Comparison Function with Right Answer for Software Design Support Tool Perseus

Tetsuro Kakeshita and Yuki Shibata
Graduate School of Information Science, Saga University, Saga 840-8502, Japan

Keywords: Software Design Education, Engineering Design, Automatic Comparison, Software Tool, XML, Levenshtein Distance, e-Learning.

Abstract: Systematic software design is a typical engineering design problem which has multiple solutions. We have developed a software design support tool Perseus for systematic software design education. In this paper, we develop and evaluate the comparison function for Perseus between student's answer and a set of multiple right answers. Perseus represents software design by a tree structure. The comparison function automatically makes correspondence between tree nodes using tree matching. The matching between nodes is performed by utilizing Levenshtein distance. Considering the nature of software design, the comparison function utilizes various parameters such as alternative answer, keyword, NG word, incorrect answer and integrates the adjustment function of the threshold value for comparison. We also develop a right answer editor named Pras.Edit. We perform an evaluation of the comparison function using 20 student answers. The number of mistakes detected by the improved comparison function is approximately 3 times larger than that of the manual checking. Furthermore 93.1% of the detected mistakes were correct.

1 INTRODUCTION

Software design is a typical engineering design problem and greatly affects maintainability, reusability and efficiency of computer software (McConnell, 2004). It is unusual that only one optimum solution exists in software design. Thus software design is a complex process which requires multiple iterations.

Systematic design of computer software is becoming more and more important due to increasing scale and complexity of software. For example, ISO/IEC 12207 defines standard process for software life cycle (ISO, 2008). Software design is an important process in this international standard.

When we teach software design at a university, practical exercise is necessary in addition to the lecture. Review of the software design produced by the students is the core of the exercise. We have proposed the software design support tool Perseus in order to support such exercise (Kakeshita and Fujisaki, 2006). Perseus provides the editing and review functions of various components of software design such as module, routine, algorithm and data structure.

A problem of Perseus was that a teacher has to manually review and correct the design produced by the students. This becomes a big problem when the number of students is getting larger and software design becomes more complex. It often happens that students make similar mistakes within a same class. We thus propose the comparison function with the right answer in this paper.

Software design is represented by a tree in Perseus. The right answer provided by the teacher is also a tree whose node contains alternative right answers, keywords, NG words and incorrect answers. The comparison function utilizes Levenshtein distance (Levenshtein, 1966) in order to automatically make correspondence between the student's answer and the right answer. Each node of the student answer is evaluated to be correct if it satisfies the following conditions for the corresponding node of the right answer: (1) the node matches to at least one of the alternative answers; (2) it contains all of the keywords and none of the NG words; and (3) it does not match to none of the incorrect answer.

Furthermore the comparison function is extended to handle multiple right answers. A teacher can register multiple right answer files to the system.

Kakeshita, T. and Shibata, Y.
Comparison Function with Right Answer for Software Design Support Tool Perseus.
In *Proceedings of the 8th International Conference on Computer Supported Education (CSEDU 2016) - Volume 1*, pages 259-266
ISBN: 978-989-758-179-3

The system automatically performs comparison between the student answer and each of the right answer.

We also developed a right answer editor named Pras.Edit to edit the right answer and the comments added to the incorrect answers.

This paper is organized as follows. Section 2 introduces the basic function and features of Perseus. We shall explain overview of the comparison function and the basic algorithm of the comparison function in Section 3. The right answer editor Pras.Edit is introduced in Section 4. We improve the original comparison function by introducing various parameters such as alternative answer and keyword. We shall evaluate the comparison function and analyze the impact of improvement in Section 5. Difference with the related tools will be explained in Section 6.

2 SOFTWARE DESIGN SUPPORT TOOL PERSEUS

Perseus is a software design support tool mainly designed for students and beginners of software design. Perseus supports both of structured design and object oriented design. It also supports various software design activities such as algorithm, data structure, routine and module design. Students can separate software design and coding processes by utilizing the design as high level comments of the source code.

We utilize Perseus at various exercises of two courses at our university: Data Structure and Algorithm, and Software Engineering.

