Technical notes on Thinknowlogy:

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

- When a sentence is entered, method *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 deleted 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 methods *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 method *addSpecificationInWord* 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 *AdminReasoning* and class *AdminReasoningMisc*, and method *addSelfGeneratedSpecification* in class *AdminSpecification*.

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 method *checkIntegrityOfStoredUserSentence* in class *AdminWrite*. 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 class *WordWrite*.

Error handling:

- All methods start by checking given arguments on, for example, null pointers;
- The result of all other methods that can raise an error, is also checked, to stop the system;
- Method "startError" is called when the first error occurs, after that all other methods will call "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 ("2746,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.

Implementation:

- I only use full variables names, to make the code more readable to others. Not "Spec", but "Specification";
- Most boolean variable names start with "has" or "is", like: **is**ExclusiveSubject;
- Most other variables end with the name of the class type, like: subject**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, for example: 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 5 single-linked lists: *activeList_*, *deactiveList_*, *archivedList_*, *replacedList_* and *deletedList_*. The *activeList_* is most commonly used;
- Assignments (see theory) can have 3 states: *active*, *inactive* or *archived*, with the corresponding lists: *activeList*, *deactiveList_* and *archivedList_*. An example of an inactive assignment: If the user selects a new set in the Connect-Four game, the previous choice of the user is inactivated, by putting it in the *inactiveList_*. An example of an archived assignment: When 3 consecutive presidents of the United States are entered (see example: ambiguity\US presidents), the most recent assignment is placed in the *activeList_*, the previous one is placed in the *inactiveList_*, and all older assignments others are archived in the *archivedList_*;
- The knowledge base is designed in such a way that commands *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. So: (123,**1**), (123,**2**) and (123,**3**) rather than (123,**20**), (123,**29**) and (123,**54**).

Knowledge structure:

WordItem and SpecificationItem are the most important Item types:

- A *WordItem* can have multiple word types (class *WordTypeItem*), for example singular noun “parent” and plural noun “parents”;
- *SpecificationItem*s form the base of knowledge structure.

Programming in Controlled Natural Language:
Unlike variables in programming languages, in Thinknowlogy a word can have multiple assignments at the same time:

- "The coffee is brown";
- "The coffee is hot";
- "The coffee is sweet".

The coffee is brown, as well as hot, as well as sweet.

- To test if "coffee" has any assignment: "if the coffee is assigned then ...";
- To test if "coffee" has a particular assignment: "if the coffee is brown then ...";
- To clear all assignments of "coffee": "The coffee is clear".

The words "assigned" and "clear" are predefined words.

It is also possible to add a set to a word: "A traffic light is red, yellow or green" declares that a traffic light can only have one state (either red, yellow or green).

- "The traffic light is red" is similar to "traffic_light = red" in programming languages;
- But if you then state: "The traffic light is green", then the previous assignment is cleared by the system and state "green" will be assigned, similar to "traffic_light = green".

If you add "The traffic light is bright", it will not affect its red, yellow or green state, because "bright" is not collected with "red", "yellow" and "green".

Except for "assigned", "clear" and the mentioned colors, none of the adjectives and nouns mentioned above are defined in the system. Some colors are defined in the file data/English/common knowledge.txt, which is read during start-up.

There is no need to declare words first. You are free to use any word, as long as it isn't a keyword defined in the grammar file. Some keywords defined in the grammar file have a special function. Examples: Verb "read" and noun "file" are used to read files, verb "display" is used to display info, and the verb "solve" is used to start a problem solving algorithm in the Connect-Four game.

Reasoning in Controlled Natural Language:
During the reasoning process, *JustificationItem*s are created, which are stored in the *JustificationList* of the involved word. The justification lists are updated if needed. And a justification report can be displayed at any moment. In this way, each step of the reasoning process is justified.

Supporting classes:

- The classes that support class *AdminItem* start with the name *Admin*…, like ***Admin****Collection* and ***Admin****Imperative*;
- The supporting classes of *WordItem* are ***Word****Question*, ***Word****Specification* and ***Word****Write*.

Grammar file:

The grammar rules are written in a kind of Extended Backus–Naur Form (EBNF):
[..] = option
[.. | ..] = multiple options
{ .. | .. } = choice
(..) = optional for reading, will not be used for writing

The grammar file should be read from the bottom of the file to the top, starting with:

```
:4000  $sentence                { statement_sentence | question_sentence | selection_sentence }
```

- Each definition word in the grammar file starts with '$', like "$sentence". And after a definition word the definition itself is given;
- A sentence is either a statement, or a question, or a selection.

A statement sentence is defined as:

```
:4010  $statement_sentence   [ answer_Yes_or_No symbol_comma ] statement [ more_statements ]
                             [ symbol_colon | symbol_exclamation_mark ]
```

- A statement sentence can start with an answer (‘yes’ or ‘no’), followed by a comma. The statement itself is mandatory, which can end with either a colon or an exclamation mark.

The definition of both colon and exclamation mark can be found at the top of the file:

```
@1     :101    $symbol_colon                .
@1     :102    $symbol_exclamation_mark     !
```

- These lines start with '@1'. The number after '@' character corresponds with the word type number in the software. (See constants file.) So, the value of '@1' corresponds with constant WORD_TYPE_SYMBOL, which has value ‘1’. When reading this grammar file, value ‘1’ will be assigned to variable “unsigned short wordTypeNr_” in class *WordTypeItem*, which indicates that this word is in fact a symbol;

- A number that starts with ':' – like ':101' – corresponds with the value of the word parameter in the software, if the value is lower than 4000. When reading this grammar file, value ‘101’ will be assigned to variable “unsigned short wordParameter_” of class *WordItem*, which corresponds with constant WORD_PARAMETER_SYMBOL_COLON;

- If the number is 4000 or higher – like ':4010' of the definition mentioned above – it corresponds with the value of the grammar parameter in the software. When reading this grammar file, value ‘4010’ will be assigned to variable “unsigned short grammarParameter_” of class *GrammarItem*, which corresponds with constant GRAMMAR_STATEMENT.

If a grammar definition doesn’t start with a value, no variable is assigned in the software.