2008

An Object-oriented natural language expert system

Kenneth Staffan

Henry Etlinger

An Object–Oriented Natural Language Expert System

Kenneth E. Staffan and Henry A. Etlinger

December 6, 1994

RIT–TR–94–018

Department of Computer Science
School of Computer Science and Information Technology
Rochester Institute of Technology
Rochester, New York 14623
staffan@clpd.kodak.com
hae@cs.rit.edu

An Object–Oriented Natural Language Expert System

Kenneth E. Staffan
staffan@clpd.kodak.com

Henry A. Etlinger
hae@cs.rit.edu

Abstract

This paper highlights the design and implementation of a prototype object–oriented Intelligent Tutoring System (ITS). There were two main goals for the prototype. One was to create a small tutoring system which fit into the niche of current commercially available systems, while exhibiting some of the more intelligent features found in larger custom systems. The other was to experiment with an object–oriented model, design approach, and implementation for such a system. A masking technique for pattern matching is one of the key paradigms driving the intelligent diagnosis. The prototype provides a means for practicing sentence construction, by checking a student's translation responses, and offering hints, if desired. It was implemented in C++.

1. Introduction

1.1. Background

Intelligent Tutoring Systems (ITS), and other forms of Computer–Assisted Instruction (CAI), attempt to extend the resources available to a student for practice and reference. This prototype system described here attempts to fill one need in the study of a foreign language. Outside of the classroom environment, a student no longer has an expert to use for understanding or clarification. Text books and other "static" media are not always effective reference (or tutoring) tools. A dictionary, for instance, can be used if a student wishes to look up an individual word, but may be of little help if the question concerns sentence construction or other concepts.

The system described in this paper addresses the area of phrase and sentence construction. Resources are less often available in this area. To verify the correctness of a sentence, a student may have to rely on finding one of similar structure in a text book. Even this may be inaccurate, as different vocabulary may require a variation in the structure of the sentence. Sentence construction is not an advanced topic. Most language courses begin introducing some sentence forms with the very first class. Already, the best CAI packages are challenged.

The prototyped ITS experiments with a masking technique of diagnosis, using run–time generated patterns to identify the discrepancies in a student response. The system was developed using object–oriented analysis and design techniques. A fully functional (though limited in knowledge depth) system has been designed and implemented to illustrate and exercise these concepts. The system presents phrases to the student for translation, and diagnoses the responses. Hints and additional chances are offered if the student desires.

1.2. Previous Work

In the early 1960's, there was not much distinction between Computer–Assisted Language Learning (CALL) systems and other disciplines of CAI. Existing systems were mainly of the drill–and–practice variety, the equivalent of electronic flashcards. By the late 1960's, attempts were being made to use interactive feedback to control the systems' operation [Uhr, 1969]. The earliest forms of feedback–control involved merely

re–asking problems which were previously answered incorrectly. By the early 1970s, attempts were being made to adjust the difficulty of problems based on performance [Woods, 1971]. Discussion ensued on when help should be offered to a student, and whether the student or the tutoring system should determine the level of difficulty of the problems [Hartley, 1973].

By the 1980s, many projects were concentrating on the best way to represent the state of a student's knowledge, in order to optimally adapt to it [Goldstein, 1981], [Burton, 1981a–b], and [Sleeman, 1981a–b]. Research had also diverged into two distinct areas, template software for teachers to author their own computer–assisted lessons, and artificial intelligence and auto–adaptation programs which supported little, if any, teacher modification [Boyd, 1982].

Today, CALL research and CALL systems typically fall into two broad categories. The largest and most sophisticated systems are those developed and maintained by large universities for internal use. They have evolved over many years, and are highly customized to support their own language curriculums. Likewise, the most sophisticated systems in related areas (for instance, business document critiquing systems), have been developed internally by large corporations and large universities. These types of organizations have the advantage of large computing resources, and projects which slowly evolve and grow as they reach the more subtle hurdles. Another distinct advantage is that, as their own customers, the researchers in these universities and corporations have a continuous open channel for feedback, both positive and negative. At the other end of the spectrum, most commercially available CALL software is much less sophisticated. The intended customers, typically individuals and pre–college educational institutions, tend to have much more limited computing resources, and only a very small budget for educational software. In order to meet these customer constraints, and to bring such products to market in a timely manner, most of these packages address very narrow topics at the beginner level.

An example of one university effort, which has been in–progress since the early 1970's, is found at Concordia University (Canada). They began experimenting with both student– and teacher–oriented CALL programs, the main focus of which were to improve the effectiveness of their own remedial and second language English programs. Narrow topics, such as specific verb tenses, were addressed individually, with the hopes of eventually building a library of software for a large range of grammatical and syntactical topics. One of the more ambitious later projects was a "whole composition" analyzer, which attempted to diagnose student–constructed sentences [Boyd,1982]. Several notable concepts were illustrated in this project. First was the decision to check sentences for common forms of faults, rather than using a true AI analysis of the sentence for correctness. It proved that a benefit could still be realized, while narrowing the scope and the complexity of the task. Another successful concept was the extensive compiling of statistics, including comments solicited directly from students at the end of lessons, for adjusting and enhancing the diagnostics (for continuous improvement).

In the early 1980's, a similar program was being tested at Kings College (Scotland). This program was used for instruction in French, and illustrated some other successful characteristics [Farrington,1982]. Common complaints about CAI software included its lack of recognition for alternate, correct answers, and its lack of user–friendliness. The Kings College program specifically researched and hard–coded multiple correct answers to each problem, and provided hints, encouragement, and praise, if desired by the student. This was an improvement over the "display a problem", "read student response", "if incorrect, print correct answer", "display next problem" cycle of earlier drill–and–practice software. Other attention was paid to user–friendliness, giving the student control over the level of help, and the ability to exit at any time. Similar to the Concordia University program, incorrect answers were stored so that future enhancements could handle new variations of answers and new types of errors.

Several researchers have addressed alternatives to the problem of single, hard–coded answers to exercises. One possibility is to hard–code a variety or hierarchy of correct answers, as mentioned earlier. Another alternative is to hard–code common wrong answers as well as correct answers, for efficient trapping of errors [Ferney,1989]. One successful program which uses hard–coded correct and incorrect answers, the LITTRE program, pushes this technique to its limits [Farrington,1986]. The program has proven very useful, and comes across as quite knowledgeable in French. The disadvantage has been that each individual sentence requires extensive analysis before the various answers can be hard–coded.

Seventeen small commercially available German–language CAI packages were identified [Staffan, 1993].

Nearly all are variations of the drill–and–practice method (as is the system described here). Approximately half of the packages handle single–word vocabulary only, while the other half attempt in some way to address sentence–level construction. Approximately half present the exercises in a game format. Two of the packages which most directly address the same domain as the prototype (sentence construction), are discussed in further detail below.

The German Word Order program, for IBM PCs and Apple II computers by COMPress, is an interesting variation on sentence construction exercises. All sentences are hard–coded. The program chooses a sentence from its internal list, then prints the words in random order. The student must attempt to reconstruct the correct sentence. Because the student must use exactly the same words as the original sentence, parsing for correctness becomes much easier. One problem with this particular program, is that it does not accept correct sentences which do not match the original. To take an English example, "The truck hit the car" would not be accepted if the system's version of the original sentence was "The car hit the truck". This can be particularly restrictive if an alternative variation of the correct sentence retains the same meaning as the original. In this case, the student may be misled into believing that only the one form is correct.

The program gives interactive feedback while the student constructs the sentence. As each word is entered, the computer tells whether or not it is correct. This does help to alleviate the problem of alternative correct sentences (in the above example, "truck" would have been flagged as incorrect, giving the student the chance to try "car" instead). Hints are also available along the way. The student can review a brief lesson on the concept which is employed in the current sentence, or receive the correct answer, at any point.