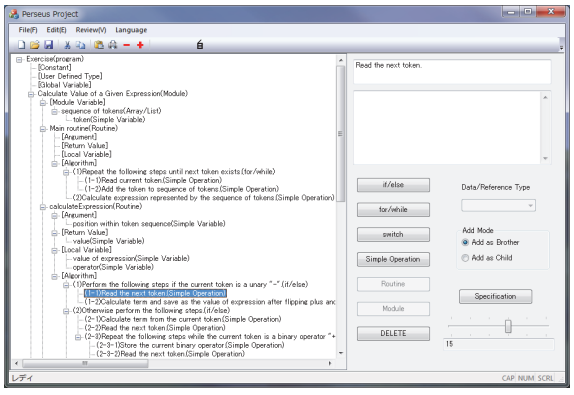

Figure 1: Perseus User Interface.

The basic function of Perseus is the editing function of the design tree and the review function. Figure 1 illustrates the user interface of Perseus.

The Tree View of Perseus shows the design tree. A student can select a node of the design tree to manipulate the tree. A student can modify text of the selected node; create a new subtree representing a module, routine, data structure and statement; delete, copy, cut or paste of a subtree; expand and shrink the design tree.

Perseus restricts the type of a new subtree depending on the selected node in order to maintain consistency of the design tree. For example, a subtree representing a compound statement can only be added to a node of a subtree representing algorithm. Similarly, certain deletion, copy, cut and paste operations are prohibited for predefined nodes and subtrees. Buttons corresponding to the prohibited operations are disabled as illustrated in Figure 1. Expansion and shrinking of a subtree can be executed at any subtree.

A teacher can add an arbitrary text to a node of the design tree using the review function. The added text is stored as a review comment of the corresponding node. Perseus also provides a function to store the comments to a CSV file. The stored comments can be classified and can be added to an arbitrary node of the design tree. The review function is designed to facilitate reuse of the comment text.

Perseus utilizes XML data to represent software design tree. Each of the module, routine, data structure and algorithm is represented by a subtree whose root has the same name of the subtree. Each node is assigned a node-id, comment tag and review-comment tag.

3 COMPARISON FUNCTION WITH RIGHT ANSWER

Although Perseus provides the review function, a teacher must manually add review comments. Teacher's workload will increase according to the increase of the complexity of the design tree and the number of students submitting the design tree. The comparison function is designed to automatically compare the design tree with the right answer provided by the teacher.

3.1 Overview

The comparison function is developed for semantic checking of the design tree which the student creates. Since software design may have multiple right answers, Perseus allows registering multiple right

answer files to the system. The comparison function compares the student tree and each of the registered design trees representing right answers. Perseus automatically selects a registered design tree which is most similar to the student tree based on the Levenshtein distance.

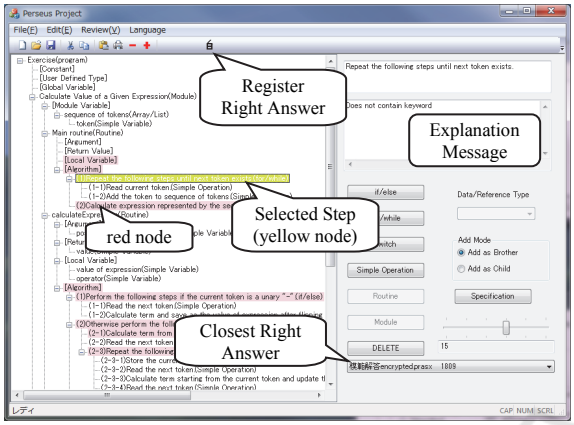

Figure 2: Comparison Result.

In Figure 2, the design tree created by the student is shown. The red nodes are the nodes different from the right answer. The yellow node is the node which do not contain keywords defined on the corresponding node of the right answer. When a student selects a colored node, explanation messages of the detected difference are displayed in the text box at the right side of the window. There are four types of messages for the red nodes depending on the types of the mismatches as explained below.

Type 1. Extra child node exists in the student's answer which does not correspond to a node in the right answer.

Type 2. Child node is missing in the student's answer which corresponds to a node in the right answer.

Type 3. There is no node in the right answer corresponding to the selected node.

Type 4. Levenshtein distance from the corresponding node in the right answer exceeds the predefined threshold.

