A Method for Efficient Extensibility Improvements in Embedded Software Evolution

Takanori Sasaki[1,3*], Nobukazu Yoshioka[2], Yasuyuki Tahara[3], Akihiko Ohsuga[3]

[1] Canon Incorporation, Tokyo, Japan.
[2] National Institute of Informatics, Tokyo.
[3] University of Electro-Communications, Tokyo.

[*] Corresponding author. Email: tsasaki@nii.ac.jp
Manuscript submitted July 9, 2015; accepted July 24, 2015.
doi: 10.17706/jsw.10.12.1375-1388

Abstract: Lightweight development processes like Agile have emerged in response to rapidly changing market requirements. However, software evolution processes including Agile are inadequate for software in embedded systems, as software undergoes frequent refactoring, targeting only immediate requirements. As a result, maintainability decreases because the system is not designed to respond to changes in the associated hardware. In this paper, we propose a method for improving extensibility. We also propose a technique for detecting and suggesting extension design patterns automatically. Our approach is based on analyses of the call graph and inheritance structure of source code to identify a layer structure that is specific to embedded software. These techniques provide us with objective and quantitative information about extensibility. We applied the proposed method to an actual product's code continuously and could verify an improvement in the system's extensibility.

Key words: Embedded system, agile, extensibility, legacy code.

1. Introduction

In software technology, innovation is accelerating and new product concepts are appearing such as mobile, cloud, augmented reality, and so on. As a result, the software system is becoming more and more complex, while market needs are also changing in parallel. Therefore, if the market needs change during development or immediately after production, it becomes difficult to make a profit using the conventional processes like the Waterfall Model, in which products are usually made after development process such as accurately forecasting future market trends, making specifications, making designs, implementing code, and testing. In response to such challenges, lightweight development processes like Agile [1] are becoming popular. These processes aim to finish developing only simple features and mechanisms, striving for shorter time-to-market without needing to forecast future market trends accurately or design mechanisms in advance. Once feedback from the market is received, additional features are developed. Thereby it minimizes the gap between market needs and product features. These processes are classified as an evolutionary software development process. It develops features needed in the next release with refactoring. Therefore, it needs to improve in the long term [1].

However, an architecture will often be broken by choosing the correct local design in an agile development process due to the above reason. To prevent this, an agile process also needs

high-extensibility design. To design considering extensibility depends on the programmer in the case of implementing needed features only.

Such a process is especially inadequate for software in embedded systems. As the system is not designed to respond to changes in the associated hardware, its maintainability gradually decreases. The reason for such behavior is that embedded software usually has numerous patches in order to accommodate hardware changes without inflating costs (full-scale refactoring is very costly). For example, some hardware needs to be initialized in a fixed order or to be accessed with a specialized format/protocol, etc.

Previously, we researched a method for extracting variable structures from legacy code [2]. In this paper, we propose a method for efficient extensibility improvements in embedded software evolution. Our approach consists of the following two steps: (1) to analyze the call graph and inheritance structure of source code and identify extension structures, and (2) to identify the extensibility reinforcement points and suggest a method to improve them. We applied the proposed method to an actual product's code and evaluated the proposed method.

This paper is organized as follows: Section 2 explains the problems in embedded system software targeted in our scope. Section 3 proposes the method for Efficient Extensibility Improvements. Section 4 describes the experiment using actual code. Section 5 discusses the results in this study. Section 6 provides the related works. Finally, Section 7 presents conclusions and directions for future work.

2. Problems in Embedded System

2.1. Scope

Embedded systems are being developed and used widely. This paper targets fresh embedded systems: products for which hardware specifications have not matured, new areas that have unknown market needs, and systems that have customizable characteristics that should be adapted by variation and improvement. These systems include wearable computers, humanoid robots, networking home electronic facilities that are always connected to internet, cooperation systems in homes or offices, automotives with new safety functions, and so on. These new embedded systems do not yet have closed specifications in the system inside.

Our target embedded system structure is like that shown in Fig. 1 illustrates a conceptual layered structure model for an embedded system. The system has at least one *controller class* that controls the features sequence [2]. Each class called by the controller class represents a feature tree. The features consist calling functions, implementing functions, abstracting hardware, algorithms, and so on. Data is transmitted from the hardware to the algorithm class via the controller class, after processing data is transmitted back to hardware for output.

The classes that can be reached on the same number of call times from the controller class are defined as the same layer. The layer nearest to the controller class is called the first layer. The further a layer is from the controller class, the higher its number.

Our targeted embedded system is implemented in object oriented language.

2.2. Problem

In our target embedded system, because of the swiftly changing market needs, it is difficult to use a sequential development process like Waterfall that ensures the detailed specification and allows development without reworking. Therefore, lightweight development process like Agile are easy to apply without implementing additional mechanisms. These processes achieve products that have shorter time-to-market while adapting them to changing market needs. However, in such processes, the quality or maintainability of the system depends on the individual programmers dealing with the code of low

abstraction. In this paper, the problems of the following two dependencies on the programmer are targeted.

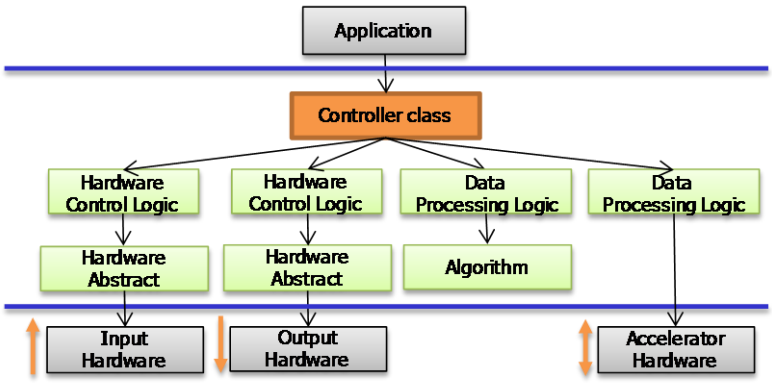


Fig. 1. Basic layer structure in embedded system.

