choice of representations is expressiveness. It is impossible to express the complicated realities of life with programming language primitives. Yet, somehow we do this every day. We create multiple abstractions – filters that finally break down the complexity – and often disconnect the final product from our initial ideas. Natural language would be ideal, but not for machines that cannot tolerate conflicts created by the language. An “almost natural” language like CycL is a better choice. It allows great flexibility in creating new expressions while preserving non-conflicting rules and data.

Yes, we want a great deal of expressiveness. We would like to create a kind of ideal comprehensive system. Does this mean we should use natural language for such a system?

The expressiveness of natural language, though, goes beyond the minimum of complexity we would like to introduce for this task. The expressiveness of natural language also gives rise to special problems if one wants not only to store, but also to reason with, the represented knowledge. Logic-based representation, in contrast, gives us enough expressiveness, and facilitates the reasoning as well.

Natural language is obviously very expressive, too expressive to be formalized for the machine. Consider the following sentences.

Jeff’s failure resulted from his error.

Jeff’s error caused his failure.

Jeff’s failure was a consequence of his mistake.

Jeff’s mistake occurred before his failure.

Each of these sentences means roughly the same thing, and each implies that Jeff’s error occurred before his failure. If we want to represent that implication, do we write a rule for every natural language expression that could possibly express this point?

CycL is a logic-based language that offers a simplified, more efficient approach. First, we identify the common concepts – for example, the relation “error caused failure.” This is a very common relationship for English sentences. Then, we formulate rules about those common concepts. For example, “if error caused failure, then error temporally precedes failure.”

Another issue in the choice of a knowledge representation language is ambiguity. Natural language is highly ambiguous. For example, if we say, “Steve is running fast,” we don’t know whether Steve is changing location, operating a piece of machinery, or running as a candidate for office. On the other hand, with a logical representation, we can precisely define the concepts we use. We can, for example, define a distinct concept corresponding to each of these three senses of “running.” This allows us to place the appropriate rules on their respective concepts, whereas they could not all be placed on the one ambiguous word.

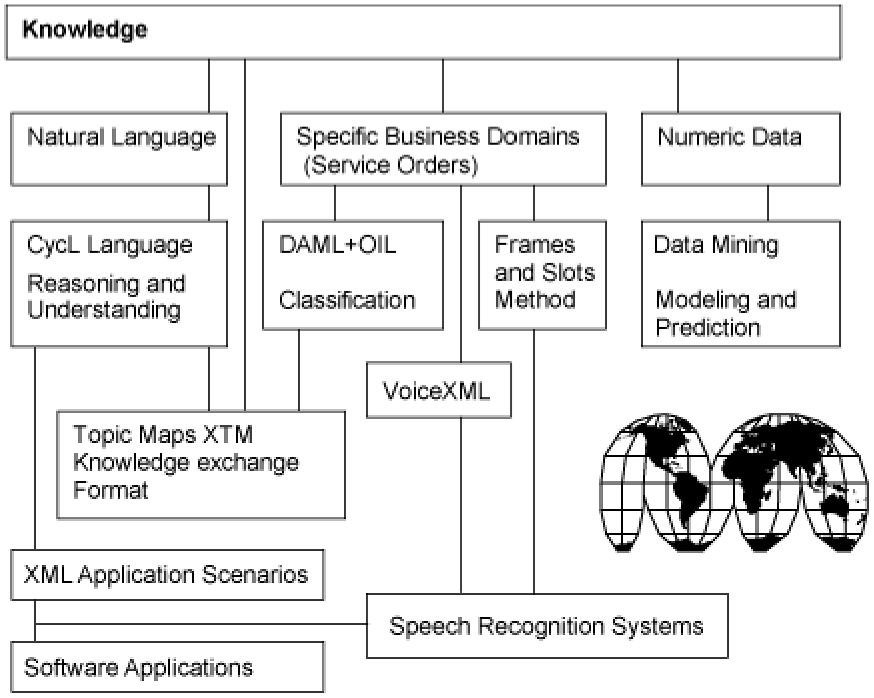
Cyc Technology: Current Status and Projections

The CycL language is probably the most advanced instrument for general knowledge capture and processing. After almost two decades of development sponsored by the U.S. government, IBM, and others, Cycorp is opening the language, sources (partially), and knowledge base (partially) to the public in the OpenCyc project. This opening is increasing the number of current Cyc clients, accelerating Cyc technology development, and providing the potential for rapid growth of the Cyc-based generic knowledge base.

Cyc technology is not yet a standard. However, the bridge from Cyc to XTM delivers the promise of a standardized interchange of such information across the boundaries of knowledge domains, computer notation, and even natural language.

Today*,* Cyc is the best technology for building a generic knowledge-based product that provides a bridge for non–computer literate users to talk to computer services. Cyc itself is not a speech recognition system; another layer must be provided to include Cyc in such systems. This additional layer can be made with mainstream technologies (based on SALT [Speech Application Language Tags], VoiceXML, or Java Speech API standards) integrated with knowledge services. Such integration can occur in the XML-based scenarios we will discuss a bit later.

Fig.5.5 illustrates relationships between knowledge, XML, and speech technologies.