The "Closest Right Answer" combo box in Figure 2 contains the name of the right answer file which is most similar to the student answer. A Perseus user can also select an arbitrary right answer file using the combo box to compare with the student answer. In either case, the user can view the detail of the comparison result and the right answer at a window illustrated in Figure 3. The comparison result is represented by the comparison table between student design and the right answer. The detailed definition of the comparison table will be explained in Section 3.3.

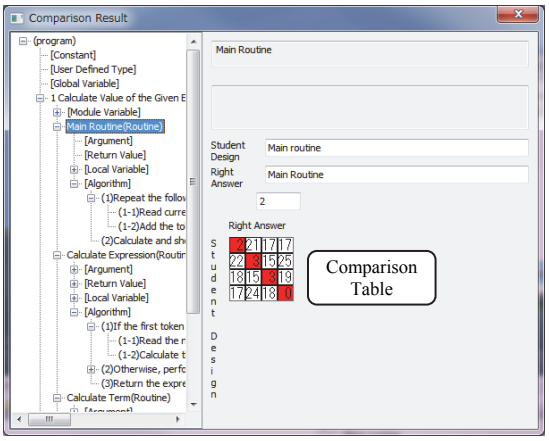

Figure 3: Detailed Comparison Result.

The comparison function is automatically executed when the user registers a right answer file or when the system reads a new student file. The registration function is executed by pressing the "Register Right Answer" button as illustrated in Figure 2.

Figure 4 illustrates the registration window of right answer files. A user can add and delete a right answer file at the window. The "compare" button is used to compare a selected right answer and the student design. The "delete comments" button is used to delete all the explanation messages added to the student answer.

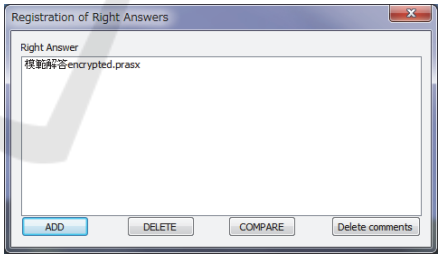

Figure 4: Registration of Right Answers.

3.2 Levenshtein Distance

Each node of a design tree carries a string so that REMEST utilizes the notion of Levenshtein distance (Levenshtein, 1966) in order to evaluate the distance between the two corresponding nodes of the design trees. The Levenshtein distance is defined by the minimum number of edit operations, i.e. insertion or deletion of a character, to convert a string to another string. We utilize the famous algorithm in order to compute the Levenshtein distance between two strings using dynamic programming.

3.3 Tree Matching

We shall propose the tree matching algorithm for the comparison function in this section. The algorithm is further extended to handle multiple right answers of a software design problem. Such extension can be realized by a simple repetition of the proposed tree matching algorithm between a student tree and each of the right answer.

Although two corresponding nodes of the design trees can be compared using Levenshtein distance, we need a tree matching algorithm in order to compare the student's answer and the right answer since a design tree can be regarded as a tree.

Let S and M be the trees representing design trees of the student's answer and the right answer respectively. Let s be the root node of S and $S_1, \cdots, S_n$ be the subtrees of S whose root node is a child of s. Similarly, let m be the root node of M and $M_1, \cdots, M_n$ be the subtrees of M whose root node is a child of m. We can assume, without loss of generality, that the numbers of subtrees of S and M are the same by adding empty subtrees to either S or M.

The distance $d(s,m)$ between two nodes s and m is defined by the Levenshtein distance of the strings representing the nodes. The distance between S_i and M_j can be defined by the following formulae if either of S_i or M_j is an empty tree $\emptyset$.

$$d(S_i, \emptyset) = \sum_{t \in S_i} |t|,\ d(\emptyset, M_j) = \sum_{t \in M_j} |t|$$

Here t is a node belonging to S_i or M_j. $|t|$ is the number of characters in t.

Assuming the above definitions, we can now define the distance between S and M in the case that S_i and M_j are not empty trees. Let p be a permutation of $1, \cdots, n$ and $p(i)$ be the i-th value of p. The distance between S and M is defined by the following formula representing the minimum distance among all permutation p.

$$d(S,M) = d(s,m) + \min_{p} \sum_{i=i}^{n} d(S_i, M_{p(i)})$$

The distance between two subtrees S_i and M_j can also be calculated recursively by applying the above formula.

