Jiang, B. et al. (Eds.) (2025). Proceedings of the 33rd International Conference on Computers in Education. Asia-Pacific Society for Computers in Education

Support System for Understanding Class Design with Reusability and Maintainability

Kazuma KUWADA[a*] & Tomoko KOJIRI[b]
[a]*Graduate School of Science and Engineering, Kansai University, Japan*
[b]*Faculty of Engineering Science, Kansai University, Japan*
*k057907@kansai-u.ac.jp

Abstract:
In object-oriented programming (OOP), creating class designs with high reusability and maintainability is important. Structure with high reusability and maintainability can be created by using hierarchical structures based on inheritance. Hierarchical structure can be created by defining abstract classes that abstract similar classes. However, OOP beginners may struggle to identify which similarities to abstract to achieve reusability and maintainability, potentially preventing them from creating such structure. This research aims to teach OOP beginners to recognize similarities between classes, enabling them to create structure that satisfy reusability and maintainability requirements.

Keywords: object-oriented programming, reusability, maintainability, similarity between classes

1. Introduction

In object-oriented programming (OOP), it is important to create class design that is highly reusable and maintainable. Class design with high reusability and maintainability can be achieved by creating hierarchical structures using inheritance between classes. To understand and leverage the benefits of OOP, it is important to understand the relationships between classes that should be set in a hierarchical structure.

A study (Takekawa & Nakabayashi, 2024) aimed at helping students understand the benefits of object-oriented languages proposed a learning method that fosters understanding of the causal relationship between fundamental object-oriented concepts and their benefits. This was achieved by having students compare object-oriented and procedural languages, consider behavior in extension tasks, and reflect on the role of polymorphism. This approach uses a pre-existing hierarchical structure for learning. Thus, although learners are able to understand the benefits of expansion in relation to class behavior, it is not guaranteed that they themselves to create structures that leverage these benefits.

A hierarchical structure can be created when multiple classes share common attributes or methods. In creating hierarchical structure, two types of classes are created; one is called superclass that holds only the common elements and the other is called subclass that holds only the unique elements. The superclass is regarded as the abstraction of the subclasses.

To introduce such abstraction into design, it is necessary to understand the common elements of classes. Research exists that aimed at extracting common elements of classes (Wang, Li, Ma, Xia & Jin, 2020). This research extracts the common methods by representing their codes as a graph structure and compares them. This research focuses only on the similarities of methods. However, in order to create the hierarchical structure of highly reusable and maintainable, not only methods but only classes themselves need to be examined. In addition, in this research, the system extracts the similarity so that the programmer is not trained to create the hierarchical structure. This research aims to teach OOP beginners (learners) to recognize similarities between classes that can get benefit from creating the hierarchical structure, enabling them to create structure that satisfy reusability and maintainability requirements.

2. Approach

2.1 Structure with Reusability and Maintainability, and Similarity between Classes

High reusability means that existing code can be reused. High maintainability means that changes can be made easily. In OOP, a good structure is one in which common process between classes is implemented in the superclass, and classes that call subclasses refer only to the superclass type. By implementing common process in a superclass, it becomes reusable when adding a class that contain the same process. In addition, since the class that calls the common process only refer to the superclass, to add a class that contain the same process does not make any change to the calling class, which leads to high maintainability.

Here we show the example of reusable and maintainable. Let's assume *PDFExporter* class and *CSVExporter* class have common process *export* that calls *formatData* and called by *Client* class. The concrete process of *formatData* is different between them. For such classes Figure 1 is the reusable and maintainable structure.

To enhance reusability through abstraction, methods with similar processing sequences must be defined across the classes being abstracted. Implementing such methods in the superclass improves reusability. To enhance maintainability, the relationship with the calling class (dependency, aggregation, etc.) must be consistent. Referencing them as similar types improves maintainability. In this study, these are considered conditions for reusability and maintainability.