Knowledge
Natural Language
Specific Business Domains (Service Orders)
Numeric Data
CycL Language Reasoning and Understanding
DAML+OIL Classification
Frames and Slots Method
Data Mining Modeling and Prediction
VoiceXML
Topic Maps XTM Knowledge exchange Format
XML Application Scenarios
Speech Recognition Systems
Software Applications

Examples:

Cycl Language

```
(genls PublicAccess AccessType)
(genls ForGroupMembers AccessType)

(implies
 (and
   (isa ?READ Access)
     (performedBy ?READ ?USER)
     (performedOn ?READ ?OBJECT)
     (hasAccessType ?OBJECT ForGroupMembers))
 (isa ?USER GroupMember))
```

XTM example

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<topicMap id="global knowledge container">
 <topic id="Data and Services">
  <baseName>Training</baseName>
  <occurrence>
    <topicRef xlink:href="http://JavaSchool.com"/>
  </occurrence>
 </topic>
...
</topicMap>
```

DAML+OIL

```
<rdf:RDF xmlns:rdf ="http://www.w3.org...
...
<daml:Ontology rdf:about="System Users">
<daml:Class rdf:ID="UserAccount">
  <rdfs:subClassOf rdf:resource="#Account"/>
  <daml:disjointWith rdf:resource="#GroupAccount"/>
</daml:Class>
...
</rdf>
```

Voice XML

```
...
<prompt>
 <audio>
  Please select one of the following service options:
  1 - Check messages
  ....
 </audio>
</prompt>
<nomatch>
 <audio>
  Sorry, I didn't understand that.
 </audio>
 <reprompt/>
</nomatch>
...
```

[Fig.5-5]

What Is CycL? How Hard Is It to Learn and Use This Promising Language?

First, Cyc is very different from frame-and-slot systems, in which creating new rules and vocabulary would be considered expensive.

Cyc encourages the expression of complex problems and ideas using more vocabulary and simple rules. In CycL:

Creating collections is not hard and is relatively inexpensive.

Creating functions is not expensive.

Creating predicates is easy and cheap.

Adding new vocabulary and microtheories is *not* expensive.

The Cyc language consists of:

Constants – *#$Login*, *#$GroupMembers*; denote individuals or collections.

Predicates – *#$likesAsFriend, #$bordersOn, #$objectHasColor, #$isa*

Logical connectives – *#$and*, *#$or*, *#$not*

Quantifiers – *#$implies*, *#$forAll, #$thereExists*

Sentences – *#$isa*, *#$Simon.Roberts*, *#$SystemUser*; form assertions or queries. The assertion in this example says that *Simon.Roberts* is an instance of *SystemUser* collection.

Denotational functions – *#$LoginFn Simon.Roberts*; relations that can be applied to some arguments to pick out something new. For example, we can interpret the formula

#$password (#$LoginFn #$Simon.Roberts) #$cessna172 as “The password in the login function for Simon Roberts is cessna172.”

Microtheories – *#$HumanActivitiesMt, #$OrganizationMt, #$JavaSchoolMt*; bundle assertions together based on time, space, or anything else that can help in knowledge organization.

Is CycL Flexible Enough to Express Complicated Logic? Can We Build Efficient Systems Based on the Language?

I would like to try to resolve these questions, which also troubled me when I started with CycL.

Is this language flexible and extensible enough to express complicated logic related to natural language? If so, can this language be efficient? Does it have validation mechanisms? (Natural language does not have validation rules, which is one of the reasons it is not suitable for computers.)

Working with CycL, I found positive answers to both of the questions above. Let us start with CycL validation mechanics.

CycL allows (but does not require) us to specify the number and types of arguments for any sentence type. This is very similar to regular programming languages so loved by computers.

How does Cyc deal with possible contradictions between existing and new knowledge? It is not feasible for Cyc to reconsider every assertion in its knowledge base every time we add new data. Cyc's truth maintenance system (TMS) and its argumentation method help Cyc deal with this issue. (I will talk more about TMS later.)

Here is an example of how CycL establishes rules for creating expressions. It is very important (especially for a machine) that there are rules and that every expression can be validated against these rules. Let us say we want to create a new predicate.

There are a couple of important features that every predicate and function has. The first is *arity*. Arity has to do with how many arguments a predicate or function requires, in other words, how many arguments to which you have to apply the function at a given time to result in a meaningful sentence or term. The second feature is the notion of *argument type*, which has to do with what types of things a predicate or function requires as a particular argument.

Arity is the number of argument places a predicate or function has. It is expressed in CycL in two ways:

1. The predicate *#$arity*

```
(#$arity  #$GroupFn 1)

(#$arity  #$loginPassword  2)
```

2. The collections

```
#$UnaryPredicate,  #$UnaryFunction, #$BinaryPredicate,  #$BinaryFunction, etc.

(#$isa  #$GroupFn  #$UnaryFunction)

(#$isa  #$loginPassword  #$BinaryPredicate)
```

Arity, as you know, refers to the number of argument places that a particular predicate or function has. There are two ways to express the arity of a particular predicate or function in the CycL language.

First, we have a predicate, *#$arity,* which you can apply to any relation – in other words, any predicate or function – in conjunction with a numeric value to denote how many arguments that relation accepts. For example, *(#$arity #$GroupFn 1)* denotes that the *#$GroupFn* function accepts only one argument.

The *#$loginPassword* predicate has an arity of two; thus it takes two arguments at a time.

Most relations in CycL have low arities (a low number of arguments); in fact, most have just one or two as their arity*. Remember that Cyc encourages simplicity!* Some relations take three or four arguments, a few take five arguments, and a very small number take more than five arguments. Seven is probably the highest arity used, although in principle, arity could be any number. Try to keep them on the low side.

Yet, there are few instances of *#$UnaryPredicate* in CycL. Instead, unary properties are usually represented either as *#$Collections* or *#$AttributeValues*. For example, see the *#$TeamLeader* collection below:

```
(#$isa  #$JeffZhuk  #$TeamLeader)
```

This is the recommended way, rather than the new predicate *#$teamLeader*, as in the assertion below:

```
  (#$teamLeader #$JeffZhuk)
```

There are a lot of unary functions, but very few unary predicates. The reason for this has to do with the Cyc inference engine and certain facts about how it works most efficiently. There are alternative ways to express what you might think of intuitively as a unary property.

I mentioned before that Cyc has its own way to maintain logical data consistency. Let us say we want to add an argument to an existing assertion. This action would immediately trigger the TMS's argumentation on this assertion. If this assertion changes its value from true to false or otherwise, Cyc looks at all the assertions supported by the newly changed assertion.

The TMS does not add new deductions or assertions, it only changes or removes them. The changing is done in such a way that infinite oscillation is impossible. If the change removes the last argument from an assertion, the assertion now has a truth-value of "unknown” and is removed from the knowledge base.

For example, someone removes the assertion “JeffZhuk is a person”. The TMS will trigger an investigation of other assertions that are based on the one just removed. All found related rules will be removed. For example, the rule “spouse of JeffZhuk is Bronia” will be removed if TMS maintains the rule that only a person can have a “spouse”.

What Is the Basic Structure of the Cyc Knowledge Base?

The knowledge base comprises a massive taxonomy of concepts and specifically defined relationships that describe how those concepts are related. The context of the knowledge is arranged by degrees of generality, with a small layer of abstract generalizations at the top and a large layer of real-world facts at the bottom.

A very powerful and simple CycL constant helps to create unlimited hierarchies. To express that one collection is subsumed by another, we use the CycL constant *#$genls*. A formula of the form below means that every instance of the first collection, *GroupMember*, is also an instance of the second collection, *SystemUser*.