Now we can explain the tree matching algorithm used by the comparison function. The algorithm computes the minimum distance based on the above formulae and utilizes the greedy method to identify the optimal permutation as explained below.

1 Read S and M.

2 Calculate the comparison table $D[i,j]$ whose element represents the distance $\mathrm{d}(S_i, M_j)$ between S_i and M_j.

3 Repeat the following steps until the table $D[i,j]$ becomes empty.

3.1 Identify the minimum distance $D_{min} = D[i_{min}, j_{min}]$ in $D[i,j]$.

3.2 Let $S_{i_{min}}$ and $M_{j_{min}}$ be the corresponding subtrees.

3.3 Output and record the above pair of subtrees with the distance $\mathrm{d}(S,M)$ between them.

3.4 Delete row i_{min} and column j_{min} from table $D[i,j]$.

Step 2 of the above algorithm is executed recursively according to the definition of the distance. If there exist more than one pairs of i_{min} and j_{min} with the minimum distance D_{min} in Step 3.1, then the permutation p with the minimum sum of the distances is selected. Perseus maintains the list of corresponding subtrees and the distance as defined in Step 3.3. The corresponding subtrees are represented by the position number assigned to the root node of the subtree.

4 RIGHT ANSWER EDITOR Pras.Edit

4.1 Overview

Pras.Edit (Perseus Right Answer Editor) is a software tool to create and edit the right answer used for the comparison function. The tool provides two major windows: the main window (Figure 5) and the detailed setting window (Figure 6). The two windows can be switched by pressing a button at each window.

The right answer is represented by an XML file and contains additional information for comparison such as alternative answer, keywords and threshold value to compare with the calculated Levenshtein distance. The right answer file is created by the teacher and is distributed to the students so that the file is encrypted before distribution. Then a student can check his own design by himself.

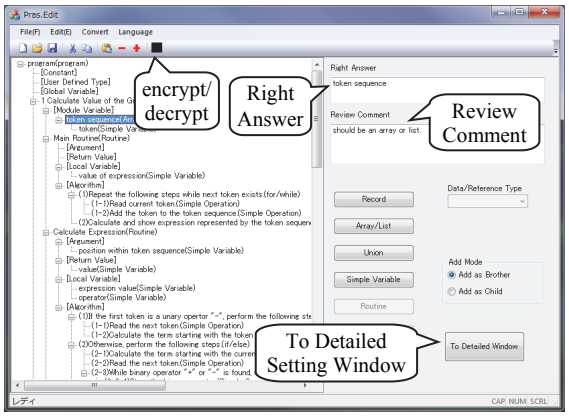

Figure 5: Main Window of Pras.Edit.

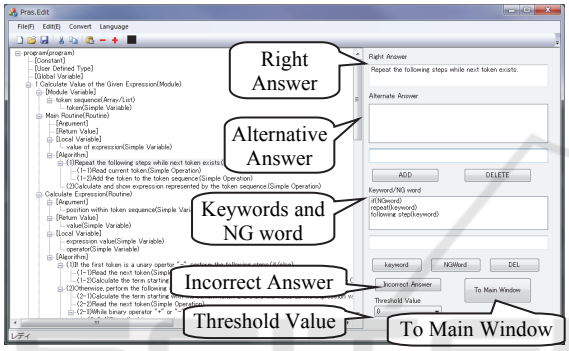

Figure 6: Detailed Setting Window of Pras.Edit.

4.2 Functions

The basic function of Pras.Edit is the editing functions of the right answer. The function is essentially the same as the editing function of the design tree explained in Section 2.

Pras.Edit also provides the following editing functions to add various types of information to the right answer file.