Another program which deals with sentence construction is the Dasher program, for Apple II computers by Conduit. It begins much like the system implemented for this thesis, by presenting a sentence for translation. The difference is that it does not attempt any intelligent diagnosis or tutoring. The student response is compared letter–to–letter with the expected response (also greatly simplifying correctness parsing), and echoed back to the student with dashes (hence the name) in place of any discrepancies. This occurs a specified number of times before the correct answer is given. The re–try count is configurable.

1.3. Scope

The prototype system combines some of the concepts described above, to provide a small package supporting semi–generated phrases, intelligent critiquing of student responses, assistance to the student, if desired, and information gathering for further improvement of its capabilities.

The semi–generated phrases mark a hybrid between the entirely hard–coded phrases of such programs as the LITTRE program, and true random sentence generation. The system is programmed to diagnose specific forms, or models, of phrases and sentences. These models draw from pools of semantically correct selections, to eliminate the problem of generating nonsensical phrases. However, all phrase construction, and diagnostic pattern construction, occurs at run–time. The details of diagnosing a specific phrase are handled at an abstract level. The run–time construction, then, allows the system to correctly diagnose whatever vocabulary happens to be used for a specific exercise. Besides the programming advantage, this also makes adding phrases or sentences, of existing forms, very easy.

Externally, the tutoring system is fairly simple. It handles a single type of exercise, sentence translation, and provides diagnostic feedback. When the student logs in, the system queries the student's current knowledge level. All vocabulary and sentence models within the system are marked with the chapter in the textbook in which they are introduced. By requesting the student's knowledge level, expressed as the highest chapter completed in the textbook, the system can present only sentence forms and vocabulary which it knows the student has been exposed to. A few general user–interface rules apply. At any user prompt, the student is able to ask for help or exit the program. At any user prompt, within a specific exercise, the student is able to abort that exercise and begin a new one. Help text includes information on what is expected at the specific prompt, as well as on general rules of control. In a typical single–exercise dialogue:

- The system presents a phrase or sentence to the student for translation.
- The system reads the student's response.
- If the response is correct, the system says so, and a new phrase is presented for translation.
- If the response is incorrect, the student is asked if he or she would like to try again, receive the correct answer, or receive a hint from the system.

- If the student chooses to try again, the system reads another translation response (and attempts to diagnose it).
- If the student requests the correct answer, it is given, and the system presents a new phrase or sentence for translation.
- If the student requests a hint, the system attempts to provide a clue as to what is wrong with the response, without giving away the answer. If subsequent hints are requested, the system attempts to give successively more detailed clues.
- During the course of the exercise, statistics and information about specific responses are logged, for the purpose of future enhancements to make the system more robust.

The prototype system does not attempt to exhaustively address large numbers of sentence constructions. It was a goal, however, to implement enough variety to illustrate the viability of its concepts. The knowledge base and processing support all aspects of the example dialogue described above, plus additional detail as described later. This includes the ability to choose random, and non–repeating exercises and vocabulary. Demonstrating these features required implementing multiple phrase models, multiple vocabulary selections, multiple knowledge levels, multiple types of errors, and multiple levels of error diagnosis (to support increasingly detailed hints). The goal of our study was to establish feasibility for all of these features, as opposed to increasing the volume of similarly processed models.

Phrase models and vocabulary are selected randomly, so that the user does not know what phrase form or concepts will come up next. Additionally, during any given session, phrases presented are marked as "used" so that they do not repeat while there are still new phrases to be presented. If the entire knowledge base ends up being marked "used" (which may happen easily at lower knowledge levels), the system clears all the "used" markers, and continues drawing random selections.

The prototype system does not dynamically adapt its behavior based on individual users' previous histories, nor is it currently teacher–configurable. These would be possible future enhancements.

As mentioned earlier, an object–oriented approach was taken. The scope of this objective was to apply object–oriented analysis and design techniques to the requirements for the proposed system, and then implement the design using a language which supports object–oriented constructs. Job–Oriented Object Analysis (JOOA) [Nichols,1991] and Class–Responsibility–Collaboration Design (CRC) [Wirfs–Brock, 1990] methodologies were used.

The internal processing revolves around the adaptation of a sentence masking analysis technique [Phillips,1983] as a processing engine for the system. It, in effect, turns the knowledge–base into the diagnostic programming language of the system. The principle objective, from a software architecture standpoint, was the implementation of this technique within the object–oriented model. Most of the software development was related to integrating and illustrating the feasibility of this approach.

The dictionary portion of the system is similar to other dictionary schemes. Words and feature information are stored by the dictionary. When the system is building phrases to present to the user, it requests the variation of words that it needs by specifying the feature information. The dictionary is also used when diagnosing a student's response, to determine if incorrect words are variations of the expected word (or synonyms of, etc.) The dictionary is limited to the vocabulary that the system currently supports (based on knowledge level).

2. User Interface Overview

An example interaction with the system has been included as an appendix. It consists of excerpts from actual system runs. They are combined to illustrate the look and feel, and various system concepts. System output is shown in **bold font**, and user input is shown in *italic font*. The letters in the far right column of the appendix are used for reference in the subsequent discussion. Since the focus of the study was to establish feasibility for handling various features, the prototype system's user interface is not terribly sophisticated. It is text–based, interactive I/O. The user types input at the keyboard, and the system prints output to the screen. Most of the user input consists of attempted phrase translations, and command numbers chosen from menus along the way.

The example in the appendix begins (–A–) as the user invokes the system by typing "glt" (short for German Language Tutor). The system prints a standard header and greeting, and asks the user to login. A "?" at this point gives the user information about login names. The system supports two types of users. The ini-

tial example is of a "student" user. Later in the session, some "system administrator" user features are shown.

Once a student user has successfully logged in, the system requests his or her knowledge level, in the form of highest chapter completed in the text (–B–). The system is "tied" to a specific textbook, both as a means to define knowledge level, and to ensure that the system complements at least one curriculum. This would not be easy to change, but it did provide a consistent method on which to base the prototype. Internally, the knowledge level is marked in the knowledge base, limiting questions to sentence forms and vocabulary which should be familiar to a student at this level. Once the knowledge level is established, the system begins exercises.

The first exercise (–C–) shows a response which the system cannot intelligently diagnose. Under–the–hood, this means that the diagnosis tree has no pattern, even with wildcards, which will match the response. The system indicates that it cannot process the response, and gives the user the option of trying again, seeing what the system was expecting, quitting the current exercise, leaving the system altogether, or receiving a more detailed explanation of the choices. In the example, the user chooses to see what the system was expecting, and the system displays the expected response.

The next exercise (–D–) illustrates hints given by the system. Note that several subsequent hints are requested, and each is increasingly more specific. Under–the–hood, the first hint is from the pattern tree node which first matched the incorrect response. Each subsequent hint comes from following right–branches (the "match path") in the tree. Eventually, the system hits a leaf node in the diagnosis tree. This is reflected in the example by the system indicating that it cannot diagnose any further. At this point, the choice for additional hints is dropped from the menu. The user chooses "try again", and the system prints the original phrase for another attempt.

The next example exercise (–E–) was chosen to illustrate a response which takes a different form than the original phrase. The user attempts a literal translation, and is given the usual options for an incorrect response. The user then requests a hint, and is told that an article is not needed in this response. The user tries again, and the system acknowledges the correct response.

The next fragment of the example session (–F–) shows the summary which the system gives, as the student exits. Additional information is logged, attempting to characterize the nature of incorrect responses, etc., but these must be accessed by a system administrator.

