ISSN 2409-2665
Journal of Logistics, Informatics and Service Science
Vol. 9 (2022) No.1, pp.97-115
DOI:10.33168/LISS.2022.0108

A Case Study of Domain Engineering in Software Product Line Engineering

Jeong Ah Kim

Catholoc Kwandong University, GangNeung, South Korea

clara@cku.ac.kr

Abstract. The software product line is a software development paradigm that arose from an effort to maximize the quality attributes of software such as reusability and maintainability. In software product line engineering, there are several methodologies for improving both the productivity and quality of software. But, engineers and organizations still have difficulties in adopting the software product line engineering. In this paper, detail guidelines for performing the activities of domain engineering in embedded software domain.

Keywords: Software product line, domain engineering, reusability, platform

1. Introduction

The rapid expansion of the software domain requires the fast development of high-quality software as it becomes more in demand to human needs and daily activities. However, it requires more careful engineering with a lot of time and effort. Thus, there is an inevitable dilemma between product quality and productivity. In order to solve this, a scheme that considers software products with similar functions together in development is required, rather than developing each software product individually. In other words, it is a method for developing the desired system by considering functions and quality attributes that must be achieved by multiple software belonging to the same domain, ensuring core assets through commonality and variability analysis, and combining these assets (K Pohl et.al 2005). The development technology applying this concept is called software product line engineering (SPLE). SPLE provides methods and tools to systematically reuse software assets and establish software product lines—portfolios of software products (a.k.a., variants) in an application domain (Robert Lindohf et. al 2021). It consists of two processes (K Pohl et.al 2005, Krüger J et.al 2020): Domain engineering for constructing the software platform and application engineering for making new software product from the platform. In this paper, variability implementation mechanism for application engineering was suggested.

2. Various methodologies of software product line engineering

In software product line engineering, there are several methodologies for improving both the productivity and quality of software. A Feature-Oriented Reuse Method (FORM) constructs a feature model based on decision-making from the initial marketing and product planning stages of a software product line, and develops software in a proactive manner using the feature model (Kyo C. Kang et.al 1998). Based on the analysis results of the previously developed system through reverse engineering, a feature model is constructed, and based on this, an engineering technique is provided to develop software in an extractive manner.

PuLSE (Product Line Software Engineering) is an easy to apply method of extracting the assets constituting the software product line from existing products through re-engineering (Joachim Bayer et.al 2008). Algebraic Hierarchical Equations for Application Design(AHEAD) mainly provides a reactive method that gradually expands the software product line through iterative refinement. GP(Generative Programming) method provides a method to generate program code based on the detailed feature model (Mikoláš Janota et.al 2008).

Currently FORM, PuLSE and AHEAD are the most widely used methodologies in the world. Among various techniques used to analyze commonalities and variabilities in software product line engineering, the feature model is used as a de-facto standard.

The world's leading companies, such as Motorola, Ford, Bosch, Toshiba, Hitachi, Ericsson, Philips, Hewlett-Packard (HP), AT&T, Lucent, ALLTEL, Boeing, and Lockheed Martine have already achieved significant cost savings in software development by adopting software product line engineering. Nevertheless, it has not been sufficiently adopted by South Korean companies, and only large companies are trying to apply it on a trial basis. Gradually, it is expanding centering on automotive electrical system software.

As a community related to software product line engineering and reuse, the Software Product Line Conference (SPLC), International Conference on Software Reuse (ICSR), and Variability Modeling of Software-intensive Systems (VaMos) are active. Unfortunately, there is no translated and published materials related to software product line engineering in South Korea. However, there is a “Guide to Applying Software Product Line Engineering Technology Based on Software Reuse” published by the POSTECH Convergence Software Development Center with the support of NIPA. The Hall of Fame by SEI presented successful cases of adopting software product line engineering every year.

Standardization for tools applied to software product line engineering has also been in progress, and the first standard was published as ISO/IEC 26550 in 2012. Among them, ISO/IEC 26551 and ISO/IEC 26555 are led by South Korea. The cases of R&D investment in the United States and the European Union show that software product line engineering technology has become an irreplaceable means of strengthening national competitiveness in today's software industry.

3. Research results: Adoption planning for SPL

In order to make a decision to adopt SPL technology, risk analysis, product group evaluation, and approach determination are carried out. These three tasks are performed sequentially or in parallel.

3.1. Risk analysis

The activities to identify and analyze risk factors associated with the adoption of technology are conducted as follows. First, identify risk factors in terms of domain maturity, stability, and business characteristics. Then, the identified risk factors are evaluated to determine which problems will pose a threat with what degree of risk in adopting SPLE. Risk analysis must be conducted by a domain expert as well as a domain analysis expert.

The things to consider carefully when evaluating risk factors in terms of domain maturity are as follows:

- Are there generally accessible technical resources and reference cases related to the technology for the domain?