1. Editing function of the review comments added to the incorrect nodes of the design tree.
2. Editing function of keywords and NG words. A node of the right answer tree can have an arbitrary number of keywords and NG words. The comparison function is extended so that the corresponding node is regarded as correct only when the node contains all of the keywords and none of the NG words associated to the corresponding node of the right answer.
3. Editing function of the threshold value at each node of the right answer. A teacher can adjust coefficient of the threshold value of each node by the function. The threshold value is used in order to compare with the Levenshtein distance between the right answer and the corresponding node of the design tree. Thus the editing function can be utilized to control strictness of the comparison.
4. Editing function of alternative answers to the right answer file. Each node of the right answer file can have an arbitrary number of the alternative answers. The comparison function is extended so that the student's design tree is correct when the tree matches to an alternative answer having the minimum Levenshtein distance.
5. Editing function of incorrect answers. A node of the design tree is regarded as incorrect when the node matches to an incorrect answer. This function is utilized to detect typical design mistakes of the student.

Editing function of the review comment and the threshold value can be executed at the main window (Figure 5). On the other hand, Keywords, NG words, alternative and incorrect answers can be edited at the detailed setting window (Figure 6).

The followings are the miscellaneous functions of Pras.Edit. These functions can be executed at the main window.

1. Conversion function from a Perseus design tree file to the corresponding right answer file
2. Encrypt and decrypt functions of the right answer file. A teacher cannot edit the right answer file when the file is encrypted. The encrypted right answer file can be distributed to the students and is useful to self-check the student answers by utilizing the comparison function.

4.3 XML Structure

As explained in the previous section, the right answer file contains various data associated to the right answer. The right answer is represented by an XML file. The DTD is designed so that various design elements such as module, routine, data structure and control structure can be represented in a flexible manner.

```
<!ELEMENT design (right-answer, error-message, alternate-answer, keywords)>
<!ATTLIST design expand CDATE #IMPLIED
                 incorrect CDATE #IMPLIED
                 LEVEL CDATE #IMPLIED>
<!ELEMENT right-answer (#PCDATE)>
<!ELEMENT errormessage(#PCDATE)>
<!ELEMENT alternate-answers (alternate-answer)>
<!ELEMENT alternate-answer (#PCDATE)>
<!ELEMENT keywords (keyword, NGword)>
<!ELEMENT keyword (#PCDATE)>
<!ELEMENT NGword (#PCDATE)>
```

Figure 7: DTD of Each Node of the Right Answer.

On the other hand, the associated data such as review comments, keywords, NG words, threshold values, alternative answers and incorrect answers, is represented by a node of the right answer. The structure of the node is defined by the DTD illustrated in Figure 7.

5 EVALUATION EXPERIMENT

We conducted an evaluation experiment to analyze correctness of the original comparison function and the extension of the comparison function.

5.1 Outline of the Experiment

We utilize 20 student answers of a software design exercise at our university. These answers were randomly selected among 68 answers so that the number of answers becomes same at each evaluation score. Students are assigned a detailed software specification and create design tree composed of modules and routines based on the stepwise refinement technique taught at the class. The supplied software specification represents a credit management system at university which the students have enough familiarity.

We evaluate the effect of the comparison function by comparing the difference among the following three cases.

Case 1. Design errors detected by manual checking of the teacher

Case 2. Design errors detected by the original comparison function, defined in Section 3, utilizing the Levenshtein distance only

Case 3. Design errors detected by the improved comparison function, explained in Section 4.2, utilizing adjustment of threshold value, keyword, NG word, alternative and incorrect answers as well as the Levenshtein distance

Table 1: Coefficient Values for Case 3.

Type of Node	Coefficient Value
Data Structure	0.25
Predefined Routine by Teacher	0.5
Nodes Fully Described by Student	2.0
Others	1.0

The detailed parameters of cases 2 and 3 are selected so that the number of correctly detected design errors is maximized and the number of incorrectly detected nodes is minimized. The threshold value for the comparison is 15 for Case 2. The coefficient of each node is defined as represented in Table 1.

There are two routines which students frequently made mistakes. These routines are defined as incorrect answers. We also defined distinctive keywords for major nodes for Case 3.

5.2 Overall Evaluation Result

We first compare the design errors detected automatically by Cases 2 or 3 with the design errors detected manually by Case 1 defined above. There are four cases of the comparison result.

Case A. The design errors can be detected by both of Case 1 and Case 2/3.

Case B. The design errors can be detected only by Case 1, but cannot be detected by Case 2/3.

