Fundamental:
The work of George Boole (“The Laws of Thought”) and Noam Chomsky (“Universal Grammar Theory”) prove the existence of Natural Laws of Intelligence in nature, and that they are hard-wired in our brain. I found a way to utilize those Natural Laws of Intelligence in grammar to create intelligence / meaning in software.

In a nutshell:
The worldwide unique steps of my approach:

- Determining word types without words list;
- Using these word types to find a path in the grammar rules;
- Switching language if no path was found;
- Deriving exhaustive information from the found grammar path;
- After the entered sentence is stored in the knowledge structure, the sentence is retrieved again and written as a readable sentence. If one or more words are missing or has changed, an integrity error shown to the user that the system has encountered a short-coming. In this way, Thinknowlogy is guaranteed to preserve the meaning;
- The information of the entered sentence is fed to the integrated reasoner, which includes information on the use of definite article “the”, conjunction “or”, possessive verb “has/have” and past tense verbs “was/were” and “had”;
- And the output of the reasoner is fed to a reversed parser, using the grammar rules to generate a readable sentence from the derived knowledge.

Reading (see classes containing the word “Read”):

- When a sentence is entered, function *createReadWords* in class *AdminReadCreateWords* creates suitable **word types** for each word of the entered sentence. Each word can have multiple word types, because at this stage it isn't clear yet what the correct word type will be. (The unused word types will be cleaned up later, after the sentence is parsed);
- The **rules** in the grammar configuration file (/data/grammar/{language name}.txt) are designed to perform the first step in parsing of the sentences, distinguishing sentence types, like: assignments, specifications, specification-generalizations, imperative sentences, selections and questions. The grammar rules provide information to the system which words are generation words, specification words, relation words, etc. The rules even disambiguate word types, like in the question “*Is a a number?*”. The numeral codes in the grammar configuration file correspond with constants in the source code. In this way, the grammar rules are able to handover the collected information to the system. See for functions *findGrammarPath* and *scanSpecification* in class *AdminReadSentence*;
- After the information about the entered sentence is collected, the sentence is stored in the knowledge structure: The new knowledge is collected (=organized) with the existing knowledge, and the new knowledge is checked for confirmation, conflicts, ambiguity, etc. See function *addSpecification* in class *WordSpecification*;
- The integrated reasoner adds new knowledge to the knowledge structure, utilizing the rich information provided by the system. The reasoner also adds justification information to the self-generated knowledge, to justify the validity of each generated conclusion, assumption and question. See classes *AdminAssumption* and *AdminConclusion*, especially function *addSelfGeneratedSpecification* in class *AdminAssumption*.

Writing (see classes containing the word “Write”):

- After the sentence is stored, it is retrieved again from the knowledge structure, as an integrity check. See function *checkForStoreAndRetrieveIntegrity* in class *AdminWriteSpecification*. If this integrity check fails, the system either failed to store the sentence, or to retrieve (=to write) the sentence. Either way, the developer needs to find the cause of the problem;
- The self-generated knowledge – as well as the user-entered knowledge – can be written as full sentences (again), by utilizing the grammar rules in the opposite direction, from information to word – rather than from word to information - enabling the system to write grammatically correct sentences. See classes *WordWriteSentence* and *WordWriteWords*.

Error handling:

- All functions start by checking crucial arguments provided by the function call, like pointers, to avoid crashes on NULL pointers, etc. Example: “if(generalizationWordItem != NULL)”;
- The result of all other functions that can raise an error, is also checked, to stop the system;
- Function "startError" is called when the first error occurs, and all other functions call functions "addError". In this way, the error message provides information that help to trace the problem;
- If an error occurs, all newly created items in the knowledge structure are deleted. So, the system is the same as before the sentence was entered and the sentence indicator ("854,Guest>") will not be incremented. In this manner, the user can retry without problems. The sentence indicator will only be incremented if the knowledge structure has changed.

Programming:

- I use Microsoft Visual Studio C++ Express 2010 (free to download) as programming environment. And I use a text replacement tool to convert the source code to Java;
- However, Java doesn't accept Call by Reference in function calls – with the "&" before the variables – like C++ does. So, you will not find them in the C++ code, only to make conversion to Java easier;
- I only use standard C++ library functions. So, I don't use e.g. Microsoft libraries;
- I only use full variables names, to make the code more readable for others. Not "Spec", but full "Specification";
- Most variable names of booleans start with "has" or "is", like "bool **is**ExclusiveSpecification";
- Most other variables end with the name of the class type, like "**WordItem** *generalization**WordItem**".

Items and Lists:

- There are a lot of classes of type **Item** and **List** (with the names ending on **Item** or **List**). The Items are stored in Lists, like CollectionItems are stored in CollectionList;
- Each Item has a unique key, like "(123,1)". In this case, the Item that belongs to sentence number 123, and it is the first Item from that sentence. After a sentence is accepted, the sentence number is increased and a new set of Items will be created with during that next sentence;
- All Lists contain 4 single-linked lists: activeList, deactiveList, archivedList, replacedList and deletedList. The **activeList** is most commonly used;
- Assignments (see theory) can have 3 states: active, deactive and archived, with the corresponding lists: activeList, deactiveList and archivedList. An example of a deactive assignment: If the user selects a new set in the Connect-Four game, the previous choice of the user is deactivated, by putting it in the deactiveList. An example of an archived assignment: When 3 consecutive presidents of the United States are entered (see US presidents example), the most recent assignment is placed in the activeList, the previous one is placed in the deactiveList, and all older assignments others are archived in the archivedList;
- The knowledge base is designed in such a way that functions **Undo** and **Redo** are possible, similar to other software. In order to implement Undo and Redo, the information must still be present: So, instead of deleting Items that has become obsolete by the new situation, they are put in the **replacedList**. If Undo is chosen, all Items from the current sentence are archived and all Items from previous sentence are activated. If Redo is chosen, the process is the other way around;
- Only when an Item has become obsolete during the same sentence, it will be deleted (put in the **deletedList)**. After the processing of that sentence is done, the deletedList is emptied. There are a lot of Items created during the processing of each sentence, that are deleted afterwards. During the clean-up, the key of each Item is renamed in such a way that there are no gaps in the numbering: Not: (123,20), (123,29), (123,54), etc, but: (123,1), (123,2), (123,3), etc.

Knowledge structure:

- WordItem and SpecificationItem are the most important Items;
- A **WordItem** can contain multiple word types - of class **WordTypeItem** - e.g. singular noun "parent" and plural noun "parents";
- A **SpecificationItem** forms the base of knowledge structure;
- ;

Supporting classes:

to be continued in the next release...