```
(#$genls  #$GroupMember  #$SystemUser)
```

In other words, *SystemUser* is a generalization of *GroupMember*.

Most abstract concepts belong to the highest layers of Cyc knowledge base hierarchy. Real and specific concepts and facts belong to lower levels of Cyc knowledge base structure. We can roughly separate the Cyc knowledge base into four layers.

1. The upper ontology – abstract layer
2. Core theories
3. Domain-specific theories
4. Ground-level facts

The highest, abstract layer is called the *upper ontology*. The upper ontology layer does not say much about the world at all. It represents very general relations among very general concepts. For example, it contains assertions to the effect that every event is a temporal thing, every temporal thing is an individual, and every individual is a thing. “Thing” is Cyc’s most general concept. Everything whatsoever is an instance of “thing.”

The next knowledge base layer is called *core theories*. Thies layer contains several core theories that represent general facts about space, time, and causality. These are the theories that are essential to almost all commonsense reasoning.

Domain-specific theories are more specific than core theories. These theories apply to special areas of interest, such as group security policies, the service request structure, sentence types, and dialog management rules. These are the theories that make Cyc particularly useful, but they are not necessary for commonsense reasoning.

The final layer contains what is sometimes called *ground-level facts*. These are statements about particular individuals in the world. For example, “Kathy started a session” is a specific statement about one person. Generalizations would not go here; they would go in a higher layer. Anything you can imagine as a fact or a headline in a newspaper (the two are not the same, of course) would probably go in ground-level facts.

What Is the Syntax of CycL? Constants and Predicates

CycL tries to model the world in terms that most people know and understand*.* Its constants are the "vocabulary words" that represent collections of concepts.

For example, *#$ComputerService* represents the set of all computer services, or *#$ServiceAction* represents all possible actions provided by a service. Each constant has its own data structure in the knowledge base. The data structure includes (besides the constant itself) the assertions (statements) that describe this constant.

Imagine that we want to express an idea that email belongs to computer services. We tell CycL:

```
(#$isa #$Email #$ComputerServices)
```

We read this sentence as, “Email is an instance of computer services.”

What is *#$isa* in this sentence? In knowledge terms, it is a predicate. Predicates establish relationships between objects. Other predicate examples are *hasFriends* and *accessType*. We form sentences by applying predicates to some arguments. For example:

```
(#$isa #$VoiceTechnology #$TrainingCourse)
```

In this sentence, the predicate *#$isa*, which means, *"is* an instance of," is applied to the arguments *#$VoiceTechnology,* which relate *VoiceTechnology* to *#$TrainingCourse*, which denotes the collection of all training courses. The resulting sentence says that *VoiceTechnology* is an instance of a training course.

CycL Has Functions

Here is a definition for the function *#$MemberRoleFn:*

```
(#$arity #$MemberRoleFn 2)
(#$arg1Isa #$MemberRoleFn #$User)
(#$arg2Isa #$MemberRoleFn #$Group)
(#$resultIsa #$MemberRoleFn #$GroupRole)
```

We read this function definition as “the *MemberRoleFn* function has two parameters: user and group.” The function returns a specific role that the user plays in the specified group.

Functions differ from predicates. Functions return a Cyc term as a result. Accordingly, function definitions describe not only the number and types of arguments (e.g., predicate definitions) but must also describe the type of the result to be returned using the predicate *#$resultIsa*.

Functions with fixed arity are similar to predicates in that the definition of the function must specify the type of each argument using the predicates *#$arg1Isa*, *#$arg2Isa*, and so forth. Functions without fixed arity are defined using the predicate *#$argsIsa*, which specifies a single type of which every argument must be an instance.

Use Variables and Logical Connectives to Create New Rules

CycL has variables. Variable names begin with a question mark and are written in capital letters: ("?OBJECT") or (“?X”).

Creating rules in CycL is easy. I will do it right now with the *#$implies* keyword.