- Does the company have experience in developing products corresponding to the domain? Or are there engineers with experience in developing products corresponding to the domain?

As for the automotive body domain, the subject of this case study, the risk of technology adoption has been evaluated as low considering the maturity level and technologies owned by the company, as well as the fact that the candidate product groups consist of products that will be applied to next-generation automobiles.

3.2. Product group evaluation

The evaluation of candidate product groups to which the product line will be applied is the most important task among activities related to technology adoption (Kalender ME et. al 2013, Rincón L et. al 2019, Berger T et.al 2020). Products to be included in the product line are selected and the common areas of the product line are identified by examining details in terms of marketing, product plan, and functionality for the candidate products discussed in the interview or risk factor analysis. The feature list obtained in the identification process is used to determine the range of the product line for the future.

3.3. Approach determination

The SPL basically provides the following three approaches (K Pohl et.al 2005): Proactive approach, Reactive approach, Extractive approach. In this case study, a mixed approach is taken with an extractive approach based on an extractive approach, supplemented by a reactive approach. The reason for the decision to take the extractive approach is that, like most companies today, this company has a legacy system for the product. As it is difficult to capitalize all modules at once considering the development period and technical application period, it has been decided to proceed by repeatedly capitalizing the modules

4. Research results: Domain engineering

This chapter presents the case of performing domain engineering to develop reusable assets using SPL technology.

4.1. Domain analysis activities

The SPL domain analysis activities consist of term definition, feature modeling, legacy system analysis, and domain requirement specification. Term definition and feature modeling are generally carried out in parallel. Legacy system analysis is an additional activity as an extractive approach is taken in this case study. The domain requirement specification is usually performed at the end of the domain analysis activities.

Term definition is the task of defining the terms used in the domain. The reason for defining the term is that various words with the same meaning may be used

depending on the knowledge, background, and experience of the stakeholders involved in the development. These differences act as a major factor hindering communication between stakeholders.

As a part of the term definition activity, a domain glossary is created. The simplest way to create a glossary is to use a spreadsheet tool such as MS-Excel. However, the spreadsheet is not suitable to meet the original purpose of knowledge sharing and quick correction. Therefore, it is recommended to use the web-based WIKI service. In this case study, the WIKI provided by Redmine is used.

The procedure for creating a glossary using Redmine WIKI is as follows.

- Edit the main page of the WIKI to create a page for the glossary.
- Edit the glossary page to create a session to categorize terms in the English/Korean alphabet.
- Create a list of terms included in the corresponding alphabet session, and create a page corresponding to each term.
- Edit the page corresponding to each term to create a description of each term.

Feature modeling is the most important task among domain analysis activities (FJ Van der Linden et.al 2007, Nešić D et. al 2019). This task aims to clearly analyze the commonalities and variabilities of the products included in the product line (El-Sharkawy S et.al 2019). It is conducted by referring to product specifications, requirements definitions, developer interviews, and legacy system codes (Berger T et.al 2014).

In this case study, 80 features are identified. Three optional features and nine alternative feature groups (20 features) are identified as variable features. The identified commonalities and variabilities are used as criteria for classifying modules in architecture design, which is an engineering task, and are referred to when classifying and processing variable attributes of modules in detailed module design. The feature model created by analyzing the slide controller domain of the company is as follows (some features are omitted). Some examples of feature model composition rules related to feature composition are as follows.

The legacy system analysis aims at analyzing the functions and structural problems of the currently used system. The recognized functions and structures are utilized as improvements when designing the product line architecture (domain architecture) (Steffen Thiel and Andreas Hein, 2002) (MA Laguna and Y Crespo, 2013) (Len Wozniak and Paul Clements, 2015). The development language of the legacy system applied to the analysis is ANSI-C, the source size is 26,000 lines of code (LOC), and the number of original files is about 80. The tool used to analyze the quality assessment metrics in terms of quality is SourceMonitor v3.5. The Kiviat Metrics Graph, analyzing the target legacy system with the SourceMonitor tool, is as shown below.

The overall analysis results for the legacy system based on the assessment are as follows.

Overall, the code complexity remains at an adequate level of 3.48. There are many codes that exceed the recommended level of complexity (2.0 to 4.5). Refactoring of codes with high complexity may be difficult with the source code provided by the provider. Among the files with high complexity, there are more than three functions with a code complexity of 40 or higher. The function with the highest complexity is seven times more complex with 57 points.

The proportion of comments is considerably lower than that of the code. Based on this assessment, the engineering direction is established as follows.

- Software has not been designed with modularization in mind.
- Functionality is concentrated on a specific function, which is likely to cause multiple maintenance issues.
- The architecture should be designed considering modularization from the architecture design stage.
- Engineering should progress in the direction of lowering code complexity by properly assigning functions to each module.