2.2.1. Problem 1: The difference in recognition regarding low extension points of a system

Currently, in the case of a system having new features added, whether a programmer recognizes if the points related to new features have low extensibility or not depends on his/her experience. Thus, extensibility that is too redundant is often implemented. On the other hand, even when there are more points that have a higher frequency of changing, these points are sometimes implemented without extensibility.

2.2.2. Problem 2: The difference in methods to strengthen the low extension points of a system

Currently, the method to strengthen the extensibility depends on the programmer's skill because the general refactoring method explains how to improve the description of the code, not the design. Thus, it is difficult to know whether the extensibility is strengthened or not by the selected method. Also, we cannot know how much the cost will be.

3. Method for Extensibility Improvements

3.1. Overview

As shown in Fig. 2, we propose a method that can find points that need to have their extensibility improved and explain how to improve the extensibility. In the case of identifying a point that needs its extensibility improved in features assigned to an individual programmer, the programmer can improve extensibility in accordance with navigation.

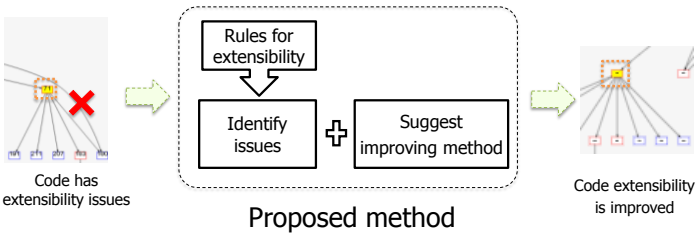


Fig. 2. Proposed overview.

3.2. Architecture of Proposed Method

As shown in Fig. 3, this method consists of two steps. First, we analyze source code from two points of view (call graph and inheritance, which are important mechanisms for object-oriented language) and identify design patterns and inheritance structures in each model by using our previous research method [2] to understand the extension mechanisms quantitatively. A source code is the only input information due to the dependency on the programmer being eliminated in this method (Step 1). Next, we compare rules defining the structure of the extension problem and the extension structures identified in Step 1. Then we identify the points of the extension problem automatically and output the guidelines for strengthening their extensibility corresponding to each rule (Step 2).

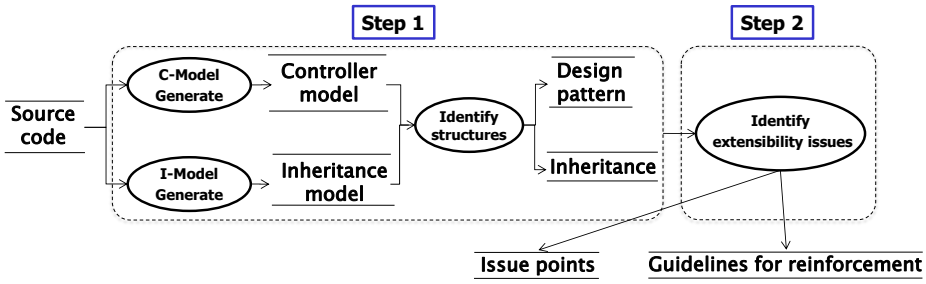


Fig. 3. Architecture of proposed method.

3.3. Identification of the Extension Structures (Step 1)

3.3.1. Extraction of features

The extensibility should be judged for each extracted feature because each feature has different requirements for changing and frequency of changing. For example, the hardware device depends on the product planning of each sensor. An algorithm also depends on requirements for changing received from the market. In particular, an algorithm valuable to a product might increasingly be used. Therefore, features need to be able to change flexibly. On the other hand, a feature can exist for which changes have not been forecasted and might not have flexibility, but it is designed for performance (often the case in embedded systems). Therefore, appropriate structures should not depend on a programmer's experience or skill but should be considered on the basis of long term maintenance cost and requirements.

First, we create a *Controller Model* and an *Inheritance Model* to extract features [2]. Then we identify each class called by the controller class and each feature in a Controller model. A feature consists of various classes that are included in the same inheritance tree in an inheritance model.

Table 1. Extension Structures

Layer / Feature	1st Layer	2nd Layer	Under 3rd layer
Feature A	–	SI	SI
Feature B	MI	SI	MI
Feature C	–	–	–

(a) Inheritance.

Layer / Feature	1st Layer	2nd Layer	Under 3rd layer
Feature A	–	–	–
Feature B	CU	AF	ITF
Feature C	–	–	–

(b) Design pattern.

3.3.2. Classification of extension structures in terms of inheritance

We extract inheritance structures and classify them into two types automatically using the controller model. Table 1 (a) shows an example of classification. *SI* means *simple inheritance structure*, and *MI* means *a different layer inheritance structure* [2].

3.3.3. Classification of extension structures in terms of design pattern

We extract five design patterns automatically by using the controller model. Table 1 (b) shows an example of classification. *CU* means *Mixed Creation and Use pattern, F* means *Factory pattern*, *AF* means *Abstract Factory pattern*, and *P* means *Plugin Factory pattern* [2]. *ITF* means *Inverse Template Method pattern*, which we define in 3.4.4.