```
(#$implies
 (#$and
  (#$hasMembershipIn ?USER ?GROUP)
  (#$hasRole ?USER ?GROUP #$Admin)
  (#$hasPrivilege ?USER ?GROUP #$ChangeMemberRoles)))
```

This rule says that if a user has membership in a group and the user has the role of an administrator in this group, the user has the privilege of changing member roles in this group. Creating rules in Cyc is *not* expensive.

#$implies, *#$and* as well as *#$or*, and *#$not* are the most important logical connectives in CycL.

Assertions and Microtheories

After a new sentence is successfully inserted (or asserted) into the Cyc knowledge base, it is stored as an assertion. Every assertion belongs to one or several microtheories.

A grouping mechanism that is an improvement over functions is offered by CycL microtheories. Microtheories offer an assertion grouping mechanism.

A microtheory provides an umbrella over several assertions. Microtheories enable better knowledge base building together with better and more scalable inference.

Microtheories focus development of the Cyc knowledge base and enable shorter and simpler assertions.

Under a microtheory umbrella, we can provide a set of short assertions instead of a single complicated one.

Here is an example:

Mt: DataAccessMt

#$isa	*#$Read*	*#$AccessType*
#$performedBy	*#$Read*	*#$AlexNozik*
#$performedAt	*#$Read*	*#$08/07/2002-23:30:56*

In this example, *Mt: DataAccessMt* is a common name (umbrella) over several assertions.

Microtheories also allow us to cope with global inconsistency in the knowledge base. In building a knowledge base of this scale and covering different points of view, different times and places, different theories, and different topics, some inconsistency is inevitable. Inconsistencies, however, can make accurate reasoning impossible. Using microtheories, we can isolate terse assertions like the one above from others with which they might be inconsistent, and reason within consistent bundles.

We can allow inference to focus on the most relevant assertions and those that share the current assumptions.

Can Cyc Understand the Concept of "Events"?

Yes. CycL has a collection of Events.

Events are represented as individuals that:

- Have components (are not empty in space or time)
- Are situations
- Have temporal extent
- Are dynamic

Events are classified in Cyc collections such as those below:

#$Reading, *#$SalesActivity*, *#$Communicating*, etc.

Events in Cyc belong to a collection called *#$Event*. Events have components or stretches of space or time. They are also *situations*. The situation can be any configuration or arrangement, such as a set of objects, a specific place, or a specific time.

Events have temporal extent: they occur over time. Events are also dynamic: they can change over time.

This is really just the tip of the iceberg. There are many more specializations of *#$Event*. The *#$Information-TransferEvent* collection can be very useful in describing computer system tasks. The *#$Information-TransferEvent* collection has specializations, such as *#$Communicating* and *#$Reading*.

Why do we reify (store) individual events (instances of *#$Event*) in Cyc? If our knowledge about an event changes, having a reified (stored) data structure to represent the event enables us to add information or alter the representation in Cyc very easily.

Events are related to each other in the *#$genls* hierarchy. We can use that hierarchy to inherit knowledge downward from the more general types of events to the more specialized types of events.

For instance, if we have the general event collection *#$UserSessionEvent* and we state that this collection consists of *#$UserSessionInput* and #$*SystemSessionResponse*, Cyc will know that this is also true of specializations of *#$UserSessionEvent*, such as *#$ToddGreanierSessionEvent* or *#$ToddGreanierSessionInput*.

How Do We Attach Events to the Things Involved?

Many things can be components of events. Events can have performers, and there can be devices that performers use during the events. Events can have subevents, or substages. Events can occur at places, and those places are somehow involved in the events.

Events take place at certain times, and times of events are also somehow involved in events (we have special predicates to relate times to events). We state how components of events are involved in events with *role predicates* – predicates that are instances of the collection *#$Role*.

In CycL we use special predicates called *roles* to relate reified events to their components. There is a lot of knowledge built into the construction of role predicates to help Cyc understand how these roles function to relate components of events to reified events.

Roles have a hierarchy that extends Cyc's ability to reason about the components – the participants and subparts – of events.

Roles are specialized predicates developed for relating components of events to events. There are two general specializations of the collection *#$Role*: *#$ActorSlot* and *#$Sub-ProcessSlot*.

Roles are arranged in a predicate hierarchy based on *#$genlPreds*. The top node of the hierarchy is *#$actors*. Every instance of *#$Role* is a specialization of *#$actors*.

These CycL examples show the roles in the conversational events during a user session:

```
(#$performedBy #$Reading003 #$MashaTishkov)
```

MashaTishkov performs *Reading003*.

```
(#$informationRequested #$Reading003 #$ListOfUsers)
```

The information requested in *Reading003* is the *ListOfUsers.*

Here is an example of a CycL rule that captures general knowledge about roles, including knowledge about the kinds of things that are related by certain roles.

```
(#$implies
  (#$and
    (#$isa ?READ #$Reading)
         (#$informationOrigin ?READ ?OBJECT))
  (#$isa ?OBJECT #$TextualMaterial))
(#$implies
 (#$and
   (#$isa ?READ #$Reading)
       (#$performedBy ?READ ?USER)
   (#$hasSecurityType ?OBJECT #$GroupMembersOnly))
 (#$isa ?USER #$GroupMember))
```

The first one says, “In every instance of *#$Reading* that has a source, that information source is textual material.” A separate assertion should tell us that every instance of *#$Reading* does have an information source. In other words, “Whenever someone reads, they read text.”

The next one says, “Any reading event done on an object with security type restricted to group members only must be done by a person who is a group member.”

By the way, we do not always need to write the strange CycL-ish characters *#$*. Cyc can add them for us internally. Therefore, the example below is as valid as the example above.

```
(implies
 (and
  (isa ?READ Reading)
      (performedBy ?READ ?USER)
  (hasSecurityType ?OBJECT GroupMembers))
 (isa ?USER GroupMember))
```

All these examples demonstrate CycL’s unmatched capability of expressing knowledge. What can we do with CycL today?

Cyc Answers Questions

We can ask Cyc questions by creating three types of queries:

1. Ask – general-purpose query
2. Prove – conditional query
3. Query – either of the above

For example, we can have a query in the microtheory *2003ScheduleMt*

```
(groupMember XML-TrainingClass ?WHO)
```

This query is a request to generate a set of names from the list *XML-TrainingClass* according to the *2003ScheduleMt*.

An example of an answer is provided below:

```
((?WHO . ScottDennison))
…
…
```

These small examples may lead you to the wrong conclusion that we can do these same basic operations with almost any database. This is not exactly true. The difference will be more visible when you try to express more complicated problems, with many factors that must be taken into consideration, in multidimensional criteria space.

The core CycL algorithm treats the inference problem as a search through proof-space for a satisfactory resolution of a particular query. Each inference step in the search is a single supporting formula in the eventual proof.

We would appreciate the very rich expressiviness of CycL and power of the core CycL inference engine algorithm when dealing with such problems.

How to Begin with OpenCyc

If you do not have Java 2 installed on your machine, please install it now. Then download the latest version of OpenCyc from www.opencyc.org, uncompress it, and follow the instructions in the readme file.

I provide an example for a Linux system, just to demonstrate how easy it is to start.