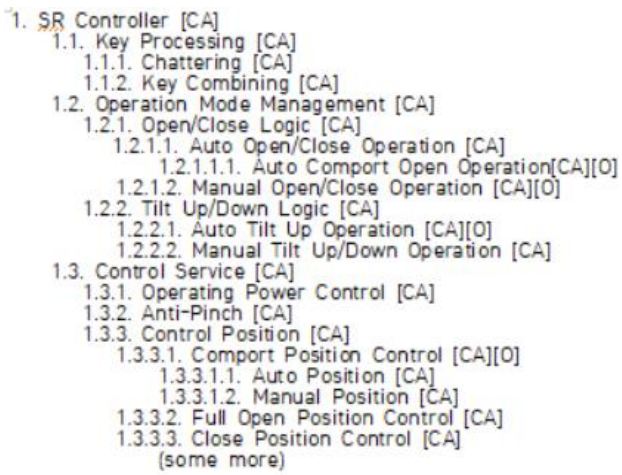

Fig. 1: Parts of the feature model

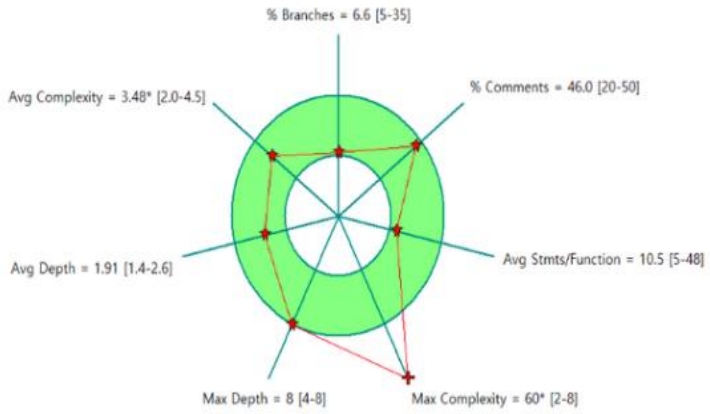

Fig. 2: Evaluation results of legacy code

Domain requirements specification is the same as the task of discovering the requirements, which is first carried out when developing a system. Additionally, discovered requirements are refined by reflecting the variable information identified based on the feature model, and the variable information inherent in the requirements is recognized and reflected back in the feature model. The relationship between the discovered requirements and the feature is a relationship in which specific requirements are satisfied by the feature. If the variable type of the linked feature is optional, the corresponding requirements are also optional requirements. In order to define a requirement as a variant or variation point, a requirement with a high level of abstraction, such as a business requirement or a customer requirement, should be concretized into one or more essential requirements through segmentation. The figure below shows the identified requirements arranged in a layered structure. Among the 10 requirements collected and refined, REQ1.1.1, REQ1.3.2, and REQ1.3.2.1 marked an orange background are satisfied by the features "Auto Tilt Up Operation," "Comport Position Control," and "Auto Comport Open Operation," respectively. Optional requirements are specified by setting these relationships.

The following items are verified by establishing the relationship between the features and the requirements.

- Each feature must be assigned to at least one requirement.
- Each requirement must be specified by at least one feature.

4.2. Domain architecture design

For SPL domain architecture design, architecture requirement analysis, legacy system architecture analysis, architecture design view development, and architecture design verification are performed. Legacy system architecture analysis is an additional activity as an extractive approach is taken in this case study. Since this company has not defined a view necessary for architecture design, views are defined

in the course of the analysis tasks and the architecture design view development task. Furthermore, architecture design view development and architecture design verification tasks are performed repeatedly. The architecture requirement analysis aims at determining the important quality attributes that must be considered in the design at the architectural level.

The determined quality attribute is used as a basis for materializing the architecture in the subsequent architecture design process (JeongAh Kim et.al 2018). The definitions of quality attributes vary, but in this case, they are limited to system quality attributes. The quality attributes are identified, and a scenario is created by referring to the definition of quality attributes from a product point of view among ISO/IEC 25010 quality attributes, with a focus on the quality attributes defined in CMU SEI. As SPL technology that emphasizes reuse is used in this case study, quality attribute scenarios are discovered, and a total of four core quality attributes are identified with quality attributes such as “modifiability” and “maintainability” as a premise.