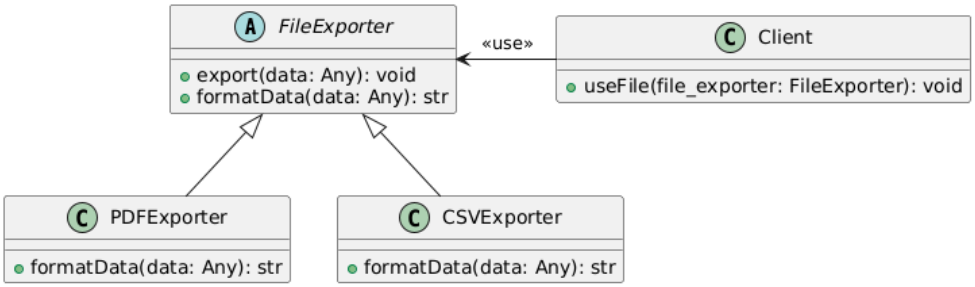

Figure 1. Highly Reusable and Maintainable Structure

2.2 Activities for Learning Similarities between Classes

Learners who mistakenly select class groups that can form hierarchical structures do not understand that the resulting hierarchy does not enhance reusability or maintainability. To make them verify whether the created hierarchical structure possesses reusability and maintainability, it is effective to assign tasks involving reuse and maintenance of created class structure. In this study, we provide learners with a set of classes that do not use hierarchical structures and have them create hierarchical structures. Subsequently, we assign extension tasks based on unrecognized similarities. By executing the extension tasks, we expect learners to notify whether the hierarchical structures they created possess reusability and maintainability.

3. Class Similarity Learning Support System

We developed a system as shown in Figure 2 enabling learner to perform the learning activities described in Section2.2. In this system, first, the learner creates abstract design of the given non-abstracted class diagram. Next, the system judges whether the abstracted class satisfy similarity conditions and presents an extension task according to the dis-satisfied conditions. If the condition for reusability is not met, it presents a task for adding a class with a set of methods identical to those possessed by an abstracted class. In order to cope with the task, the learner must describe all the methods so that the learner recognizes that his/her design is not highly reusable. If the condition for maintainability is not met, it presents a task for adding a class with relationships similar to those between the abstracted class and its calling classes. In order to cope with the task, the learner must implement each class in a form that cannot be

referenced as a superclass, which makes him/her recognize that his/her design is not a highly maintainable. After completing the extension task, the learner receives feedback that indicates the degree of improvement achieved through abstraction and suggests whether modifications are necessary.

Figure 3 shows the interface for creating the abstract design. This interface consists of a class diagram display area and a method display area. The class diagram display area is for editing the class design and the method display area is for modifying the method as the flower chart form. When the system starts, the class diagram of the non-abstracted design is shown in the class diagram display area. The learner can create abstract class by pressing the class add button. Additionally, learner can edit a class's fields and methods by pressing the button in the upper-right corner of that class. When the learner clicks a method name, the method's processing sequence is displayed in flowchart format in his/her selected half of the split of the method display area. When the learner clicks a processing node where he/she wishes to add further processing, the processing addition screen appears, allowing him/her to add processing by describing the desired operation. When the learner presses the complete button, the extension task appears at the top of the screen and the learner is asked to modify the class diagram using the same interface. When the learner presses the complete button again, feedback is displayed.

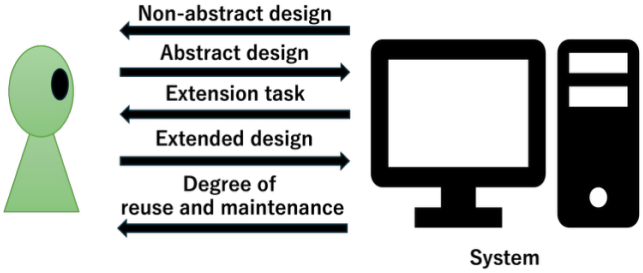

Figure 2. Class Similarity Learning Support System

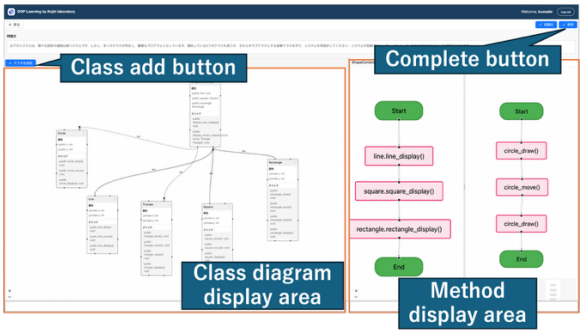

Figure 3. Interface for Creating Abstract Class

4. Conclusion

In this paper, we proposed a system designed for beginners in object-oriented programming (OOP) that teaches them to learn highly reusable and maintainable structures by learning similarities between classes. Our future work is to conduct experiments evaluating the effectiveness of the proposed system.

References

Takekawa, N., & Nakabayashi, K. (2024). A learning method for novice learners on object-oriented programming focusing on its extendability. *Transactions of Japanese Society for Information and Systems in Education*, *41*(3), pp. 224–239. (in Japanese).

Wang, W., Li, G., Ma, B., Xia, X., & Jin, Z. (2020). Detecting code clones with graph neural network and flow-augmented abstract syntax tree. *arXiv preprint arXiv:2002.08653.*