Case C. The design errors can be detected only by Case 2/3, but cannot be detected by Case 1.

Case D. The design errors detected by Case 2/3 are incorrect.

Granularity of the detected errors is different between Case 1 and the other cases because of the difference of manual checking and automatic checking. It is our experience that a manually detected error corresponds to approximately 5 to 6 errors detected automatically.

Table 2 summarizes the number of the detected design errors of the three cases of the error detection classified by the type of errors defined in Cases A to D. The numbers in parentheses represent the number of errors manually detected by Case 1. Other numbers represent the number of errors detected automatically by Cases 2 or 3.

Table 2: Distribution of Detected Design Errors.

Type of Errors	Comparison between		Difference
	Cases 1 & 2	Cases 1 & 3	
Case A	379 (68)	419 (71)	+10%
Case B	(4)	(1)	-75%
Case C	664	851	+28%
Case D	174	94	-46%
Total	1221 (72)	1364 (72)	+12%

There are 72 design errors detected manually by the teacher. The original comparison function (Case 2) detects 94.4% among them, while the improved comparison function (Case 3) detects 98.6% of them. The numbers of design errors which cannot be detected by the comparison function were reduced from 4 to 1 by improving the comparison function.

The number of correct design errors detected by Case 2 is 1047 so that 85.7% of the detected errors

are correct even in the original comparison function. The percentage can be further improved to 93.1% by integrating various techniques explained in Section 4.2. The numbers of incorrect errors are reduced by 46% as can be observed by the difference at Case D. This is mainly due to the effect of the adjustment function of the threshold values.

The readers should also note that there are a significant number of design errors which could not be detected manually by the teacher both in Cases 2 and 3 by observing the row of Case C. This is an advantage of utilizing automatic error detection proposed by the comparison function.

It can also be said that the errors detected by the comparison function is more concrete than manual detection. We often observe that students prefer concrete instruction so that automatic error detection method is useful to improve software design education.

5.3 Detailed Analysis of the Improved Comparison Functions

We discuss the detailed impact of the associated functions of the original comparison function in this section. The discussion will clarify the reason of the improvement of the original comparison function. We also analyze the incorrect detection by the improved comparison function.

5.3.1 Adjustment of Threshold Value

We detect 158 design errors by adjusting the threshold values. This was achieved mainly because slight mistakes can be detected at the design of data structure and algorithm by utilizing coefficient less than 1. Another reason is that tree matching can be performed more accurately so that child nodes of the trees can be matched correctly.

The number of incorrect design nodes was 80 for the original comparison function. But the number was reduced to 12 by utilizing the adjustment function. Main reason of the remaining nodes is the missing or fluctuation of description in the student answer. We consider that the number of these nodes can be further reduced by integrating alternative answer and proper keywords or NG words.

5.3.2 Incorrect Answers

We detect 30 design errors by utilizing incorrect answers. No error was detected incorrectly. 11 incorrect errors detected by the original comparison function were not detected by introducing the incorrect answers. This implies that proper setting of incorrect answers is a powerful means to detect more design errors without increasing the number of incorrect errors.

However we experienced that addition of alternative answer may cause incorrect matching of the nodes with registered incorrect answer. Although such incorrect matching can be avoided by careful definition of the alternative answer, we are investigating a systematic means in order to avoid such incorrect matching.

5.3.3 Keywords and NG Words

We detect 49 design errors by utilizing keywords and NG words. Among them, 8 were incorrect. One reason of the incorrect detection is spelling mistake within the student answer. Another reason is the checking of the keyword within an incorrect node due to incorrect tree matching. The incorrect tree matching can be reduced by utilizing alternative answer.

21 incorrect errors detected by the original comparison function were not detected by introducing the keywords and NG words. The effect of keywords and NG words exceeds the effect of incorrect answers. This is mainly because keywords and NG words can be adopted more widely to detect design errors, while an incorrect answer only represents a specific design error.

5.3.4 Incorrect Detection of Design Errors of the Improved Comparison Function

94 design errors were incorrectly detected by the improved comparison function. These errors can be classified into 5 types as shown in Table 3.