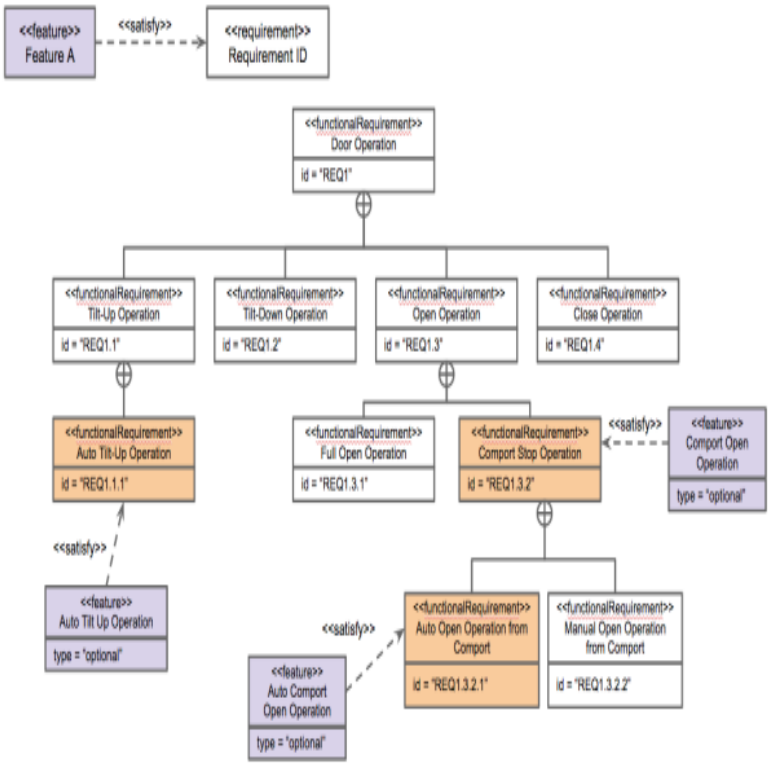

Fig. 3: Mapping between features and associated requirements

Table 1 Quality attributes that need to be achieved in the slide control system

Quality attribute	Content
Availability	Anti-pinch should not be operated by misrecognizing the force caused by vibration from closing the slide, voltage fluctuation, or vehicle operating environment.
Availability	When controlling the position of the slide, the position must not be changed by an external interruption (removing the battery, fast on/off operation, turning off the engine, etc.).
Performance	All functions related to motor control in the controller must be performed within 2ms.
Reliability	As the reference position of the physical-mechanical part is slightly changed by the continuous operation of the slide, the reference position must be continuously updated to prevent the slide from stopping.

Legacy system architecture analysis is performed through source code review. A useful structure that can be referenced for architecture design is found by identifying the software structure implementing the control function, and the direction of architecture design is determined. In the code structure analysis, after selecting files directly related to the application based on the code quality assessment, the codes are reviewed by a person. In this process, it is also possible to use a tool to examine the dependency between files based on the call relationship. When a folder is recognized as a module in terms of the source code structure managed in the file system, the dependency of the modules in the legacy systems and source code review and developer analysis results are summarized as follows.

There are no criteria for the implementation of logical modules. The codes corresponding to the application of legacy systems have similar structures. The reason for the similarity in code structure despite the absence of software architecture design is that the codes have been written by the same developer. Codes related to control logic are developed without the concept of modularization. Although it has the unique code characteristics of embedded systems, modularization at the architectural level is low. In particular, the part related to the control logic is judged to be a monolithic system except for input/output as the module is too large. The architecture corresponding to the control logic is made more specific based on the current codes, and the design direction is determined in the form of proposing an implementation model for a logical module recognized in the architecture. In order to establish a clear architecture, a design view must be defined. Design views can be created naturally by applying the typical software development methodology. However, most small and medium-sized controller development companies tend to have no design views in the absence of any appropriate development methodology defined for the level of

the company. Or, the architecture view is vaguely classified into static architecture and dynamic architecture according to the criteria proposed by the automobile assembly company (OEM), with no definition of what information should be expressed and described in what way according to the characteristics of the domain and organization. As a result, even developers in the same organization end up writing the details of the architecture in different ways in the specification. The following questions are used to determine the level of understanding of the architectural design view.

- Is there a software architecture specification?
- Do you understand the architecture view?
- Can you explain exactly what kind of diagram you use for architecture specification and the relationship between each diagram?
- Can you accurately describe the difference between a general-purpose modeling language like Unified Modeling Language (UML) and domain languages like SysML, AADL, and EAST-ADL.

As neither a specific design methodology nor a definition of the design view is identified for the company in this case study, the architecture design view is defined, and the representation method is determined prior to developing the architectural design view. Figure 4 summarizes the architecture design activities, including the creation and verification of the architecture design view.

The module architecture design aims to clearly define the modules that make up the actual software based on the functional blocks recognized in the conceptual architecture. Blocks defined in the conceptual architecture can be collected and defined as one module. Although not covered in the conceptual architecture, a module that supports the function of the module can also be defined. Since a module is an important unit that leads to actual implementation, modularization must be taken into account. Modular architecture is basically expressed in layered architecture style. A block diagram is used to express how each module interacts with other modules through which interface. In the process of designing a conceptual architecture, all recognized blocks have their own roles. The role is suggested by the name of the block. The layer where the module will be located is designed based on these characteristics of the blocks. Previously, the characteristics of the blocks recognized in conceptual architecture have been classified into six types. Input Interface, Output Interface, Network Communication, and Application/Task are easily found in the reference architectures of embedded software. Status Management and Logical Calculation, Operation Handler, and Fault Handler are unique parts of the slide control software. However, as it is designed under the principle of separating control and operation, Status Management is in charge of managing the status for control,

and Calculation or Handler is in charge of the operation to be performed in a specific state.