```
tar xvfz opencyc-version.tgz

cd opencyc-version/scripts/linux

./run-cyc.sh
```

At this point, the OpenCyc server is up and running.

You can enter expressions from the command line or access the OpenCyc Web server running on your machine with your local browser. The URL is *http://localhost:3602/cgi-bin/cyccgi/cg?cb-start*.

Good luck browsing OpenCyc using the *Guest* or *CycAdministrator* account.

How to Include OpenCyc in the Bigger Picture of Your Distributed System

OpenCyc has several communication options. We already looked into the simplest options that provide access to OpenCyc directly from the command line or via a Web browser. These options are helpful in exploring Cyc's behavior. To include the OpenCyc server into your business network, use one of the following options:

- Peer-to-peer JXTA interface
- Direct TCP/IP socket communications with *org.opencyc.api.CycConnection*
- Remote TCP/IP communications via *org.opencyc.api.CycRemoteConnection* class with powerful methods like *converse(message)* and *getTrace().*

A few words about the JXTA project: JXTA is not an abbreviation. The name was picked up by Sun Microsystems from the word *juxtapose*, which means to put things next to each other. The JXTA project is Sun Microsystems' peer-to-peer technology initiative supported by Java communities.

We can look at the knowledge engine not only as a smart database, but also as a possible service brain that can add some smartness to our services. Naturally, this would require some interaction between existing services and the knowledge engine. An example might be XML-based APIs that allow user programs to request knowledge engine services.

We would also like to enable the knowledge engine to directly invoke existing services. What we actually need is two-way XML based communications from a user program to *OpenCyc* and back.

Let us start building a service-knowledge bridge that would greatly complement existing *OpenCyc* APIs.

We start with the *ServiceConnector* class that is present in Fig.5-6.java.

[Fig.5-6]

The *ServiceConnector* class invokes any service and can download additional service classes at run-time if necessary. The *ServiceConnector* class uses the service name to obtain a needed instance of a service-class and invokes a selected method-parameter on this currently acting object. The *ServiceConnector* class can be considered as an actor that can actually play multiple (object) roles.

Two of the most important methods of the class are the *act*() method and the *registerObject*() method.

The *act*() is responsible for invoking the proper method on the proper service-object**.** The *registerObject*() method stores service-objects in the table of services and helps to reuse the same service object for multiple method invocations.

Remember that service objects live their own life, and keep their state, which can have an important influence on invoked service behavior.

The *KnowledgeService* class (Fig.5-7.java) represents one of multiple services that can be invoked by the *ServiceConnector*.

[Fig.5-7]

There are no dependences between the *ServiceConnector* and services in the J2SE (standard) environment. If the application operates in J2ME, any services implemented would be known to the *ServiceConnector* interface. This would compensate for the absence of a reflection package in J2ME.

The *KnowledgeService* serves as a wrapper around the *org.cyc.api.CycAccess* methods. The KnowledgeService class uses the *org.cyc.api.CycAccess* class to communicate to the Cyc engine over TCP/IP sockets. The constructor of the *KnowledgeService* initiates access to the knowledge engine and prepares Cyc to talk. The wrapper helps to simplify and unify access to Cyc via XML based descriptions translated into hash tables.

The user can request a set of actions directly asking for “service=\”className\” “ with “action=\”methodName\” “. The user can also request service instructions from the knowledge engine via the *getIstructions*() method.

The *getInstructions*() method provides for the possibility of a scripted dialog between a user program and the knowledge engine. The set of instructions (script) can be stored in-line in the knowledge base, or the knowledge engine can point to a file with instructions.

Here is a separate example of a request to the Cyc knowledge engine to create a constant, insert (assert) assertion, query, etc.

Example:

*<act service=”KnowledgeService” action=”createNewPermanent”
constant=”JeffZhuk” />*

You can also include a request for your own (non-Cyc) service in the script.
In this case, all parameters that follow the action name will be passed as a single string to your method in your class.

A general recommendation is to pass key-value pairs to your method in a single string and provide the parameter interpretations inside your method.

Example:

*<act service=”Mail” action=”send” from=”Jeff.Zhuk@JavaSchool.com”
to=”reed@cyc.com” subject=”test” body=”testing mail service” />*

The actual communication API between *Cyc* and Java is still evolving. The basic *Cyc*-Java dialog and service operations described above are dressed into XML tags. An external user program can send XML based instructions to the KnowledgeService.

The *Cyc* response to the KnowledgeService is also XML formatted. The *KnowledgeService* can work with a simple dialog manager (in this example it is the *ScenarioPlayer* class) that implements the Scenario interface.
The Scenario interface is provided in Fig.5-8.java.

[Fig.5-8.java]

The Scenario includes several methods that provide screens to the user and accept user input. The *prompt*() method, for example, supports a pre-arranged dialog between a user and the KnowledgeService, and helps to retrieve necessary data from a user with XML based dialog scenarios.

An example of an XML based dialog scenario is provided in Fig.5-9.xml.

[Fig.5-9]

The dialog scenario in this example includes a set of questions that helps a user introduce a new fact to the knowledge engine.

The *prompt()* method (implemented in a subclass, for example the *ScenarioPlayer*) will translate a multi-line message passed as an argument into a set of questions for a user. The method will store user answers and use them according to the instructions in the message.

The instructions can prompt a user for questions or refer to user services or to *KnowledgeService* methods. The *prompt*() method will be able to replace script variables at run-time with the user's answers.

The *prompt*() method reads an XML based dialog scenario with pre-arranged questions and consecutive service-actions that can follow such questions.

The XML based dialog scenario usually includes a set (or several sets) of question sequences.

Here is an example of a question that might be repeated, as some of us have more than one favorite rock band.

```
<prompt variable="REQUESTED-SUBJECT"
msg="What do you want to know?" />
```

This XML description will invoke the *prompt*() method on a service object of the *UserAVI* (audio-video interface) class.

The service will select a proper presentation layer to show or to speak out the question to a user.

The user's answer will be stored under the variable name "*REQUESTED-SUBJECT"* and will be used in the following instruction of this scenario.