3.4. Identifying the Points of Extension Problem and Outputting Guidelines (Step 2)

Table 2 defines the rules of improving the extensibility. Also, we can improve the extensibility in accordance with the guidelines in Table 3 because they are related to the rules for improving the extensibility.

Table 2. Rules for Improving Extensibility

ID	Rules for extensibility	Corresponding guide ID
R1	Encapsulate class creation from user	G1
R2	Do not access encapsulated class	G2/ G3
R3	Encapsulate class creation related to the same devices	G4
R4	Remove inverse template method patterns	G5/ G6
R5	Implement clarified interface	G7

Table 3. Guidelines for Improving Extensibility

ID	Guidelines of method for reinforcement
G1	Insert Factory layer
G2	Change to abstract interfaces
G3	Move functions of upper layer to lower layer
G4	Collect using points and introduce Abstract Factory
G5	Introduce Template Method
G6	Change inheritance to delegation
G7	Insert abstract interfaces

The following describes how to use these rules and guidelines.

3.4.1. Encapsulate class creation from user (R1)

As shown in Fig. 4 (a), classes A and B should not be created by the user who uses them. By avoiding this, the programmer can add new classes without changing the user class.

R1 can be specified when an extension structure in terms of inheritance is detected on the first layer in Table 1 (a). R1 can also be specified when an extension structure in terms of inheritance is located under a layer where an extension structure in terms of design pattern is identified as CU in Table 1 (b).

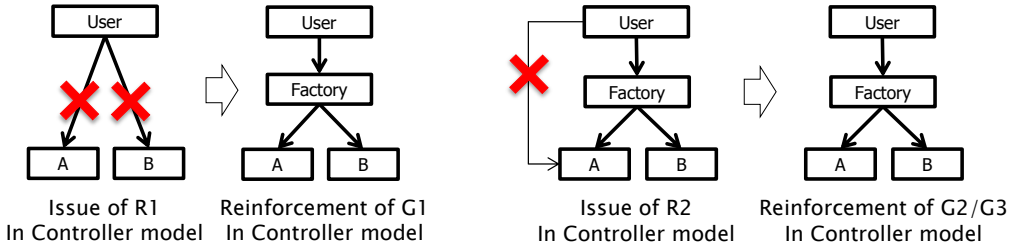
As a guideline for improving the extensibility, *insert a factory layer (G1)* is effective in both above cases. G1 suggests that a programmer should create a factory class that can create an abstract class such as A or B. Then the user class should call it.

3.4.2. Do not access encapsulated class (R2)

As shown in Fig. 4 (b), a user class located in the upper layer of a factory layer should not access concrete classes such as class A or class B. By avoiding this, the system can regain a lost encapsulation effect.

R2 can be specified when a user class located in the upper layer calls an extension structure in terms of

inheritance located under a layer where an extension structure in terms of design pattern is identified in any pattern in Table 1 (b). As guidelines of improving the extensibility, *change an abstract interface (G2)* and *move functions of class located in the upper layer to a concrete class located in the lower layer (G3)* are effective. G2 and G3 suggest that the programmer should relocate functions to appropriate classes.



(a) R1. (b) R2.

Fig. 4. Example of R1 and R2.

3.4.3. Encapsulate class creation related to the same devices (R3)

Fig. 5 shows a model in which classes depend on whether the system has a device A or a device B and changes device. Classes depending on a device should not be dispersed in a system because the change cost will become higher when a specification is changed for a lot of classes depending on a device.

R3 can be specified when classes with similar names exist in different inheritance trees.

As guidelines for improving the extensibility, *collect points that used it and add a factory class (G4)* is effective. G4 suggests that a programmer should create a factory class and an abstract factory class and relocate functions that depend on a device into appropriate classes.

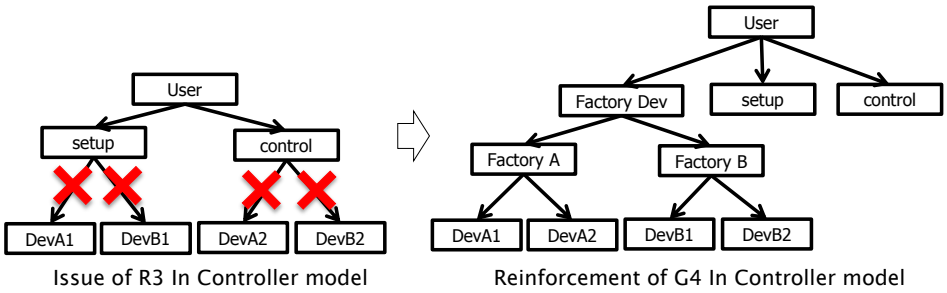


Fig. 5. Example of R3.

3.4.4. Remove inverse template method patterns (R4)

Generally, a super class can replace an inheritance class for an inheritance structure. Then the template method pattern is used. As shown in Fig. 6 (a), this inheritance class A should not call a super class. By avoiding this, the system can be prevented from behaving unexpectedly even if a super class is changed.

R4 can be specified when class A is inherited from super class A in the inheritance model and class A calls a super class A in the controller model. This case is called an Inverse Template Method (ITF) pattern in this paper. As guidelines for improving the extensibility, *change to the template method pattern (G5)* and *change the relationship from inheritance to delegation (G6)* are effective. G5 suggests that the programmer should add virtual functions in a super class and modify these functions in a concrete class. Also, G6 suggests that the programmer should use delegation instead of inheritance.