Following the principle of the basic layered architectural style in which control is placed in a higher layer and input/output in a lower layer, the layers are designed as follows. There are a total of four layers. The lowest layer is horizontally divided into an input/output interface and a network communication layer. The highest layer is where the modules responsible for the entire life cycle of the application are located. After determining the module based on the blocks recognized in the conceptual architecture in the module architecture layer, the module is assigned to the corresponding layer.

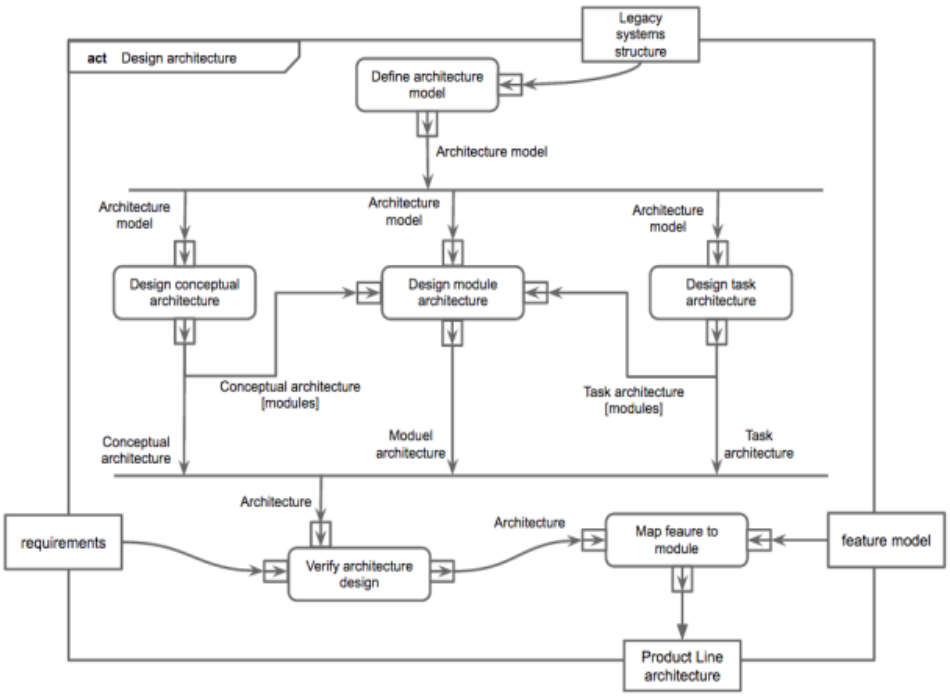

Fig. 4: Activities and work products of architecture design view development

One block can be defined as one module, or several similar blocks can be gathered and defined as one module. When defining similar blocks as one module, it is important to group the blocks included in the conceptual block together. Blocks that can no longer be divided at conceptual architecture level 1 are defined as a single module. Since the module architecture view has the role of showing the developer an overview of the system, it is recommended to indicate not only the internal components of the system being designed, but also the external elements that directly interact with the system. A library or module that is not directly developed is an example of an external element that is typically described in the module architecture view. In order to reduce the strong dependency between the module closely related

to the hardware and the target software, a method of placing a glue layer is generally used. In the module architecture view, the layers that act as a glue layer between the software and hardware layers include the Input Interface Layer, the Output Interface Layer, and the Communication Layer. In other words, while the modules assigned to these layers provide a consistent interface to the software, their logic may have to be modified according to the contents of the device driver provided by the hardware whenever hardware is changed.

The factors directly affecting software development are added to the architecture view. Data objects referenced by the module are taken into account, although they are not a module that directly composes software. Tools used directly in the development of the data object under consideration, as well as the modules that comprise the software, are also considered.

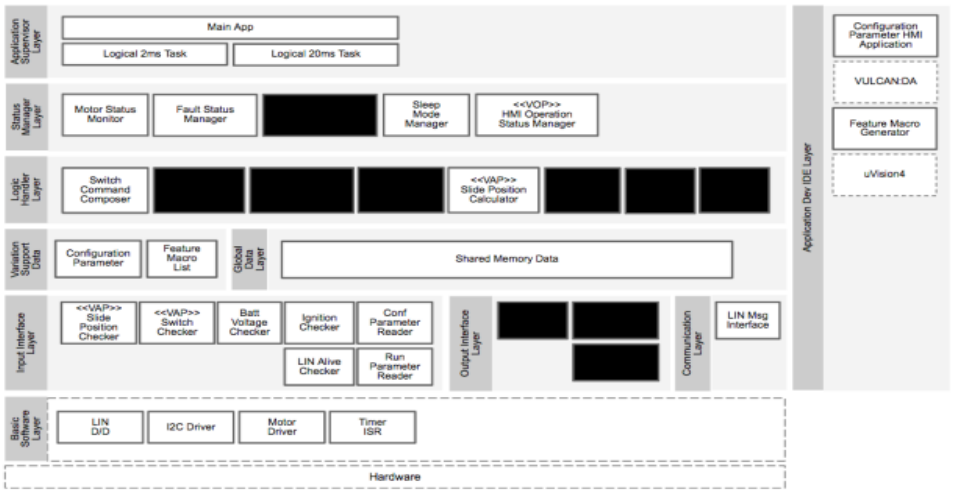

Fig. 5: View of module architecture considering the SW development supporting items