```
<act service="KnowledgeService" action="query"
msg="(?X  #$REQUESTED-SUBJECT ?Y)"  />
```

This instruction will query *Cyc* for a requested subject and translate the knowledge engine's response into the proper presentation format.

Here is a bigger example of a set of questions about your lovely rock band. The answers will be stored in Cyc under a name:

"LOGIN-NAME" + "FavoriteBand"

in the *microtheory* (that we create at run-time) with the name

"LOGIN-NAME" + "*HobbiesMt"*

The login name will be supplied at runtime as a parameter to the scenario.

Note that "What is your favorite rock band?" is the actual prompt for the user that invites her/him for the first answer. There are several assertions we make after the answer using "*createNewPermanent*" and "*assert*" methods of the *KnowledgeService* class.

Note that since JDK1.4 supports "assert" as a new keyword, the real method name for knowledge assertion must be different. It is "*assertGAF."* A mechanism of aliases implemented in the *ScenarioPlayer* treats such names properly.

```
<prompt variable="FAVORITE-BAND-ANSWER"
msg="What is your favorite rock band?" />

<act service="KnowledgeService"
   action="createMicrotheory"
Mt="LOGIN-NAMEHobbiesMt"
msg="Personal interest areas of LOGIN-NAME" />

<act service="KnowledgeService"
   action="createNewPermanent" msg="LOGIN-NAMEFavoriteBand" />

<act service="KnowledgeService" action="assert"
 Mt="LOGIN-NAMEHobbiesMt"
 msg="(#$isa #$LOGIN-NAMEFavoriteBand #$AttributeValue)" />

<act service="KnowledgeService" action="assert"
   Mt="LOGIN-NAMEHobbiesMt
msg="(#$isa #$LOGIN-NAMEFavoriteBand #$InterestArea)" />

<act service="KnowledgeService" action="assert"
   Mt="LOGIN-NAMEHobbiesMt
msg="(#$hobbies #$LOGIN-NAMEFavoriteBand \" FAVORITE-BAND-ANSWER \")" />
```

The dialog scenario includes a prompt to a user, a request to knowledge services or a request to regular services like e-mail, etc.

Example:

```
<act service="EMailClient" action="send" from="Jeff.Zhuk@JavaSchool.com"
 to="reed@cyc.com" subject="test" body="testing mail service ... etc. " />
```

This approach might help us to provide a standard way of connecting the world of services written in Java, or other languages, to the world of knowledge.

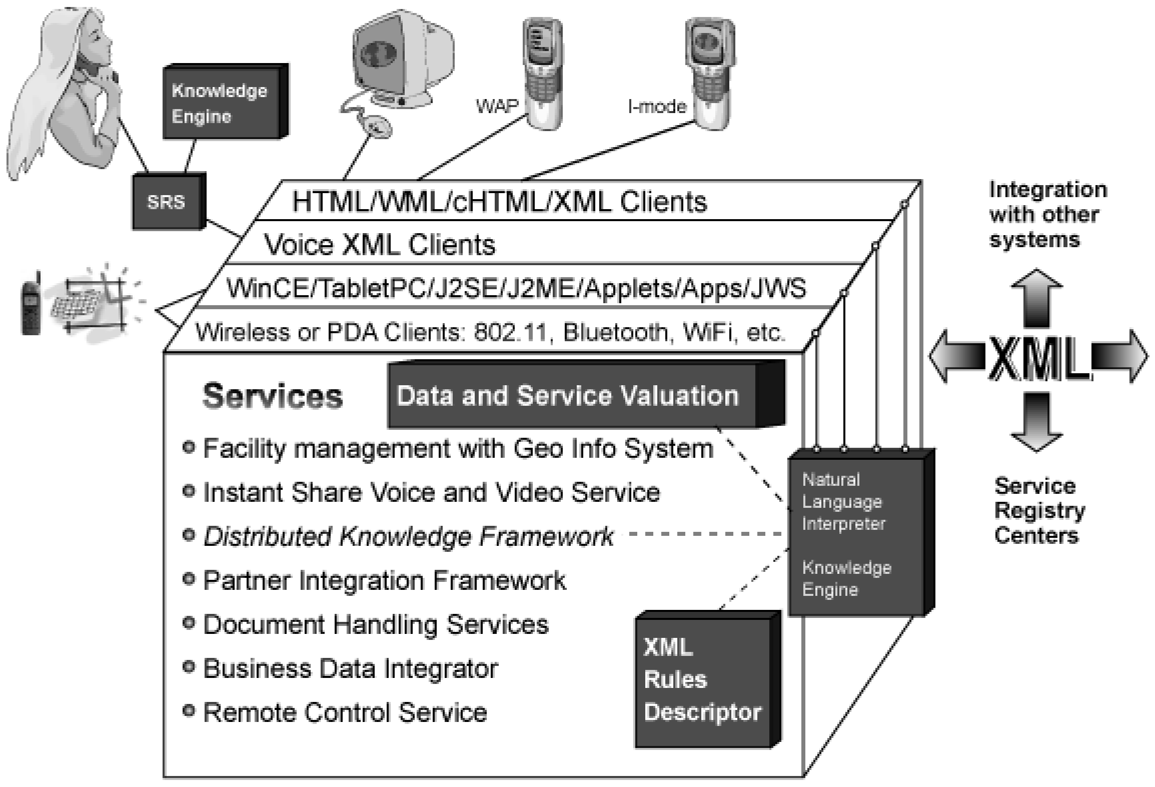
Knowledge Technologies and IT Efficiency

I started the chapter with very pragmatic reasoning, arguing for the learning and use of knowledge technologies by service providers. With these technologies, service providers increase their capacities and client base, and accordingly, their profit. From my point of view, there is an even more important change that knowledge technologies help to achieve: they elevate everyday work effectiveness.

The famous formula “write once” is not working *anywhere* today, for several reasons. One is the absence of a mechanism capable of accepting, classifying, and providing meaningful information about new data or services created by knowledge producers. We are all knowledge producers, but we almost never share what we produce with the rest of the world.

Imagine a global knowledge and service container in which everyone can *easily* find and access data and services, and can contribute and be rewarded for their contributions. (Want more details on the reward policy? Look further in the book.) We would greatly reduce work replication and redundancy, and drastically increase efficiency. Distributed knowledge systems with the Cyc engine can make this dream come true.

Existing and upcoming Cyc tools and related products multiply the powerful features of the CycL language. Cyc-NL, the natural language processing system associated with the Cyc knowledge base, brings closer the possibility of using Cyc in SRSs. Distributed knowledge systems with built-in speech recognition would help the average person, not just the computer geek, to participate in computerized knowledge and service consumption and contribution. The architecture of a distributed knowledge system with the OpenCyc engine is displayed in Fig. 5.10. The OpenCyc and the Cyc-NL interpreter are connected to presentation layers to simplify human access to data and services.


Knowledge Engine
SRS
WAP
I-mode
HTML/WML/cHTML/XML Clients
Voice XML Clients
WinCE/TabletPC/J2SE/J2ME/Applets/Apps/JWS
Wireless or PDA Clients: 802.11, Bluetooth, WiFi, etc.
Services
Data and Service Valuation
Facility management with Geo Info System
Instant Share Voice and Video Service
Distributed Knowledge Framework
Partner Integration Framework
Document Handling Services
Business Data Integrator
Remote Control Service
XML Rules Descriptor
Natural Language Interpreter
Knowledge Engine
Integration with other systems
XML
Service Registry Centers

[Fig.5-10]

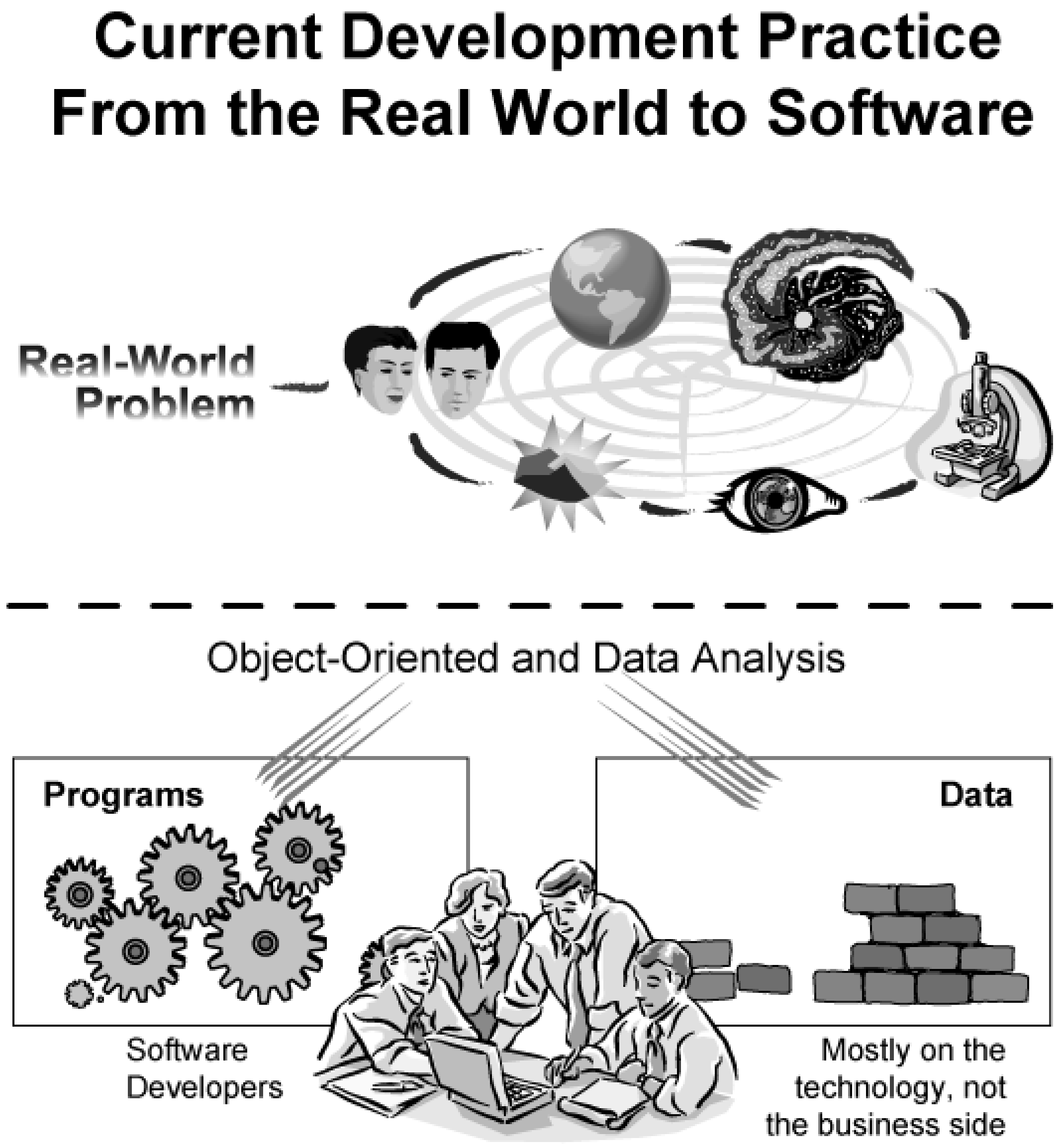
Some Intriguing Questions

How does the Cyc-NL interpreter work?

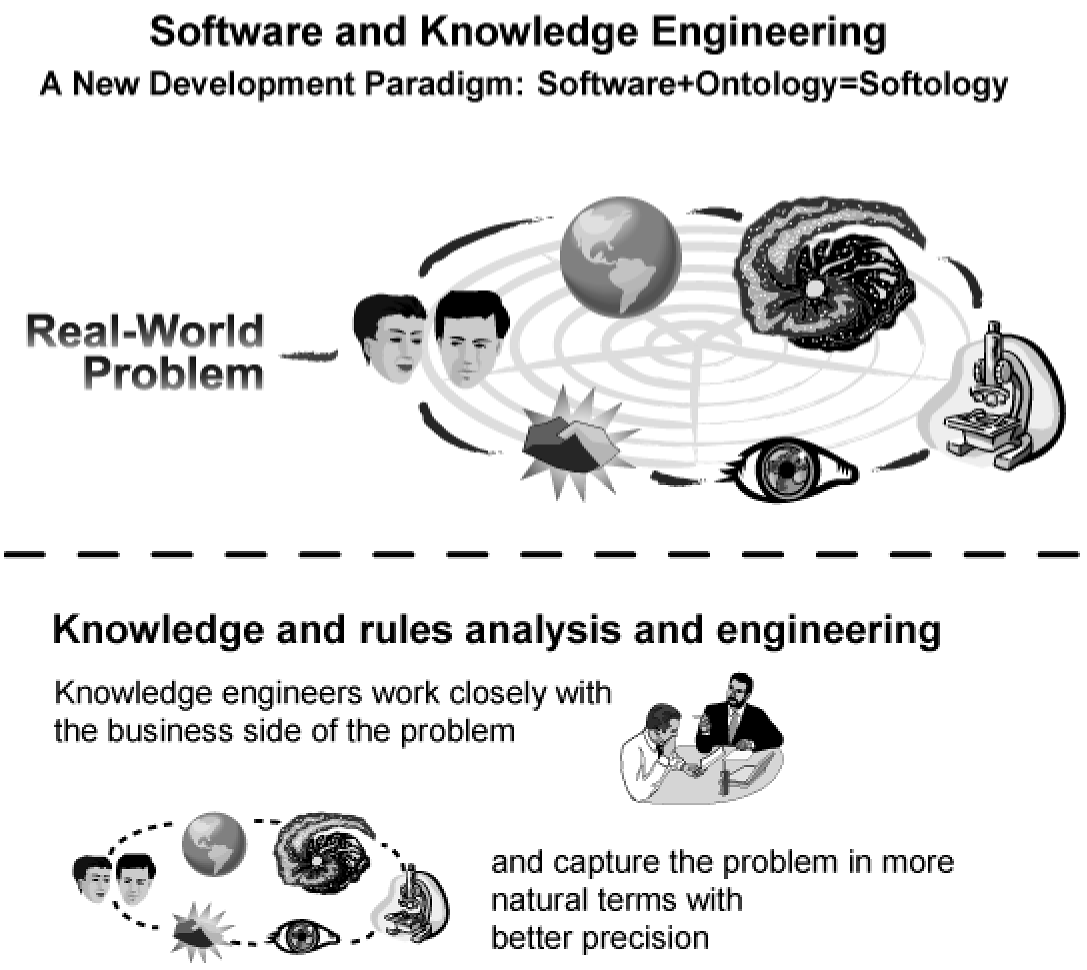
Can we plug natural language parsers directly into an SRS?

How can we use Cyc for software development, the details of knowledge to service integration, and other tasks?

For the answers, we require another chapter. This is just the beginning of a new development paradigm we can call *Software+Ontology=Softology*. Figs. 5.11 and 5.12 illustrate current and new approaches to software engineering.


Current Development Practice
From the Real World to Software