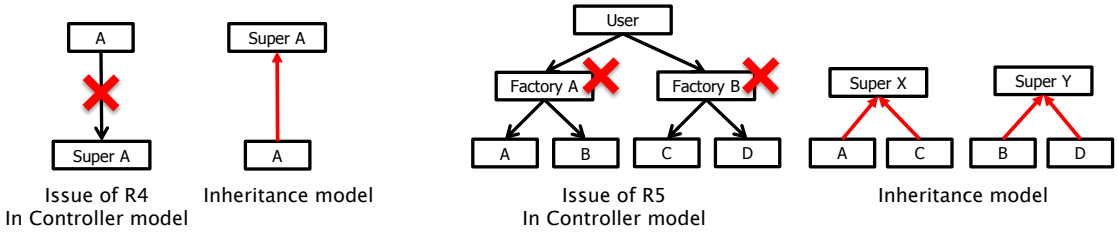
3.4.5. Implement clarified interface (R5)

As shown in Fig. 6 (b), similar classes such as factory A and factory B should be inherited. By doing this,

duplicate functions can be removed and the system can have explicit interfaces.

R5 can be specified when no extension structure in terms of inheritance is located under a layer where an extension structure in terms of design pattern is identified as CU in Table 1 (b).

As a guideline for improving the extensibility, *add an abstract interface (G7)* is effective. G7 suggests that a programmer should create an abstract class with common methods and add an Abstract Factory pattern.



(a) R4. (b) R5.

Fig. 6. Example of R4 and R5

3.5. Analyzing Tool

We created a tool to analyze part of the proposed process. It can create a controller model diagram and an inheritance model diagram.

4. Experiments and Result

We conducted experiments using one of the products of Canon Inc. The product is written in C++ language and is developed for a new market area. Part of this code was evaluated by the proposed method.

4.1. The Outline of the Object Code

Table 4 shows a list of object code for our experiment. ID1 to ID3 are consecutive developments for a same product. On the other hand, ID4 had its extensibility improved using our proposed method.

Fig. 7 shows the code size (NCSS), the number of the classes included in each ID, and the number of the classes in the controller model by our proposed method. As features increase, NCSS and the number of classes increase. The proposed method can decrease the number of the classes to display under 50% of all classes.

Table 4. List of Object Code

ID	Target source code status
1	Initial product code from starting experiment
2	Code added to some features in ID1
3	Code added to some features in ID2
4	Code was improved more than ID3 by proposed method

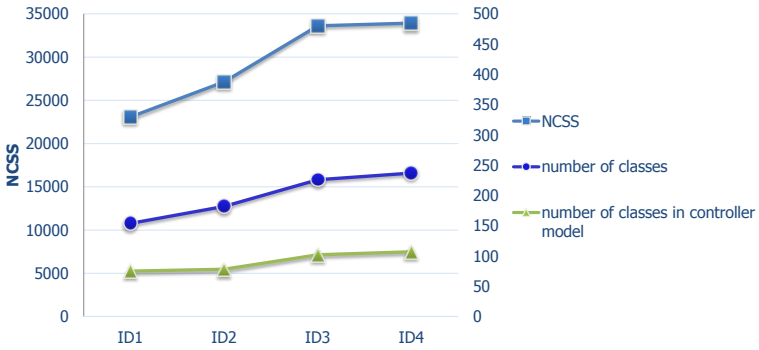
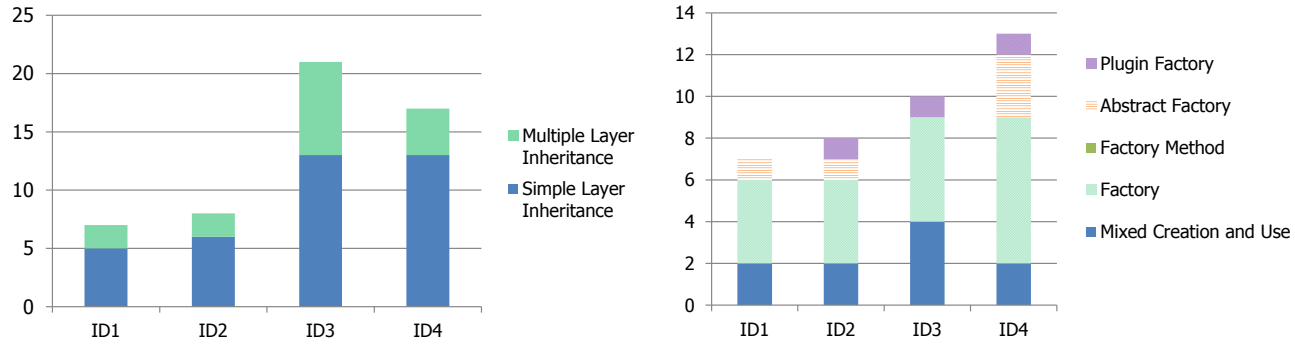


Fig. 7. Progress of code size and number of classes.

4.2. Identification of the Extension Structure (Step 1)

Fig. 8 shows the results of identification of the extension structures for each ID. These are suddenly increased in ID3 regarding inheritance (a). The Mixed Creation and Use pattern is suddenly increased in ID3 regarding design pattern (b). However, in ID4, the Mixed Creation and Use pattern is decreased and Factory pattern and Abstract Factory pattern are increased by improving the extensibility.



(a) Inheritance (b) Design pattern

Fig. 8. Change of the extension structures.