A total of three modules are defined in the data object layer. Configuration Parameter and Feature Macro List are data used to process the variability of software. These data objects are used to handle product variability by creating different instances for each product. The configuration parameter is a data object used directly for control in software, and the value can be set differently for each product. Feature Macro List sets a macro corresponding to the feature list selected from the feature model according to the product. Depending on whether the feature macro is set or not, the architecture structure and the logic of the module implementing the corresponding feature are affected. Shared Memory Data is an object in which data shared between modules is defined. Since most modules that comprise software have a strong dependency on the data structure defined in this object, the data structure is always constant for each product, unlike the two data objects described above. The list of data defined in the shared memory data can be derived from the data defined in the

conceptual architecture. A total of four modules are defined in the Application Dev IDE Layer. Among them, VULCAN:DA and uVision 4 are development support tools, but they are marked with dotted lines as they are not actual subjects of development. The remaining two modules, Configuration Parameter HMI Application and Feature Macro Generator, are supporting tools developed directly to implement the product line.

Configuration Parameter HMI Application is a tool to set the value of the configuration parameter described above for each product. Although this HMI application is not software running on the target hardware platform, it has been added to the product line architecture as an element setting the value of the data object utilized by the target software. This HMI application is an element derived from the process of designing how to reflect the parameter variation points identified in the process of creating the feature model for the product.

Feature Macro Generator is added to the product line architecture as a tool that generates the Feature Macro List that affects the variability of software according to the selection of the feature list. In order to verify that the function is well reflected in the architectural design, the acquired use case scenario can be used. Whether the use case scenario is performed correctly based on the interface designed in the architecture can be examined. In other words, whether the interaction between modules is made according to the scenario is checked. Information on the interface identified based on the scenario created for the use case should be reflected in the module architecture design. Whether the functional features identified during domain analysis are all implemented as modules of the architecture can be examined by establishing the relationship between the features and the modules constituting the architecture. Whether the optional or alternative variable information identified in the feature model is well reflected in the module is examined to make sure that an architecture suitable for the product line has been designed.

4.3. Product line module development

The SPL module development activities consist of module design, module implementation, and module verification. In module design, what design should be considered in order for the module to flexibly absorb the variable information recognized from the features assigned to the module during the architectural design process is explained. In module implementation, the rules for developing the source code based on the designed contents are explained, and the method of implementing the variable information is considered in the design at the source code level. Module verification means the testing of units and modules to evaluate the developed source code. This study focuses on how to reflect variable information in design and its content in product line module design. For effective explanation, variation points and variants identified in the application process are introduced by type. Design examples for data objects identified in the conceptual architecture view are introduced.

4.4. Product line module implementation

Product line module implementation serves two purposes. First, it aims to create concrete implementation models based on the conceptual modules acquired from architecture design. Second, it aims to determine the technique for implementing the variable information reflected in the design. This section describes the implementation rules based on the C language used for automotive electrical system software development.

An object-oriented language such as Java provides various methods to implement the module defined in the architecture as an independent component at the language level. However, as for the C language, there is a limit to implementing the logical module defined in the architecture as an actual independent module at the language level. This is because calling it an actual module makes it impossible to forcibly block the access restrictions of the developed module. As a result, developers make arbitrary use, and architecture and consistency are gradually lost as development progresses. In order to prevent such a problem in advance, it is necessary to define the rules for developing modules in the C language. The things to consider when writing the rules are as follows.

- What is the physical implementation unit of the module? Is it a folder? Is it a file? Is it a function?
- How should the layer of design be implemented in the architecture?
- How should the interface defined in the architecture design be implemented?

As there are more team members involved in software development, it becomes increasingly difficult for architectural design to be consistently implemented into codes in the absence of such rules. If some developers implement the module as a file while other developers implement the module as a function included in the file, module management becomes impossible from then on. In this case study, the following criteria are applied based on the module architecture for implementation.

Table 2: Guideline for structure of module implemetation

Architecture Element	Implementation Technique
Layer	Folder/Package
Module	Module file(.c)
Module Interface	Header file containing the interface declaration
Sub Module	Function

There should be as many C files as the number of modules defined in the module architecture in the package. How to implement the variable elements inside the module described in the product line module design should be determined. The most easily used variable information processing mechanism in the C language is to utilize the macros and the pre-processor provided by the C compiler. A macro is the easiest way for organizations or developers to deal with variable elements for the first time. However, with an increase in the number of features, the source code becomes messy, reducing readability. Accordingly, a more advanced technique should be applied thereafter.