At this point (–G–), the system is invoked again, with a system administrator logging in. As can be seen, the menu options available to the system administrator are different, and involve mainly manipulating the logged data. The first command chosen (–H–) reports on an individual diagnosis failure. An entry like this is made whenever the system encounters a leaf node when diagnosing a student response. There are two types of entries – one which is logged when a leaf node is hit trying to supply additional hints to the student, and one which is logged when a leaf node is hit trying to match the response originally. Both types of entries are logged so that a system administrator/designer can enhance the system to prevent similar diagnosis failures in the future (i.e. by adding appropriate branches to the diagnosis tree).

The next excerpt (–I–) shows the system administrator choosing to look at session statistics. This is essentially the same information which is given as a student user logs off. The system administrator can step through individual session statistics, or look at cumulative statistics. In the final excerpt of the example session, the system administrator requests cumulative session statistics (–J–), then logs off.

As can be seen by the various command menus in the log, not all features were shown in this example.

3. Internal Architecture Overview

The design and implementation of the system was undertaken using object–oriented methods, and the associated terminology is used here for consistency. For those unfamiliar with the methodologies used, the terminology will probably be at least intuitive. If needed, additional details can be found in [Nichols, 1991a], [Wirfs–Brock, 1990] and [Staffan, 1993].

3.1. High–Level Description

The fundamental object, from a processing standpoint, is the PhraseModel. The PhraseModel represents everything that the system knows about a specific form of phrase or sentence – what semantically correct vocabulary can be used, how to build a syntactically correct phrase from this vocabulary, and how to diagnose a student's translation attempt. The normal execution of the system is basically a sequence of exercises generated by choosing PhraseModels from the list which the system supports.

The fundamental object, from a functionality standpoint, is the TranslationTree. This is a binary tree of masking patterns, through which the student response falls as the system tries to diagnose it. An initial diagnosis is made at the first pattern which matches the response, and additional help (hints) can be supplied to the user by matching subsequent patterns. The TranslationTree and its nodes generate these patterns at run–time, so that they work with the vocabulary which was chosen for the specific exercise. The normal execution of a single exercise consists of the PhraseModel generating a phrase to be translated, and the TranslationTree processing student responses.

Another important object, as far as objectives of the system go, is the Log. This object gathers run–time data to be used in characterizing system usage, and improving its diagnostic capability.

The remainder of the objects in the system function primarily as support for those just mentioned. These are discussed further in the next section.

3.2. Detailed Description

A PhraseModel works for a particular form of phrase or sentence. This is represented by a PhraseDescriptor. The PhraseDescriptor contains the information necessary to create a syntactically correct phrase of the intended form. In order to create a phrase to be presented to the student for translation, the PhraseModel draws a semantically correct BasePhrase from a pool of choices. A BasePhrase is an indication of valid vocabulary selections for this form of phrase, with no syntactic information included. The PhraseDescriptor uses the BasePhrase, and builds the syntactically and semantically correct phrase which the PhraseModel presents to the user.

The TranslationTree works from the same BasePhrase, and builds the system's expected response. All diagnosis is done by matching the student response against patterns, and the expected response is essentially the first pattern in the binary tree – a root pattern with no wildcards. If the student response matches the expected response, the current exercise is complete, the system logs some data and statistics, and another PhraseModel is chosen.

If the student response does not match the expected response, then it is fed to the first PatternNode in the binary tree. The pattern node builds a pattern, which consists of the expected response with some strategically placed wildcard(s). This pattern is matched against the student response, and if it fails, the student response is fed down the left branch of the tree until a match is found. Each generated pattern is different, and/or increasingly more generic, in an effort to isolate the problem area.

Once the student response matches a pattern, the system reports that the response was not the expected one, and asks the student how to proceed. If the student wishes to try again, the system resets to the root of the TranslationTree, and waits for another attempt. If the student asks for a hint, the one which is stored at the current PatternNode will be printed. If the student asks for additional hints, the system will feed the student's response down the right branch of the tree, from the current node. Subsequent patterns will try to further isolate or qualify the problem, and hints at those levels will be increasingly more specific. Once the tree–parsing hits a leaf node, though, the system will tell the user that it can diagnose no further.

A Dictionary object supports the system, by returning the derived forms of words based on specified syntactic information. The PhraseModel and TranslationTree use this as they build the presentation and expected response phrases.

A Statistics object maintains run–time counts, such as how many exercises have been attempted during the current session. This information is made available to the user periodically, as well as logged for future inspection.

The following section is a system walk–through to illustrate how these objects work with an example.

3.3. Walk–through to Illustrate Internal Function

This walk–through is included as a way to show how the different objects interact, and how the various dynamic objects are created and used. It also shows how the pattern–tree diagnosis concept works.

3.3.1. Static Data

Before actually beginning the processing, it is important to see what static objects are in place. This is the system's knowledge base. A BaseWord is a primitive data element in the system. It is not actually a word, but rather a key into the dictionary indicating a specific word, but not the word's form. For example the BaseWord for "man" would also be the base word for "men". As will be seen later, a WordDescriptor contains in-

formation about what form of a word is required, and a BaseWord and WordDescriptor are all that is needed to obtain a Word from the dictionary. BaseWords are combined to form BasePhrases. BasePhrases are grouped into BasePhraseLists, which in turn are grouped into BasePhrasePools. The need for two levels of grouping will be illustrated below. Each PhraseModel has its own BasePhrasePool to draw from. To begin the example, here are two BasePhraseLists, each containing two BasePhrases:

BPL–1: "the", "man", "to be", "good"
"the", "dog", "to be", "hungry"

BPL–2: "a", "boy", "to be", "energetic"
"a", "bird", "to be", "noisy"

As mentioned earlier, a PhraseModel has a PhraseDescriptor which specifies the form of phrase or sentence represented by this model. This contains syntactic information. The PhraseDescriptor also contains information on the order in which the phrase must be built. This is so that words which must agree with each other (in gender, number, etc.) will. Here is an example PhraseDescriptor, for BPL–1 and BPL–2:

PD–1: "article, gender–dynamic", "singular noun",
"present tense verb", "adjective"
Fill–in order: 2, 1, 3, 4

The actual elements of the PhraseDescriptor are WordDescriptors. The WordDescriptors contain feature (or attribute) information describing the required form of the word. For example the noun specified above is singular, as opposed to plural. A special characteristic of WordDescriptors is that features may be marked as "dynamic", as is the gender of the article in the above example. This supports the run–time building of phrases where words must agree with each other, and will be shown later.

The following should illustrate the difference between BasePhrasePools and BasePhraseLists. A BasePhrase and a PhraseDescriptor together specify a syntactically and semantically correct phrase. The following example BasePhrasePool is composed of both of the example BasePhraseLists.

BPP–1: BPL–1
BPL–2

A PhraseModel using the above example PhraseDescriptor, PD–1, and the example BasePhrasePool, BPP–1, would be able to generate the following valid phrases:

The man is good.
The dog is hungry.
A boy is energetic.
A bird is noisy.

BPL–1 and BPL–2 are separate entities, though, because they will not always be valid for the same PhraseModel/PhraseDescriptors. As an example of this, suppose the fixed_phrase_descriptor for a different PhraseModel is:

PD–2: "article, gender–dynamic", "plural noun",
"present tense verb", "adjective"
Fill–in order: 2, 1, 3, 4

A valid BasePhrasePool is:

BPP–2: BPL–1

which would generate the valid phrases:

The men are good.
The dogs are hungry.

BPP–2 cannot contain BPL–2, however, because the generated phrases would not be valid:

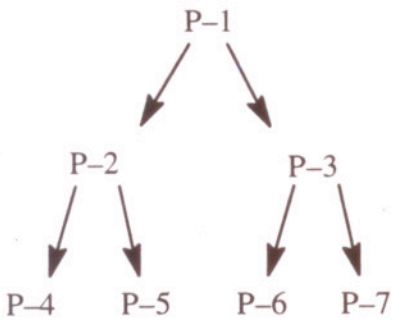
A boys are energetic.
A birds are noisy.