4.3. Identification of the Extensibility Reinforcement Point and Reinforcement Guidelines (Step 2)

Fig. 9 shows changes in the results when extensibility reinforcement points are identified on the basis of five rules in Table 2. R1 (Encapsulate class creation from user) and R2 (Do not access encapsulated class) are simply increased from ID1 to ID3. This data indicates that it is easy to degrade the extensibility by adding features in R1 and R2. On the other hand, R3 (Encapsulate class creation related to the same devices) and R4 (Remove inverse template method patterns) are identified in only ID3. This means that it is easy to degrade the extensibility depending on requirements for ID3 and ID4. Reinforcement points are decreased in ID4 due to the extensibility being improved.

Table 5 shows the difference between ID3 and ID4. In ID4, some classes were added and other classes from ID3 were modified. These were conducted by using the guidelines of extensibility with corresponding rules of the reinforcement points.

R1 and R3 have more added classes than modified classes because construction of new encapsulation structures is mainly a strategy for R1 and R3. For other rules, there are more modified classes than added classes. We improved the extensibility about 18 points (the sum of ID3's reinforcement points minus the remaining points in ID4). About half of all classes in the controller model were changed to improve the 18 points.

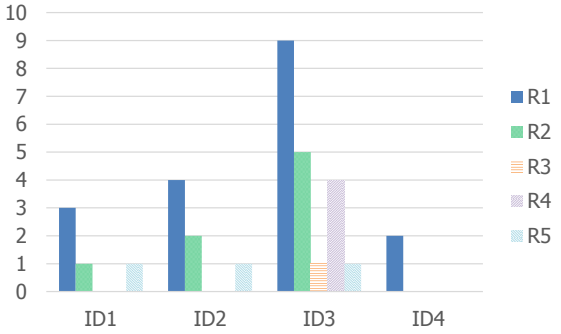


Fig. 9. Changes in results when extensibility reinforcement points are identified.

Fig. 10 shows models before improving the extensibility (ID3), while Fig. 11 shows models after improving the extensibility (ID4). These models are a controller model and an inheritance model [2].

The following subsections (4.3.1 to 4.3.5) show actual examples in ID3 and ID4 regarding the identification of the extensibility reinforcement points and reinforcement guidelines.

Table 5. Number of Additional Classes and Modification Classes

ID		Guidelines of method for reinforcement	Add classes	Modify classes
R1	G1	Insert Factory layer	6	1
R2	G2	Change to abstract interfaces	0	12
	G3	Move functions of upper layer to lower layer		
R3	G4	Collect using points and introduce Abstract Factory	4	2
R4	G5	Introduce Template Method	0	14
R5	G7	Change inheritance to delegation	1	4
amount			11	33

4.3.1. Encapsulate class creation from user (R1)

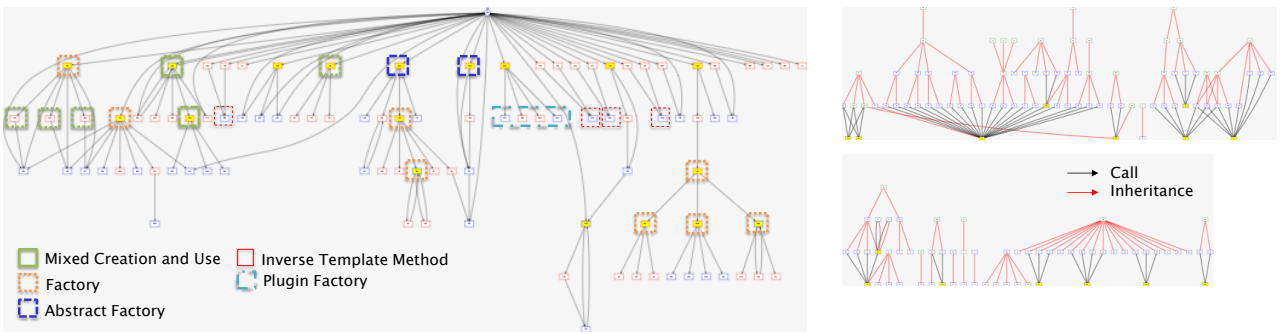
As shown in Fig. 9, in ID3, nine extensibility reinforcement points are identified. We improved them by using the extensibility guideline G1. For three points, we created a factory class for each. For four other points, we performed an Abstract Factory pattern by creating an abstract factory class and two concrete classes inherited from it because concrete classes have a combination. The two remaining points were false positive cases like those described below.

Fig. 12 shows the difference between ID3 and ID4 for the inheritance model. In ID3, a controller class is relevant to a lot of inheritance trees. By reinforcing the extensibility in ID4, relations are divided as shown in the squares with broken lines. Therefore, a controller class is not a concentration of relations any more.

4.3.2. Do not access encapsulated class (R2)

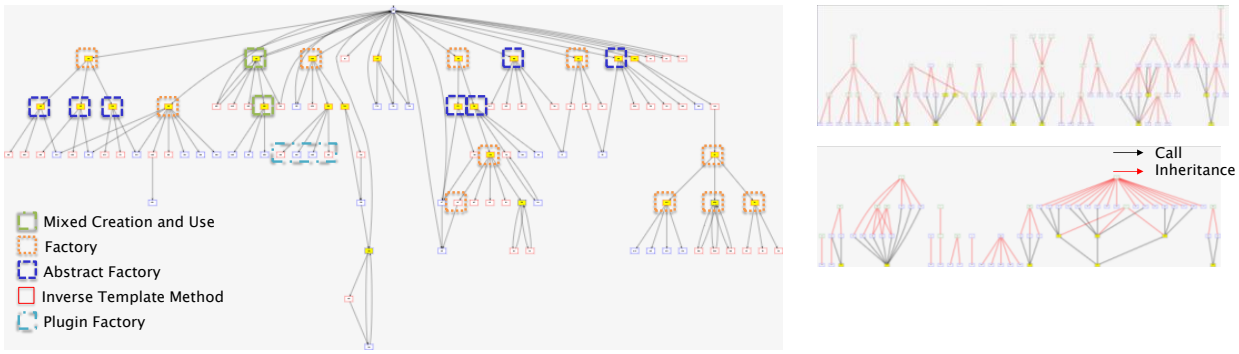
As shown in Fig. 9, in ID3, five extensibility reinforcement points are identified. We improved them by using the extensibility guideline G2. First, we changed the dependency of the upper class from using concrete classes to using abstract classes. Next, we abstracted plural member functions of concrete classes to call by using an abstractive interface. Furthermore, we moved functions defined at upper classes to abstract classes or concrete classes by using the extensibility guideline G3.