The mechanism applied to implement the variable information applied in this case study is as follows.

Table 3: Impmentation mechanism for variation type

Variation Type	Variable Part	Implementation Mechanism
State of Control	Behavior Model	Macro
Condition of Transition	Priority of Key Command based on Key	Script for data creation
Operating Condition	Value	External HMI program
Parameter	Logic	Function and macro
Algorithm		

In this case study, three implementation mechanisms were mainly used: macro, declaration of data, and external HMI program. A case of application to simplifying detailed design using comments when implementing source code is presented. For electrical system software developers, this may be a strategy that can be used in situations where it is not possible to spend a lot of time documenting the design content. As of now, tools that automatically generate documentation based on source code comments include JavaDoc and Doxygen. Since JavaDoc is dependent on the Java language, Doxygen is used when development is based on the C language. As Doxygen provides various comment formats and commands, it has the advantage of automatically obtaining various types of documents. Doxygen, however, cannot contain all the details of the design, and it should be used for the purpose of assisting the simplification of the design. An example of writing comments on source code using the comment command of Doxygen is shown below. Comments can be added to the beginning of files, functions, and variables.

The command supported by Doxygen starts with @ to write a brief introduction, date of creation, version, rights, etc. Additional information can be included by using

the @note and @par commands. In this case study, additional information such as @par Feature:Fault Priority is added to indicate that the module implements a specific feature. In the comments applied to the function, the @note command can be used to specify whether the function is an internal module function or an interface function, in addition to the basic description of the function. Although these comments are not enforceable, they can be used to make a promise between the developers that the module will be used through the interface by providing such information to the developer. The following function is the interface function provided by the ABC module. In order for another module to interact with this module, this function must be called. When this function is called, the function that internally determines the priorities and the internal function that executes the selected command are called in turn.

```
/*****************************************************************************
***
 * @brief            ABC Module.
 * @details          xxx Software Product Line Project.
 * @author           ooo
 * @date             2020-10-15 AM. 8:40:59

 * @version          1.0.0

 * @file             abc.c
 * @copyright        Copyright by ooo.
 * @warning          Target is unknown. Complier is unknown
 * @note
 * @par
 * Project: xxx Software Product Line Project.
 * @par
 * Target: Unknown.
 * @par
 * Complier: Unknown.
 * @par
 * Features: Fault Priority

*****************************************************************************
**
*/
```

Fig. 6: Example of using doxygen with comments

5. Conclusion

This paper is case study of domain engineering for developing the software product line. Business domain of this software product line is sunroof of automotive. This company hoped to shorten the development period of products for various customers and efficiently reuse the products that were developed by improving the engineering skills of developers through appropriate development methods. Domestic automobile parts development firms have only one or two developers, with a shortage of human resources. In order to overcome this, this company decided to adopt software product line engineering technology as a solution to increase the reusability and maintainability of software. In this paper, several activities and mechanism for planning, architecture design, and module development for the software product line are introduced.

This case study was performed at small organization so that not many developers have maintained the software. Also there were no standard for implementation and no supporting environments. The size of software was not huge and the maturity of

engineers are good. The domain maturity is good and very stable so that adoption of product line engineering is a critical success factor for improving the quality and productivity (Ahmed F et. al 2011, Krüger J, Berger T 2020, Faheem Ahmed 2006, Jinseub Kim et.al 2021).

The lessons learned from this case of technology application are as follows. First, the reference examples of SPL application of automotive embedded controller software have been obtained, and lessons were learned for performing SPL application in the automotive domain. Second, a case of SPL application for a small development team with less than five developers has been acquired. Lastly, a foundation for collecting quantitative results in the future has been laid based on the case of the SPL application for the development of a product used for mass-production in the near future

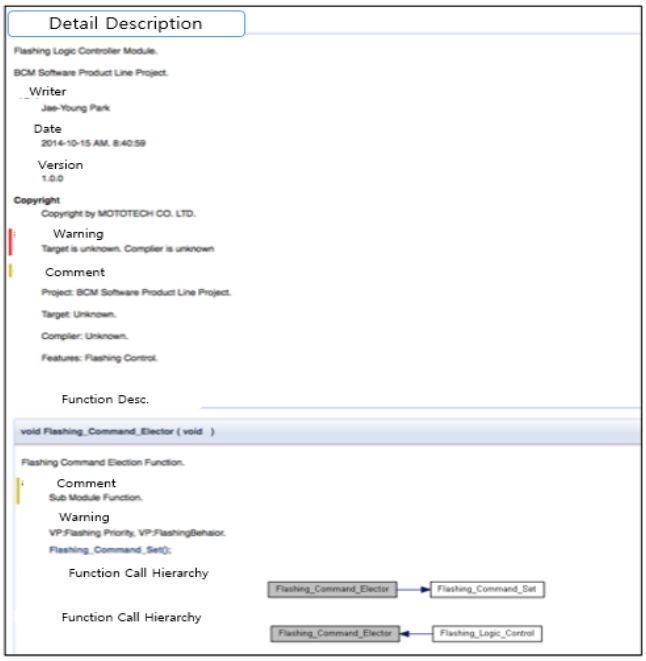

Fig. 7: Generated design documents from doxygen

Acknowledgments

“This work is supported by the Korea Agency for Infrastructure Technology Advancement (KAIA) grant funded by the Ministry of Land, Infrastructure and Transport (Grant 21RSCD-C163348-01).”