BasePhraseLists, then, exist so that subsets of pools can be shared between PhraseModels.

The first example PhraseModel, so far, consists of BPP–1 and PD–1. The next component of a PhraseModel is its TranslationTree. A TranslationTree object is at the root of the binary pattern–matching tree. Like the PhraseModel, it also contains a PhraseDescriptor. This describes the expected response. In many cases, this will be the same PhraseDescriptor that the PhraseModel uses. In those cases were it is not the same, the TranslationTree will also contain a BasePhraseTranslator object which describes how to translate the original BasePhrase to the new language. This handles cases where the translated phrase does not take on the same form as the original (e.g. greater/fewer words or different word order). The example here does not require this, so no additional PhraseDescriptor is included.

The final static components of the PhraseModel and TranslationTree are the individual PatternNodes. The tree itself is pre–determined, but there are dynamic actions which occur during an exercise, and these will be illustrated below. In order to have an example to work from, consider the following tree (without yet specifying the details of each pattern):

Figure 1. Example Pattern Tree



That completes the description of what the system knows before starting an exercise. The next section examines the processing which would occur using this example data.

3.3.2. Run–time Processing

The first step of an exercise is choosing a BasePhrase to work with. It is intended that this be a random selection from the BasePhrasePool (BPP–1, in this case). For this example, say that "the" "man" "to be" "good" was chosen. The PhraseDescriptor (PD–1) is used to generate the phrase to be presented to the student. A special object, the DynamicWordDescriptor is used along the way to help resolve dynamic information in the WordDescriptors. As the phrase is constructed, feature information is set in the Dynamic WordDescriptor. Using the chosen BasePhrase, and following the fill–in order specified by the PhraseDescriptor, the second word of the presentation phrase would be obtained from the dictionary (for this example we're assuming a German to English translation by the student, so the German forms of the word are filled in):

presentation phrase = ____ Mann ____ ____

In addition to setting this word, its feature information is set in the DynamicWordDescriptor (specifically, the fact that the gender of this word is masculine). As each subsequent WordDescriptor is used, it is first checked to resolve any dynamic feature information against the DynamicWordDescriptor. The next WordDescriptor would then be resolved to "masculine article", and would be obtained from the dictionary:

presentation phrase = der Mann ____ ____

These steps are repeated until the entire presentation phrase has been constructed. It is then presented to the user:

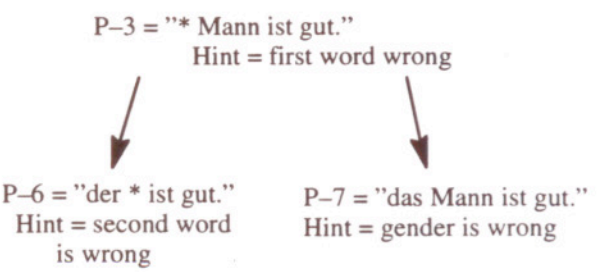
Translate:
der Mann ist gut
to English:

At this point, the TranslationTree takes over, and waits for the student's response. It builds the Phrase that it expects (in this case "the man is good") and compares the two. If the response is what the system expected, the system would inform the user, and this exercise would be over. For sake of example, though (and because gender is a convenient error to illustrate), consider the following exercise, with the languages reversed:

Translate:
the man is good
to German:

In this case, the correct response would be "der Mann ist gut". To illustrate error diagnosis, assume that the response was "das Mann ist gut" (gender of the article is incorrect).

Because the match with the expected response fails, the TranslationTree object feeds the actual response into the pattern tree. The following is a simplified example of how a fragment of a pattern tree might look. Every PatternNode in the tree is similar in structure. Each contains the rules necessary for building the pattern needed at that level, and a hint as to the nature of the discrepancy at that level. The patterns are dynamically built, and the ones shown below indicate how they might look for this example vocabulary. (In the notation here, '*' represents the single–word wildcard.)



The student response would fall through the tree until it first matched a pattern. In this case, that would be P–3. At this point the system will ask the student what he or she wishes to do. As mentioned earlier, if the student chooses to try again, or to receive the correct answer, the system will leave the diagnosis tree. If the student requests a hint, though, the system will print the hint associated with the current PatternNode. In this case, it would indicate to the user that it perceives a

problem with the first word in the response. If the user asks for another hint, the system will attempt a match down the right branch of the tree. In this case, a match is found with the pattern P–7, and the next hint is printed, this time telling the user that it was the gender of the first word that caused the problem. This can continue as long as the system has a right branch to follow. Once it hits the leaf, though, it will tell the user that no further diagnosis is possible. The left branch, P–6 would only have been taken if the response had not matched at P–3 originally. Internally, the left branch is considered the "no–match" path (taken while looking for the first match) and the right branch is considered the "match" path (taken to supply hints, as long as it continues to match the user's response).

That concludes a single exercise processing walk–through. Diagnosis will continue until the user answers correctly or quits the current exercise, then a new exercise will begin.

4. Design and Implementation

4.1. Introduction

A detailed examination of the design of this system is beyond the scope of this paper. However, a brief description with a few examples is included, to give the reader a sense of the process used. There are various object–oriented analysis and design techniques in use today, many of which borrow elements from one another. This effort relied heavily on two methodologies – Job–Oriented Object Analysis (JOOA) [Nichols,1991a] and Class–Responsibility–Collaboration (CRC) Design [Wirfs–Brock,1990].

In general, object–oriented approaches tend to be more iterative, as opposed to using the more rigid phases and gates of other methodologies. The lines between analysis, design and implementation tend to blur, as earlier decisions are re–visited to address modularity, reusability, and responsibility issues uncovered in later phases.

In addition to the traditional development phases, the system itself was approached as having two distinct design components. The traditional "system design" defined the internals of the system – the mechanics of the algorithms, data structures, etc., and the "knowledge base design" supplied the actual data which drives the system. The system design alone specified an empty shell. The knowledge base design filled in the data. For this type of data–driven processing, the data design is critical. Diagnosis of student responses is done by using trees of patterns to isolate and identify errors. These trees are part of the knowledge base data, and must be carefully chosen to correctly identify errors.

4.2. System Design

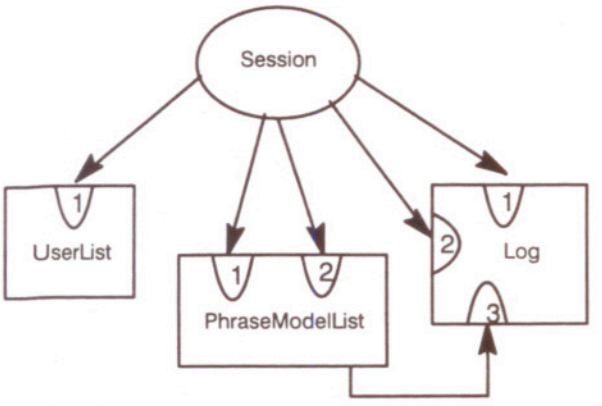
Using the JOOA method mentioned above, two workplaces were identified for this system. These can be thought of as users, or as differing outside views of the system. They are listed below with their basic responsibilities.

Student	– ID User – Quiz User
System Administrator	– ID User – View and Manipulate Logs

The responsibilities became the initial "agents" of the system. The JOOA concept of agent can be thought of as a broad conceptual object. Most will break down into distinct classes/objects as a design firms up.