Fig. 13 shows the difference between ID3 and ID4 for the controller model. In ID3, there are five calls in which the upper class uses a concrete class. These are described with "X" and should be encapsulated. By reinforcing the extensibility in ID4, we could remove these calls.



(a) Controller model (b) Inheritance model

Fig. 10. Models for ID3.



(a) Controller model (b) Inheritance model

Fig. 11. Models for ID4.

4.3.3. Encapsulate class creation related to the same devices (R3)

As shown in Fig. 9, in ID3, one extensibility reinforcement point is identified. We improved it by using the extensibility guideline G4. First, we gathered the class creation process related to the same device dispersed in a system in one place.

Fig. 14 shows the difference between ID3 and ID4 for the controller model. In ID3, there are three dispersed processes, which are circled. These are classes that can be changed depending on device type. By reinforcing the extensibility in ID4, these classes are gathered in one place. Also, the upper class is changed from Mixed Creation and Use pattern to Abstract Factory pattern, which is circled with a broken line.

4.3.4. Remove inverse template method patterns (R4)

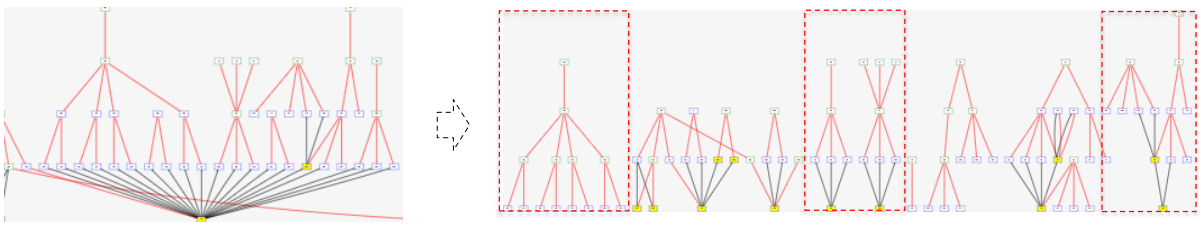
As shown in Fig. 9, in ID3, four extensibility reinforcement points are identified. We improved them by using the extensibility guideline G5. First, we added pure virtual functions into classes identified as the Inverse Template Method pattern. Next, we changed it so that member functions of super classes called by a derived class call these pure virtual functions. Furthermore, we changed member functions of derived classes to overwrite the definition of virtual functions instead of calling the function of super classes. This means that we changed the Inverse Template Method pattern to the *Template Method pattern*.

Fig. 15 shows the difference between ID3 and ID4 for the controller model. In ID3, there are four Inverse Template Method patterns in red squares. By reinforcing the extensibility, these classes disappear in ID4.

4.3.5. Implement clarified interface (R5)

As shown in Fig. 9, in ID3, one extensibility reinforcement points is identified. We improved it by using the extensibility guideline G7. We extracted a common interface from three existing factory classes and created an abstract factory class. Furthermore, we performed an Abstract Factory pattern. It is made by changing existing factory classes to classes inherited from abstract factory classes.

Fig. 16 shows the difference between ID3 and ID4 for the inheritance model. In ID3, three classes call classes inherited by the same super class. By reinforcing the extensibility in ID4, these classes in ID3 are inherited by the same super class and called from another class.



ID3 ID4

Fig. 12. Change of inheritance model by rule R1.

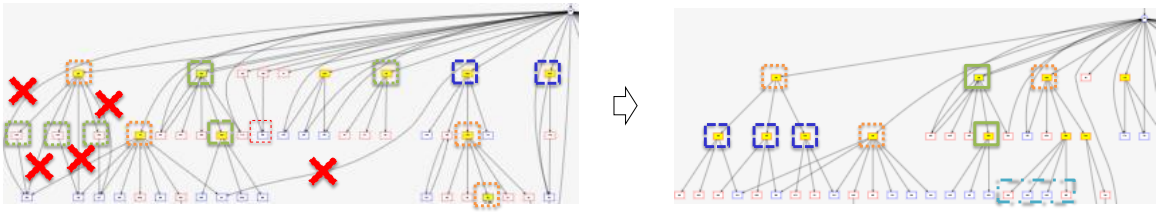


Fig. 13. Change of controller model by rule R2.

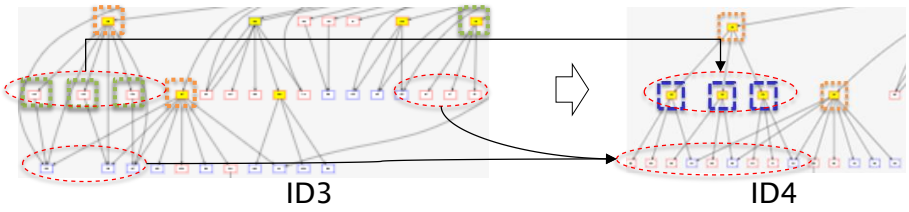


Fig. 14. Change of controller model by the rule of R3.