Real-World Problem
Object-Oriented and Data Analysis
Programs
Data
Software Developers
Mostly on the technology, not the business side

[Fig.5-11]


Software and Knowledge Engineering

A New Development Paradigm: Software+Ontology=Softology

Knowledge and rules analysis and engineering

Knowledge engineers work closely with the business side of the problem

and capture the problem in more natural terms with better precision

[Fig.5-12]

Current development process includes multiple teams providing multiple transformations of complexity of original business ideas into simplicity of programming functions and data tables. Business ideas can be easily lost or diluted on the way.

In the new development world, multiple layer-filters that separate business ideas from their implementations will disappear and business experts or SMEs (subject matter experts) will directly participate in the design process, working with “softology” engineers in a knowledge engine–powered environment.

Summary

This chapter teaches skills that are becoming increasingly important in the new spiral of software and business development. You learned about ontology, or knowledge-handling methods, but the subject is too broad for a complete overview. This chapter focused on standards established by the World Wide Web Consortium (W3C.org), such as RDF, DAML+OIL, Topic Maps, and XTM, and open (not proprietary) technologies, such as OpenCyc, that have great potential for the whole industry and can be immediately used in distributed knowledge systems that help connect people and organizations into knowledge federations**.**

Integrating Questions

1. What is the Semantic Web?
2. What are the main technology roles and targets of RDF and DAML+OIL?
3. Which of the knowledge technologies described in this chapter are applicable to your workplace?

Case Study

1. Create a DAML+OIL file describing email service with *Compose* and *GetMail* abilities.
2. Create an XTM file describing a hierarchy of groups of computer users and members of the groups.

3. Create a CycL microtheory describing a user's profile, and provide a query that requests a user's profile with the user's login name.
4. (Advanced). Consider the Java source in Fig. 5.6. Suggest additional Cyc keywords that can be added to the *talkToCyc()* method. Change the code that would allow you to extend this method's vocabulary at run-time.
5. Describe several related facts or rules related to your workplace: first in plain English (try to limit yourself to three to five lines), and then in CycL language.
6. Describe as Topic Maps several facts or rules related to your workplace.

References

1. Semantic Web: *http://www.w3.org/2001/sw/*.
2. Resource Description Framework (RDF) Model and Syntax Specification: *http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/*.
3. DAML+OIL reference description: *http://www.w3.org/TR/daml+oil-reference*.
4. XML Topic Maps (XTM) 1.0: *http://www.topicmaps.org/xtm/1.0/*.
5. Open Source Project, Topic Maps 4 Java: *http://tm4j.org*.
6. Dubina, T. L, A. Y. Mints, and E. V. Zhuk. 1984. "Biological Age and Its Estimation." *Experimental Gerontology* 19:133–143.
7. JSR 73, Data Mining API Specifications *http://www.jcp.org/en/jsr/detail?id=73*.
8. CU Communicator, Dialog Manager: *http://communicator.colorado.edu/*.
9. Cycorp, commonsense knowledge base and inference engine: *http://www.cyc.com*.