It turns out that this system is not very complex from this external perspective. The agents identified above eventually gave up their functions to various objects, leaving only one agent – the system–level "session". This was actually a convenient outcome, since this single agent could then become the main() function of the C++ implementation. The following picture (figure 2) illustrates how this agent relates to the major objects in the system. In this type of diagram, ovals represent agents, and rectangles represent objects. The numbered interaction points represent "contracts" between agents and/or objects. A one–line description of the functionality covered by the contract is included.

Figure 2. System–level Diagram



UserList

1. read and validate user

PhraseModelList

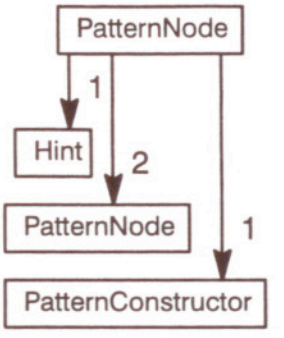
1. set current knowledge level
2. do exercise

Log

1. examine data
2. manipulate data
3. add data

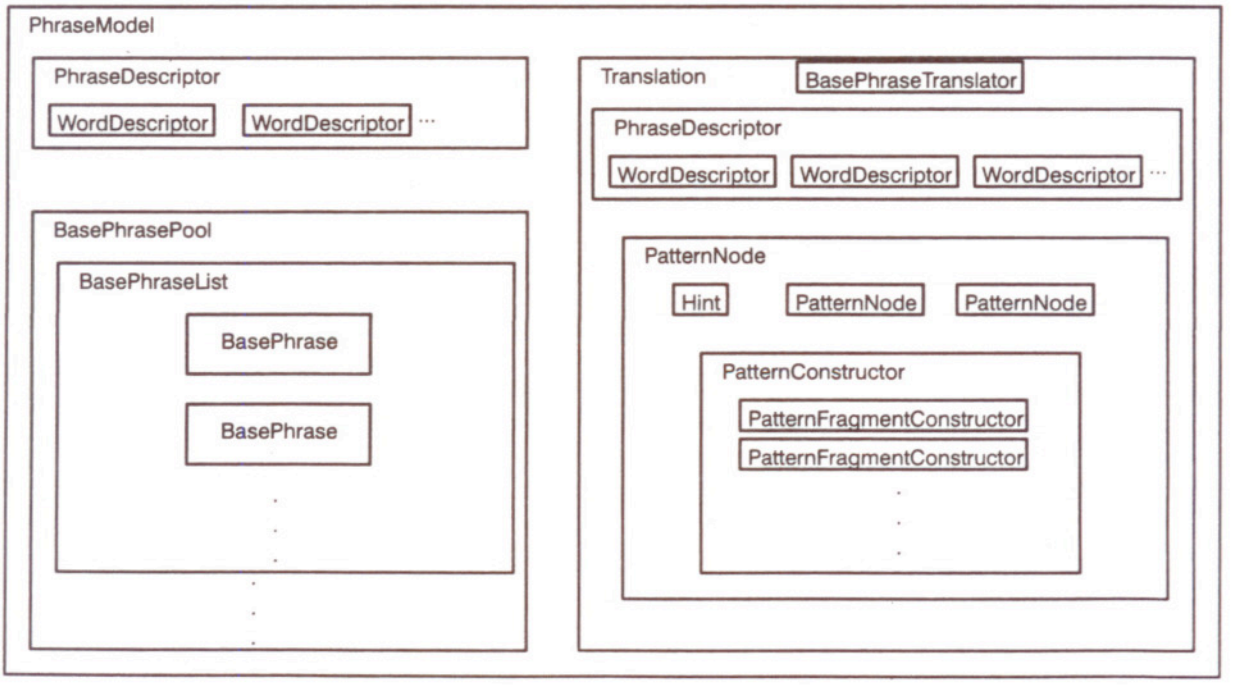
Each individual object, when identified, was documented in a "composition graph", which showed any objects which it in turn contained. The PatternNode graph below is an example (figure 3). The numbers on the arrows indicate how many of each object are contained.

Figure 3. Composition Graph



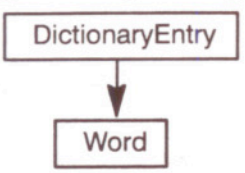
It was also useful, for some of the more complex objects, to create a composite illustration to show how a large group of objects was interrelated. An example of one of these, the PhraseModel object, is shown in figure 4 (note that the PhraseModel object does contain PatternNode objects).

Figure 4. Composite Object Illustration



Another design/documentation tool used was the begetter graph. These illustrate objects that can create other objects at run–time. The DictionaryEntry example below (figure 5), stems from the fact that a dictionary request can produce a word. A similar graph, with a slash through the arrow, is used to illustrate objects which destroy other objects at runtime. (These are generally only used for exception cases. If no destruction graph is shown, it is assumed that an object is destroyed by the object which created it.)

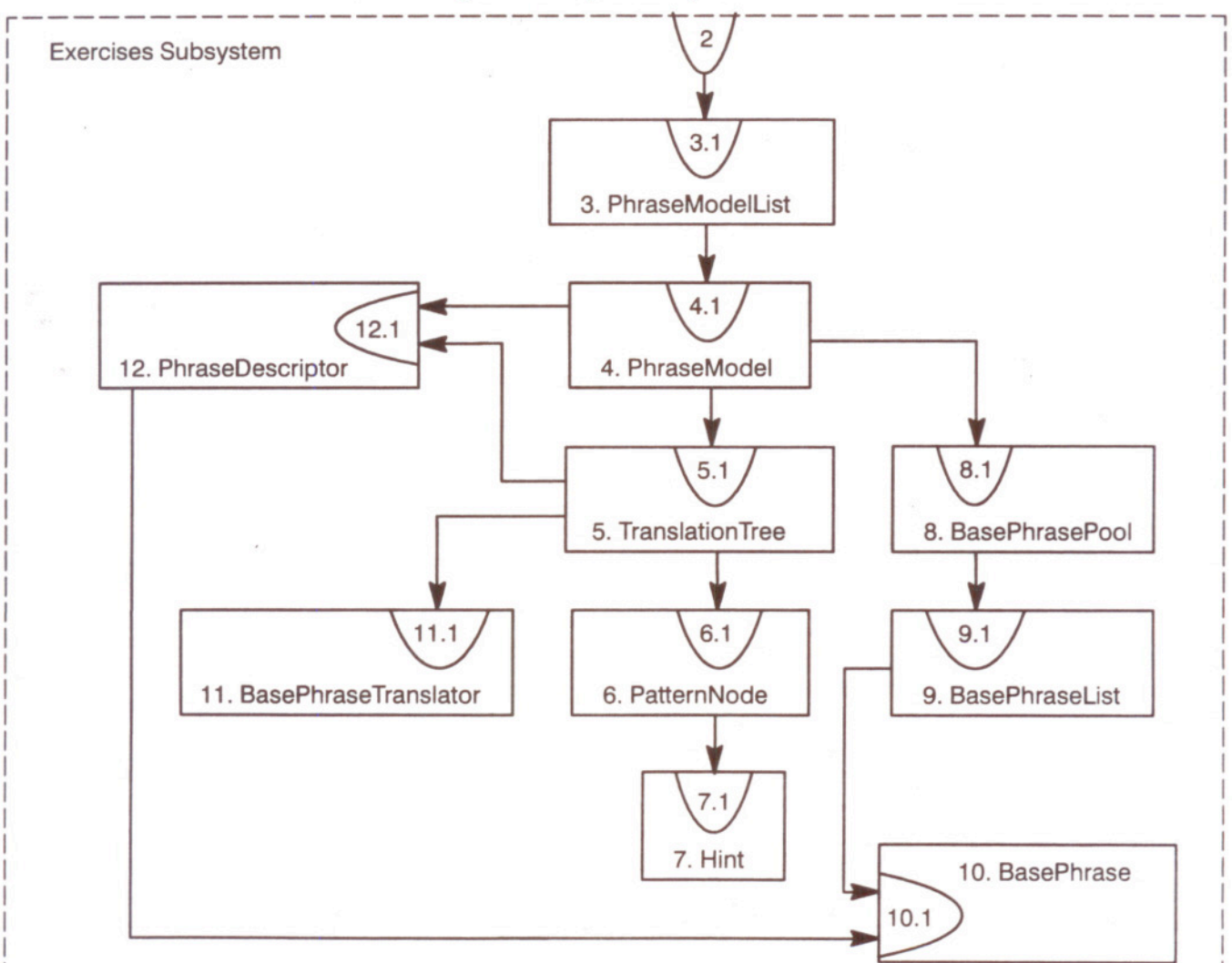
Figure 5. Begetter Graph



Besides the various graphs, each object is documented textually, with responsibilities, attributes and actions listed.