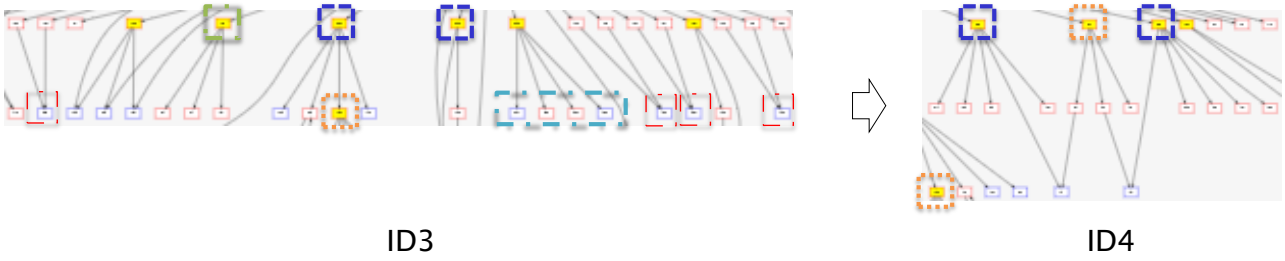


Fig. 15. Change of controller model by the rule of R4.

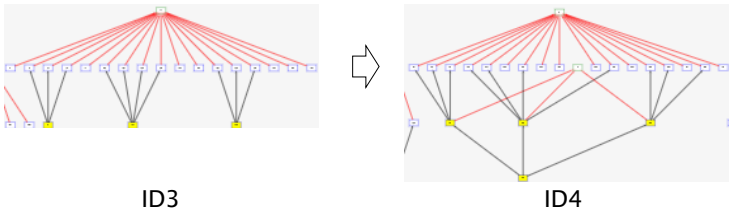


Fig. 16. Change of inheritance model by rule R5.

5. Evaluation

We evaluated the proposed method.

5.1. The Ability of Resolving the Problems

According to subsection 4.3, problem 1 can be resolved by the rule of the extensibility reinforcement that identifies issues of extensibility automatically and quantitatively. Therefore, users of the proposed method can recognize low extensibility points without depending on the experience of the programmer. However, our method outputted two false-positive results. These two cases were points where each class instance was necessary, not points needing encapsulation for an upper layer. It is difficult to judge whether a variation point needs encapsulation or not in the case of the proposed method because it is based on the call relationship of classes and inheritance structures.

Also according to subsection 4.3, problem 2 can be resolved by using the guidelines for the extensibility reinforcement except in false positive cases. Therefore, users of the proposed method can introduce the same mechanism against the same issue without depending on the skills of the programmer.

5.2. Restriction of Proposed Method and Expansion of Application Area

In this paper, this proposed method is limited to the embedded system in a specific condition. We need to study whether it can be applied to any other area's products that already have codes.

In this paper, we deal with object-oriented language [3], which has features like class inheritance, delegation, and others to capture the designer's intent. However, there are many projects that use non object-oriented code in embedded systems. Therefore, we need to study how this proposed method can deal with non-object-oriented code.

6. Related Work

Walkinshaw *et al*. [4] proposed an approach to object-oriented feature extraction using a landmark method and barrier method. This approach extracts features or use cases by slicing the call graph. They removed unnecessary call graphs to avoid explosions. In this paper, features are extracted by using the call graph and inheritance graph, not by a slice-based approach. Also, unnecessary classes are reduced in terms of variability mechanisms.

Keepence and Mannion [5] defined three patterns of class structure design. These patterns are very similar to the mandatory, alternative, and optional feature properties in the feature oriented domain analysis. In this paper, five patterns are identified in terms of encapsulation of changing. These patterns correspond to the degree of flexibility.

Makkar *et al*. [6] said there is a relationship between depth of the inheritance tree and reusability. As the depth of the inheritance tree increases, reusability decreases. They also said the threshold is within three layers. Therefore, they created an equation relating the two quantities. In this paper, we compute the inheritance tree containing a variability mechanism, which is placed under two layers.

Bansiya and Davis [6] mapped object-oriented design components to design metrics and design quality attributes. Hudli *et al*. [8] validated various object-oriented metrics. For object-oriented language, various evaluation methods have been proposed in terms of design or metrics. This paper described the method for evaluating design in legacy code in terms of extensibility.

Babar [9] and Bengtsson *et al*. [10] proposed methods for evaluating software product line architecture. As they described, a method based on a scenario such as SAAM and ATAM has been established. In this paper, we propose a method for evaluation by analyzing legacy code.

Kaur and Singh [11] compared the maintainability index calculating the package size and metrics like complexity of package. The summarized value cannot obtain concrete implemented structures such as extension structures like in this paper.

Munro [12] modeled the bad smell of code and identified it by using code metrics. He used lines of code, complexity, and so on. His method improves code when metrics exceed references. In this paper, issues of extensibility were identified and improvements suggested by evaluating extensions by using code metrics and extension structures.

7. Conclusion and Future Work

This paper has proposed a method for Efficient Extensibility Improvements in Embedded Software Evolution. This method is focused on extension structures such as inheritance and design patterns in legacy software. We applied the proposed method to an actual product's code, which improved in some times. The method could detect extensibility reinforcement points automatically and improve the extensibility of the system. Furthermore, we could verify the efficiency of the proposed method. This proposed method enables us to improve the extensibility efficiently and consecutively regardless of the programmer's experiences or skills, especially in an Agile development process.