Table 3: Classification of Incorrect Design Errors of the Improved Comparison Function.

Type of Incorrect Errors	# of Errors
Different Description	15
Fluctuation of Description	12
Incorrect Tree Matching	30
Incorrect Matching of Nodes	7
Others	30

The incorrect detection due to the difference of description can be reduced by alternative answers. Many of the incorrect detection due to fluctuation of description can be reduced by adding appropriate keywords and NG words. 16 of the incorrect detection due to incorrect tree matching can be improved also by defining alternative answers. The incorrect detection of design errors caused by

incorrect matching of nodes can be reduced by utilizing NG words.

However the remaining incorrect detection cannot be reduced by the proposed method. Such incorrect detection includes the following cases. The reduction of incorrect detection is left as a future research topic.

- Incorrect matching of design tree for data structure consisting of multiple subtrees
- Misspelling of student

6 RELATED WORKS

There are many software design tools such as Astah* Professional (Change Vision) available for professional use. Although few of them are developed for educational use, there is an educational UML design evaluation tool utilizing various software metrics (Sato, Tamura and Ueda, 2008). However the messages produced by the tool tend to be rather abstract for the students. The comparison function proposed in this paper can provide more concrete information about the incorrect nodes.

7 CONCLUSION AND FUTURE VISION

We developed and evaluated the comparison function of software design support tool Perseus in this paper. Although the original comparison function can detect more design errors than manual checking by the teacher, it can be further improved by adding alternative answer, keyword, NG word, adjustment of threshold value, and incorrect answer. The improved comparison function will be a powerful support tool for the teachers of various aspects of software design.

Current limitation of the comparison function is the workload of fine tuning of the right answer. We are currently developing a software tool to improve the right answer during the reviewing process of the student's design tree by integrating Perseus and Pras.Edit.

We are currently developing a series of education tools for other process of software development. A tool named REMEST (Kakeshita and Yamashita, 2015) is developed for the education of software requirement management. A tool named pgtracer (Kakeshita, Yanagita, Ohta and Ohtsuki, 2015) is developed for programming education. These tools will be utilized to improve education of systematic development of computer software. They are also useful to collect the learning log of the students. The analysis of the collected data will be valuable to analyze and evaluate the understanding level of each student. It will also be useful to quantitatively analyze the effect of various learning techniques and technologies.

Our future vision is to integrate various education support tools to develop a systematic learning environment covering the entire process of software development including requirement management, software design and computer programming.

REFERENCES

Change Vision, Astah* professional, http://astah.net/

ISO, 2008. ISO/IEC 12207:2008, Systems and software engineering – Software life cycle processes (to be revised).

Kakeshita, T., Fujisaki, T., 2006. Perseus: An educational support tool for systematic software design and algorithm construction, *Proc. 19th Conf. on Software Engineering Education and Training* (CSEE&T), pp. 13-16.

Kakeshita, T., Yamashita, S., 2015. A requirement management education support tool for requirement elicitation process of REBOK, *Proc. 3rd Int. Conf. on Applied Computing & Information Technology* (ACIT 2015), Software Engineering Track, pp. 41-46.

Kakeshita, T., Yanagita, R., Ohta, K., 2015. A programming education support tool pgtracer utilizing fill-in-the-blank questions: overview and student functions, *Proc. 2nd Int. Conf. on Education Reform and Modern Mana*gement (ERMM 2015), pp. 164-167.

Kakeshita, T., Ohta, K., Yanagita, R., Ohtsuki, M., 2015. A programming education support tool pgtracer utilizing fill-in-the-blank questions: teacher functions, *Proc. 2nd Int. Conf. on Education Reform and Modern Management* (ERMM 2015), pp. 168-171.

Levenshtein, Vladimir I., 1966. Binary codes capable of correcting deletions, insertions, and reversals, *Soviet Physics Doklady*, 10 (8), pp. 707-710.

McConnell, S., 2004. Code Complete: A Practical Handbook of Software Construction, *2nd Edition*, Microsoft Press.

Sato, M., Tamura, S., Ueda, Y., 2008, A study of quality evaluation model for UML design, *Journal of Information Processing*, Vol. 49, No. 7, pp. 2319-2327. (in Japanese).