As the bulk of the objects were identified, the relationships between the objects were documented. Some of the objects were grouped into subsystems (logical groupings of related objects). Three subsystems were defined for this system – the dictionary, the exercise handler, and the data log. Subsystems can be represented in collaboration graphs, using the notation shown earlier in the paper for the session agent. An illustration of the exercise handling subsystem is shown in figure 6 below. The numbered interface points on each object represent the contracts which each object supports for the operation of the subsystem. These are described following the example.

Figure 6. Subsystem Graph



The numbered interfaces for the exercise handling subsystem are described below (they don't begin with "1" because this example was pulled out of the larger scope of the system):

3. PhraseModelList
    3.1 Set active knowledge level, do exercise.
4. PhraseModel
    4.1 Set active knowledge level, do exercise, reset depletions
5. TranslationTree
    5.1 Check student response.
6. PatternNode
    6.1 Process response.
7. Hint
    7.1 Print.
8. BasePhrasePool
    8.1 Set active knowledge level, get BasePhrase, reset selections.
9. BasePhraseList
    9.1 Set active knowledge level, get BasePhrase, reset selections.
10. BasePhrase
    10.1 Get index.
11. BasePhraseTranslator
    11.1 Translate.
12. PhraseDescriptor
    12.1 Build Phrase.

4.3. Knowledge Base Design

It was a design goal for this system to support the gathering of data which would allow its diagnosis ability to be improved. Since this is the best way to find out what type of responses will be encountered, it is not unlikely that, over time, the internal knowledge base reflect more of this feedback than of the original design. This is not unreasonable, or unexpected, but it is still necessary to put in place an initial knowledge base from which to work. This involved identifying several things: the phrase/sentence models to be supported, the vocabulary to be supported for different levels, the types of errors which the system would be able to diagnose for a given model, and the processing details to support the different types of errors.

As stated earlier, this incarnation of the system is highly tied to a specific textbook. It was also decided that "chapter completed" would be a reasonable definition of knowledge level, at least for the purpose of implementing the prototype system. The book chosen was German Made Simple [Jackson, 1985]. The selection of the phrase models implemented resulted from identifying the various models introduced in early chapters of the text, and narrowing it down to a few which illustrated important features of the system. The random selection of models and vocabulary is illustrated merely by having enough different choices built into the initial knowledge base.

The selection of the phrase models implemented resulted from identifying the various models introduced in early chapters of the text, and narrowing it down to a few which illustrated important features of the system. The random selection of models and vocabulary is illustrated merely by having enough different choices built in to the initial knowledge base. Other features are described in the sections below.

Vocabulary selection was fairly straightforward, once phrase models were selected. Each chapter introduces a finite set of vocabulary. Valid vocabulary choices for the different phrase models were compiled from each vocabulary set. The knowledge level associated with an exercise becomes the higher of the phrase model (chapter in which the sentence form was introduced) and the highest level of any individual word (chapter in which that word was introduced). This allows older (lower level) vocabulary to be used in more advanced phrase models, and more advanced vocabulary to be used in the earlier phrase models.

The diagnosis tree design was done on a per–phrase model basis. The types of errors to be diagnosed for the given model were identified first. These come from the topics covered in the text where this phrase form was introduced, from previous concepts introduced, from predicting potential incorrect responses, and can be verified and enhanced by run–time information gathering. An example portion of a diagnosis tree is described below. The hint text which the system will display is nothing more than a description of the type of error which the system catches.

As notational shorthand, several pieces of information are represented in the tree. The emboldened lines indicate the specific concepts which are tested by the given model. The indentation represents the sequential nature of the diagnosis, using the binary tree. The binary tree can be visualized from the textual layout (figure 7).

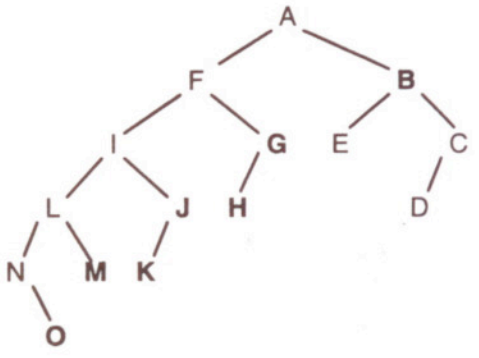
and is illustrated immediately following the text (figure 8). Starting with any line in the text, a left branch (the no–match path) can be followed by falling down to the next line which is at the same indentation level. A right branch (the match path) can be followed by going to the next contiguous line of increasing indentation. If there is no line with the same indentation level to fall to, the current line represents a left leaf node, and if there is no contiguous line of greater indentation, the current line represents a right leaf node. The path followed on the way to an emboldened line indicates the diagnosis of a particular concept. The path which can be followed from an emboldened line, if any, indicates further diagnosis which the system is capable of (typically for giving the user hints).

The phrase represented by this model is a complete sentence taking the form "direct article, singular noun, present tense verb, adjective". The knowledge level of this particular model is 2. An example presentation phrase is "the man is good", and an example expected response is "der Mann ist gut".

Figure 7. Diagnosis Tree Excerpt

- A) First word is different from expected
  - **B) Gender of article does not agree with noun**
    - C) Gender variation 1 (see below)
    - D) Gender variation 2
  - E) Expecting a direct article
- F) Second word is different from expected
  - **G) Noun is not capitalized**
  - **H) Synonym of noun used**
- I) Third word is different from expected
  - **J) Tense of verb is incorrect**
  - **K) Synonym of verb used**
- L) Fourth word is different from expected
  - **M) Synonym of adjective used**
- N) First and second words different from expected
  - **O) Synonym of noun used**

Figure 8. Same Excerpt in Tree Form



Note that use of a synonym for the noun appears twice in the diagnosis hierarchy. This is because an incorrect (or synonymous) noun may be of a different gender, which might also cause the article to be different from that expected. As mentioned earlier, the types of errors initially diagnosed are taken from the text. At run–time, data is stored to indicate how parsing of diagnosis trees terminates (e.g. at a leaf node, trying to get help, a correct answer after a hint, etc.) This data can be analyzed so that the diagnosis trees can be enhanced. For example, if a certain exercise causes a large number of students to give up, even after viewing all of the available hints, then perhaps the tree needs to be made deeper, so that more detailed hints can be given.

4.4. Implementation

The system was implemented in C++. Prolog and C were used for earlier prototypes, and a variety of logic, pattern matching and object–oriented languages were considered for the system as described here. A language which supported object–oriented constructs was selected in order to ease the implementation of the object–oriented design. C++ was chosen for its availability across platforms, and its popularity in current commercial development.