In the future, we intend to compare the flexibility to change between different embedded systems and add to the extensibility rules.

Acknowledgments

This work was supported by University of Electro-Communications and National Institute of Informatics. Canon Incorporation provided the software and computer resources in this work.

References

[1] Ghanam, Y., Andreychuk, D., & Maurer, F. (2010). Reactive variability management using agile software development. *Proceedings of the International Conference on Agile Methods in Software Development* (pp. 27-34).

[2] Sasaki, T., Yoshioka, N., Tahara, Y., & Ohsuga, A. (2014). Evaluation of flexibility to changes focusing on the variable structures in legacy software. *Knowledge-Based Software Engineering*. Springer International Publishing.

[3] Breesam, K. M. (2007). Metrics for object-oriented design focusing on class inheritance metrics. *Proceedings of the 2nd International Conference on Dependability of Computer Systems* (pp. 231-237).

[4] Walkinshaw, N., Roper, M., & Wood, M. (2007). Feature location and extraction using landmarks and barriers. *Proceedings of the International Conference on Software Maintenance* (pp. 54-63).

[5] Keepence, B., & Mannion, M. (1999). Using patterns to model variability in product families. *IEEE Software*, *4*, 102-108.

[6] Makkar, G., Chhabra, J. K., & Challa, R.K. (2012). Object oriented inheritance metric-reusability perspective. *Proceedings of the International Conference on Computing, Electronics and Electrical Technologies* (pp. 852-859).

[7] Bansiya, J., & Davis, C. G. (2002). A hierarchical model for object-oriented design quality assessment. *Software Engineering*, *28(1)*, 4-17.

[8] Hudli, R. V., Hoskins, C. L., & Hudli, A. V. (1994). Software metrics for object-oriented designs. *Proceedings of the IEEE International Conference on Computer Design* (pp. 492-495).

[9] Babar, M. A. (2007) Evaluating product line architectures-methods and techniques. *Proceedings of the 14th Asia-Pacific Software Engineering Conference*.

[10] Bengtsson, P., Lassing, N., Bosch, J., & Van, V. H. (2000). Analyzing software architectures for modifiability.

[11] Kaur, K., & Singh, H. (2011). Determination of maintainability Index for object oriented systems. *ACM SIGSOFT Software Engineering Notes*, *36(2)*, 1-6.

[12] Munro, M. J. (2005). Product metrics for automatic identification of "bad smell" design problems in java source-code. *Proceedings of the 11th IEEE International Symposium on Software Metrics* (pp. 15-15).

Takanor Sasaki is a Ph.D. candidate at the University of Electro-Communications, Tokyo. He received his B.E. and M.E. degrees in mechanical system engineering from Hiroshima University in 1997 and in 1999, respectively.

From 1999 to 2008, he was with Mitsubishi Precision Co., Ltd., Japan. Since May 2008, he has been with canon incorporation, Tokyo, Japan. His research interests include software engineering, cloud computing, formal verification of software, and software evolution. He is a member of the Information Processing Society of Japan (IPSJ).

Nobukazu Yoshioka is a researcher at the National Institute of Informatics, Japan. Dr. Nobukazu Yoshioka received his B.E degree in electronic and information engineering from Toyama University in 1993. He received his M.E. and Ph.D. degrees in School of information science from Japan Advanced Institute of Science and Technology in 1995 and 1998, respectively.

From 1998 to 2002, he was with Toshiba Corporation, Japan. From 2002 to 2004 he was a researcher, and since August 2004, he has been an associate professor, in National Institute of Informatics, Japan. His research interests include security and privacy software engineering, cloud computing, agent technology, software engineering, and software evolution.

He is a member of the Information Processing Society of Japan (IPSJ), the Institute of Electronics, information and Communication Engineers (IEICE) and Japan Society for Software Science and Technology (JSSST), the Japanese Society for Artificial Intelligence (JSAI) and IEEE CS. He has been a chair of IEEE CS Japan Chapter since 2015.

Yasuyuki Tahara is an associate professor in the University of Electro-Communications. He received his BSc and his MSc in mathematics from the University of Tokyo, Japan, and his PhD in information and computer science from Waseda University, Japan, in 1989, 1991, and 2003, respectively.

He joined Toshiba Corporation in 1991. He was a visiting researcher in City University London, UK, from 1995 to 1996, and in Imperial College London, UK, from 1996 to 1997. He left Toshiba Corporation and joined NII in 2003. He left NII and joined the University of Electro-Communications in 2008. His research interests include formal verification of software and requirements engineering. He has been working for an education program called Top SE since 2004.

Prof. Tahara is a member of the Information Processing Society of Japan and Japan Society for Software Science and Technology.

Akihiko Ohsuga received a B.S. degree in mathematics from Sophia University in 1981 and a Ph.D. degree in computer science from Waseda University in 1995.

From 1981 to 2007 he worked with Toshiba Corporation. Since April 2007, he has been a professor in the Graduate School of Information Systems, the University of Electro-Communications (UEC). Since April 2012, he has been also a visiting professor in National Institute of Informatics (NII). He has published more than 210 papers (100 papers at journals and 110 papers at international conferences). His research interests include agent technologies, web intelligence, and software engineering.

He is a member of the IEEE Computer Society, the Information Processing Society of Japan (IPSJ), the Institute of Electronics, Information and Communication Engineers (IEICE), The Japanese Society for Artificial Intelligence (JSAI), and Japan Society for Software Science and Technology (JSSST). He received the 1986 Paper Award from the Information Processing Society of Japan.