References

K Pohl, G Böckle, FJ van Der Linden (2005). Software product line engineering: foundations, principles and techniques, Springer.

Robert Lindohf, Jacob Krüger, Erik Herzog, Thorsten Berger (2021). Software product-line evaluation in the large, *Empirical Software Engineering*, 26, 30(2021).

Krüger J., Mahmood W., Berger T. (2020). Promote-pl: A round-trip engineering process model for adopting and evolving product lines. *International conference on systems and software product line engineering*.

Kyo C. Kang, Sajoong Kim, Jaejoon Lee1 , Kijoo Kim, Gerard Jounghyun Kim, Euiseob Shin (1998), FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures, *Annals of Software Engineering*, 5(1), 143–168.

Mikoláš Janota, Joseph Kiniry, Goetz Botterweck (2008). Formal Methods in Software Product Lines: Concepts, Survey, and Guidelines, Lero Technical Report Lero-TR-SPL-2008-02.

Joachim Bayer, Oliver Flege, Peter Knauber, Roland Laqua, Dirk Muthig, Klaus Schmid, Tanya Widen, Jean-Marc DeBaud (1999). PuLSE: A Methodology to Develop Software Product Lines," *Proceedings of the 1999 symposium on Software reusability*

Kalender ME, Tüzün E, Tekinerdogan B. (2013). Decision support for adopting SPLE with Transit-PL. *International software product line conference*. ACM, SPLC, pp 150–153,

Rincón L, Mazo R, Salinesi C (2019) Analyzing the convenience of adopting a product line engineering approach: An industrial qualitative evaluation. *International systems and software product line conference*. ACM, SPLC, pp 90–97

Berger T, Steghöfer J, Ziadi T, Robin J, Martinez J. (2020). The state of adoption and the challenges of systematic variability management in industry. *Empirical Softw. Eng.*, 25(3), 1755–1797.

FJ Van der Linden, K Schmid, E Rommes (2007). Software product-line in actions, Springer,

Nešić D, Krüger J, Stănciulescu Ş, Berger T. (2019). Principles of feature modeling. *Joint meeting on European software engineering conference and symposium on the foundations of software engineering*. ACM, ESEC/FSE, 62–73.

El-Sharkawy S, Yamagishi-Eichler N, Schmid K. (2019). Metrics for analyzing variability and its implementation in software product lines: A systematic literature review. *Information and Software Technology,* 106, 1–30.

Berger T, Nair D, Rublack R, Atlee JM, Czarnecki K, Węsowski A. (2014). Three cases of feature-based variability modeling in industry. *International conference on model driven engineering languages and systems. MODELS*, pp 302–319.

Steffen Thiel and Andreas Hein (2002). Modeling and Using Product Line Variability in Automotive Systems, *IEEE Software*, 22(4). 66-72

MA Laguna, Y Crespo (2013). A systematic mapping study on software product line evolution: From legacy system reengineering to product line refactoring, *Science of Computer Programming*, 78(8), 1010–1034.

Len Wozniak, Paul Clements (2015). How automotive engineering is taking product line engineering to the extreme, *Proceedings of the 19th International Conference on Software Product Line*. DOI: 10.1145/2791060.2791071

JeongAh Kim, JinSeok Yang (2018). Practice of Hybrid Approach to Develop a State-based Control Embedded Software Product Line. Advanced Multimedia and Ubiquitous Engineering, FutureTech 2018.

Ahmed F, Capretz LF. (2011). A business maturity model of software product line engineering, *Information Systems Frontiers*, 13(4), 543-560.

Krüger J, Berger T. (2020). An empirical analysis of the costs of clone- and platform-oriented software reuse. *Joint European software engineering conference and symposium on the foundations of software engineering*. ACM, ESEC/FSE.

Faheem Ahmed, Luiz Fernando Capretz. (2006). Maturity Assessment Framework for Business Dimension of Software Product Family. *Interoperability in Business Information Systems*. 1(1), 9-32.

Jinseub Kim, Hakyun Kim. (2021). A Study on the Influence of the Service Quality of Container Terminal on the Reuse Intention, *Asia-pacific Journal of Convergent Research Interchange*, 7(8), 29-38.