Because the pattern matching is such a core element of the system, two brief excerpts of code are included here to illustrate how runtime pattern generation occurs. Various static pattern fragments or regular expressions are pre–defined to be used by phrase constructors, as needed. This first example pattern constructor

has a need for a match–a–single–word wildcard, so it uses the following:

```
typedef char * pattern_fragment;

pattern_fragment any_word = "[a–zA–Z][a–zA–Z]*";
```

The following is the complete constructor function which generates the pattern to determine if the second word of a phrase is different from what was expected.

```
Pattern PC_unexpected_second_word(Phrase &
                                  in_phrase)
{
    Pattern       new_pattern = in_phrase;
    Word          new_word;

    new_word.set_string(any_word);

        // The first argument in the set_word member
        // function call is the word position:
    new_pattern.set_word(2, new_word);

    return(new_pattern);
}
```

The creation of new_pattern works, because the Pattern object has a constructor which tells it how to make a Pattern from a Phrase. The pattern created is a regular expression which can be matched against the student response.

The second example, below, does not use any pre–defined pattern fragments. It illustrates a pattern constructor which uses the dictionary to create a variation of the expected phrase. This type of pattern is typically used in the tree when it is trying to isolate or diagnosis a potential anticipated error.

```
Pattern PC_first_gender_var_first_word(Phrase &
                                       in_phrase)
{
    Pattern       new_pattern = in_phrase;

        // Argument to get_word member function
        // is word position in phrase. Extract the
        // word for which a variation is required.
    Word          old_word = in_phrase.get_word(1);

        // Create a new WordDescriptor, which
        // initially describes the old word.
    WordDescriptor      new_word_WD(old_word);

        // Get the word ID and the current gender.
    int   word_identifier = old_word.get_index();
    Gender new_gender =
                    new_word_WD.get_gender();

        // Calculate the first gender variation (see
        // explanation, below).
    new_gender = (Gender) ((((int) new_gender) %
                            (G_none–1)) + 1);

        // Change the gender in the WordDescriptor,
        // obtain this variation of the word from the
        // dictionary, and plug it into the Pattern in
        // the same position as the old word.
    new_word_WD.set_gender(new_gender);
    Word  new_word = dictionary.get_word(
                    word_index, new_word_WD );

    new_pattern.set_word(1,new_word);

    return(new_pattern);
}
```

This function could no doubt use a bit more explanation than the previous, but really it is not doing anything too difficult (and this is one of the most complex pattern constructors in the system currently).

This function is more complicated than the previous example, but it is really not doing anything too difficult (and this is one of the most complex pattern constructors currently in the system). As in the previous example, the initial pattern is merely the input Phrase. The function then gets the current first word from the input Phrase. A WordDescriptor is created from this word (the WordDescriptor containing the feature attributes of the word), and the integer word identifier is obtained. (A word identifier and a WordDescriptor are the two things necessary for getting words from the dictionary. The final piece of set–up involves getting the current Gender setting from the WordDescriptor. The Gender, and most attributes, are simply enumerated constants which represent the various values that the attribute can hold. The Gender definition looks like:

```
enum Gender
{
        G_DYNAMIC,
        G_masculine,
        G_feminine,
        G_neuter,
        G_none
};
```

The next cryptic–looking line in the example function just calculates what is internally known as the first gender variation. It amounts to the next gender setting modulo the valid value. If the input gender had been masculine, the new gender would be feminine. If the input gender had been neuter, the new gender would be masculine. This new gender attribute is then set in the WordDescriptor, and the dictionary is called on to supply the new derived form of the indicated word. This new word is set in the pattern, and the pattern is complete. If the input phrase is "Der Mann..." the output pattern would be "Die Mann...".

5. Conclusions

This project is described in greater detail in the thesis report, [Staffan, 1993]. This includes additional discussion of problems encountered and limitations of the system, as well as areas for future enhancement (though this prototype was not designed specifically to support such enhancements). Also included are User's and Administrator's Guides for the system, and a brief discussion and list of various commercial packages which fall in the same domain.

One limitation worth mentioning here is the fact that significant vocabulary and logic associated with different types of words and phrases would need to be added in order to make this system practical as a real–life tutoring tool. Several examples have been implemented to illustrate the concepts, objects, and functionality, but the prototype is a long way from being a robust system. Besides that requirement, other areas of potential improvement include teacher configurability, lesson/concept material to be presented as help for the student, adaptive behavior, and a better method/notation for designing and implementing extensions to the knowledge base. The current user interface is primitive, and could be replaced by a module which presents a friendlier front–end.

Quite a bit was learned from the prototype system. While a re–design would probably lead to a much cleaner implementation (now that the ramification of object–oriented design decisions have been experienced in implementing, debugging, and enhancing the final system), the basic premise of the system, its underlying architecture, and the object–oriented approach appear to have been generally sound.

6. Acknowledgements

The authors would like to acknowledge the assistance of Kevin Donaghy, PhD., who lent guidance to the prototype project from its inception.

7. References

Boyd, Gary, Arnold Keller and Roger Kenner (1982), "Remedial and Second Language English Teaching Using Computer–Assisted Learning", Computers and Education, Vol 6, pp 105–112.

Burton, R. R. (1981a), "An Investigation of Computer Coaching for Informal Learning Activities", in Intelligent Tutoring Systems, Academic Press, pp. 79–97.

Burton, R. R. (1981b), "Diagnosing Bugs in Simple Procedural Skills", in Intelligent Tutoring Systems, Academic Press, pp. 157–182.

Byrd, Roy J. (1983), "Word formation in natural language processing systems", Proceedings of IJCAI–VIII, pp. 704–706.

Byrd, R. J. and N. Calzolari, M. S. Chodorow, J.L. Klavans, M. S. Neff, and O. A. Rizk (1987), "Tools and Methods for Computational Lexicology," *Computational Linguistics*, vol. 13, numbers 3–4, July–December, pp. 219–240.

Byrd, Roy J. and Tzoukermann, Evelyne (1988), "Adapting an English Morphological Analyzer for French", Proceedings of the 26th Annual Meeting of the Association for Computational Linguistics.

Dhaif, H. A. (1990), "Computer–Assisted Language Learning: A Client's View", CALICO Journal, Vol 7(4), pp. 67–81.

Farrington, Brian (1982), "Computer–Based Exercises for Language Learning at University Level", Computers & Education, Vol. 6, pp. 113–116.

Farrington, Brian (1986), "Computer–Assisted Learning or Computer–Inhibited Acquisition", in Computers and Modern Language Studies, Cameron, K. C. and W. S. Dodd and S. P. Q. Rahtz, eds., Ellis Horwood Limited.

Ferney, Derrik (1989), "Small Programs That "Know" What They Teach", in Computers and Modern

Language Studies, Cameron, K. C. and W. S. Dodd and S. P. Q. Rahtz, eds., Ellis Horwood Limited.

Goldstein, I. P. (1981), "The Genetic Graph: A Representation for the Evolution of Procedural Knowledge", in Intelligent Tutoring Systems, Academic Press, pp. 51–75.

Hartley, J. R. and D. H. Sleeman (1973), "Towards Intelligent Teaching Systems", International Journal of Man–Machine Studies, Vol. 5, pp. 215–236.

Heidorn, G. E. and K. Jensen, L. A. Miller, R.J. Byrd, and M. S. Chodorow (1982), "The EPISTLE Text–Critiquing System," IBM Systems Journal, vol. 21, pp. 305–326.

Inwood, Clifford (1992), "Analysis Versus Design – Is There a Difference", *The C++ Journal*, Vol. 2, No. 1.

Jackson, Eugene and Adolph Geiger (1985), *German Made Simple*, Doubleday, textbook for St. John Fisher College's Beginning German I and II courses.

Kline, Paul J. and Steven B. Dolins (1985), "Choosing Architectures for Expert Systems", Rome Air Development Center.

Koenig, Andrew (1992), "Checklist for Class Authors", *The C++ Journal*, Vol. 2, No. 1, pp. 42–46.

Last, R. W. (1989), Artificial Intelligence Techniques in Language Learning, Ellis Horwood, Chichester.

Nichols, Tim (1991a), "Job Oriented Object Analysis – A Method of Finding and Validating Objects", internal report, Eastman Kodak Company.

Nichols, Tim (1991b), course notes: "Object Oriented Concepts", "Object Oriented Analysis", and "Object Oriented Design", Eastman Kodak Company.

Phillips, Robert (1983), "Masking Techniques to Identify and Diagnose Errors in Foreign Language CAI", proceedings of the Sixth International Conference on Computers and the Humanities, Burton, S. K. and D. D. Short, eds., Computer Science Press.

Sleeman, D. H. (1981a), "Assessing Aspects of Competence in Basic Algebra", in Intelligent Tutoring Systems, Academic Press, pp. 185–197.

Sleeman, D. H. and R. J. Hendley (1981b), "ACE: A System Which Analyzes Complex Explanations", in Intelligent Tutoring Systems, Academic Press, pp. 99–116.

Staffan, Kenneth E. (1993), "An Intelligent Tutoring System for the German Language", Master's thesis, Rochester Institute of Technology.

Stroustrup, Bjarne (1987), *The C++ Programming Language*, Addison–Wesley.

Uhr, L. (1969), "Teaching Machine Programs that Generate Problems as a Functionof Interaction with Students", Proceedings of the 24th National Conference, pp. 125–134.

Wirfs–Brock, Rebecca, Brian Wilkerson, and Lauren Wiener (1990), *Designing Object–Oriented Software*, Prentice Hall.

Woods, P. and J. R. Hartley (1971), "Some Learning Models for Arithmetic Tasks and their Use in Computer–Based Learning", British Journal of Educational Psychology, Vol. 41, No. 1, pp. 35–48.

Wybolt, Nicholas (1990), "Experiences with Object–Oriented Software Development", proceedings of the USENIX C++ Conference, April 9–11.

Appendix

```
Sys27> glt                                                          -A-
                        German Language Tutor, version 1.0

If you are not sure how to begin, type "?".

Please login: ?

        In order the use this system, you must have an authorized
        username.  If you have not received a username yet, see
        your system administrator.  If you do have a valid username,
        type it in now to gain access to the tutoring system.

Please login: ken

Enter highest chapter completed (currently 5)? 5                    -B-

        Exercises will be limited to chapter 5 and earlier.

        You will now be asked to translate various phrases and
        sentences.  You may stop at any time - the important command
        to remember is "?".  Type a question mark at anytime to
        be told your options and be given the opportunity to leave.

Translate:
                How old is the child?                               -C-
to German:
                this attempt resembles the original in no way

        That response is incorrect, but the system is unable
        to isolate the specific problem.

        1 - Try again.
        3 - Have the system give you its answer.
        4 - Quit this exercise and go on to next.
        5 - Exit the tutoring system.
        6 - Help.

Choice? 3

        The system was expecting:

                Wie alt ist das Kind?


Translate:
                the table is round                                  -D-
to German:
                die Tisch ist rund

        That is not the expected translation.  Enter the number
        indicating the action you would like to take:

        1 - Try again.
        2 - Have the system give you a hint.
        3 - Have the system give you its answer.
        4 - Quit this exercise and go on to the next.
        5 - Exit the tutoring system.
        6 - Help.

Choice? 2
```

```
        There is something wrong with the first word in your response.

        1 - Try again.
        2 - Have the system give you a hint.
        3 - Have the system give you its answer.
        4 - Quit this exercise and go on to the next.
        5 - Exit the tutoring system.
        6 - Help.

Choice? 2

        The gender of the first word is wrong.

        1 - Try again.
        2 - Have the system give you a hint.
        3 - Have the system give you its answer.
        4 - Quit this exercise and go on to the next.
        5 - Exit the tutoring system.
        6 - Help.

Choice? 2

        The gender of the article must agree with the noun.  The noun is
        masculine, but the gender of the article you used is feminine.

        1 - Try again.
        2 - Have the system give you a hint.
        3 - Have the system give you its answer.
        4 - Quit this exercise and go on to the next.
        5 - Exit the tutoring system.
        6 - Help.

Choice? 2

        The system cannot diagnose this any further.

        1 - Try again.
        3 - Have the system give you its answer.
        4 - Quit this exercise and go on to the next.
        5 - Exit the tutoring system.
        6 - Help.

Choice? 1

Translate:
                the table is round                                        -E-
to German:
```

[Portion of log deleted...]

```
Translate:
                the man is a merchant
to German:
                der Mann ist ein Kaufmann

        That is not the expected translation.  Enter the number
        indicating the action you would like to take:

        1 - Try again.
        2 - Have the system give you a hint.
        3 - Have the system give you its answer.
        4 - Quit this exercise and go on to next.
        5 - Exit the tutoring system.
```

```
        6 - Help.

Choice? 2

        An article should not be used in this case.

        1 - Try again.
        2 - Have the system give you another hint.
        3 - Have the system give you its answer.
        4 - Quit this exercise and go on to next.
        5 - Exit the tutoring system.
        6 - Help.

Choice? 1

Translate:
                the man is a merchant
to German:
                der Mann ist Kaufmann

        Yes! That is the expected response.
```

[Portion of log deleted...]

```
        1 - Try again.
        2 - Have the system give you a hint.
        3 - Have the system give you its answer.
        4 - Quit this exercise and go on to next.
        5 - Exit the tutoring system.
        6 - Help.

Choice? 5                                                    -F-

German Language Tutor exiting...

Duration of session - 20 minutes
Number of different phrases asked to translate - 50
Number of correct responses - 44
Number of total responses - 57
```

[Portion of log deleted...]

```
Sys27> glt                                                   -G-

If you are not sure how to begin, type "?".

Please login: sys_user

        Select desired function from menu below.

        5  - Exit the tutoring system.
        6  - Help.
        7  - Write current log entries to file.
        8  - Delete log file.
        9  - Save log file to new name.
        10 - View system diagnosis failures.
        11 - Generate report, system diagnosis failures.
        12 - View individual past session statistics.
        13 - Generate report, individual past session statistics.
        14 - View cumulative past session statistics.
        15 - Generate report, cumulative past session statistics.
        16 - Generate report, all information available.
```

```
Choice? 10                                                          –H–

Leaf node encountered while looking for additional hints.
        Date - Sat Apr 03 1993     Username - stu_user
        Index of model chosen - 1
        Phrase which was to be translated - the house is old
        Expected response - das Haus ist alt
        Actual response - der Haus ist alt

        5  - Exit the tutoring system.
        6  - Help.
        4  - Return to main system administrator menu.
        17 - View next entry.
```

[Portion of log deleted...]

```
Choice? 12                                                          –I–

Statistics after a student session.
        Date - Sat Apr 03 1993     Username - stu_user
        This session lasted 2 minutes, 12 seconds.
        Number of different phrases asked to translate - 2
        Number of correct translations - 1
        Number of total responses - 3

        5  - Exit the tutoring system.
        6  - Help.
        4  - Return to main system administrator menu.
        17 - View next entry.
```

[Portion of log deleted...]

```
Choice? 14                                                          –J–

Cumulative statistics from log file.
        Number of individual sessions counted - 2
        Total duration of sessions - 10 minutes, 26 seconds.
        Average duration of sessions - 5 minutes, 13 seconds.
        Total number of phrases presented by system - 5
        Total number of student responses - 6
        Total number of correct translations - 1

        5  - Exit the tutoring system.
        6  - Help.
        4  - Return to main system administrator menu.

Choice? 5


        Your session with the German Language Tutor is complete.
```