An approach to feature-based software construction for enhancing maintainability

Jungyoon Kim*† and Doo Hwan Bae

SUMMARY

While the way we build software affects significantly its maintenance in terms of the effort and cost, the experience level of the maintainer in a software acquirers’ organization is also one of concern. In this context, often the maintainer is the user of the system. Unfortunately, it is quite possible to lose the trustworthiness of the software due to the inexperience of the maintainer, especially when the maintainer is without the help of the original developers. One remedy for providing security against the effects of the maintainer’s software modifications is to restrict the access to software parts (modules) relative to the experience level of the maintainers. For such a remedy to be successful, the software should be constructed in such a way that its parts under maintenance affect others as little as possible. We propose an approach to software construction aligning the dependencies among software parts in one direction so that they are allocated to maintainers based on their experience level. Our approach decomposes the software into parts based on functionality and orders the parts by essentiality, which indicates how difficult it is to change each part. Then, we align the dependencies in such a way that the less essential functionality is dependent on the more essential functionality. Consequently, any modification on less essential functionality does not affect the essential functionalities. To demonstrate the feasibility of our proposed approach, we applied it to a military application and found that the constructed software enables us to confine maintainers’ activity within a limited working area, and thus the software is safer against maintainers’ modification. Copyright © 2006 John Wiley & Sons, Ltd.

Received 31 January 2005; Revised 30 September 2005; Accepted 14 November 2005

KEY WORDS: feature; dependency alignment; essentiality; maintainer; unidirectional dependency
1. INTRODUCTION

Software maintenance takes a significant portion of the life cycle of large software systems. However, it is software construction rather than the maintenance itself that has a major impact on the maintainability [1,2]. Suggested solutions for improving maintainability include minimal coupling (dependencies among modules) [3] and aligning dependencies in one direction. Layered architecture [4–6], where an application component layer is dependent on other layers or shared libraries, is an example of the latter solution. Our approach has a similar view where dependency alignment alleviates the burden of maintenance.

In large logistics information systems, the trustworthiness of fundamental functionalities, such as resource distribution policies or engagement rules§, must be preserved during maintenance. Berstein [7] listed constraints (reliability, security, privacy, business integrity, etc.) that keep a system trustworthy. He also pointed that the human capability is limited and system defects are correlated with the personnel practice. We believe that satisfying these constraints is the major maintenance problem. To alleviate such a problem, it is recommended to limit peoples’ working area (i.e. module) and their usage of language features. For large logistics information systems, maintainers in software acquirers’ organizations need to maintain such software by themselves, because the organizations cannot always afford to hire skilled experts (or original developers) for frequently recurring or tedious maintenance. Also, some military software systems need on-field modification for certain situations. The problem arises when such systems are maintained by unskilled maintainers who might deteriorate the trustworthiness in that constraints are not preserved as originally planned. Such deterioration can be controlled by dependency alignment. Dependency alignment enables the confinement of ripple effects caused by the maintainers’ modifications within a limited working area.

We call such confinement of the maintenance activities ‘leveled maintenance’. In leveled maintenance, maintainers are categorized into different levels based on their expertise. This is inspired by the maintenance policy of the U.S. Department of Defense [8] for aircraft maintenance in which the leveled maintenance policy categorizes each maintenance activity for aircraft in different levels.

Leveled maintenance for software requires the limiting of different working areas according to maintainer’s expertise (novice or expert) as shown in Figure 1, but we could have more than two levels, depending on the expertise level. We wish that each working area should not affect others. However, this is not possible unless each working area becomes an independent system. Although the dependencies among working areas are unidirectional, we would like to control them by allowing unidirectional dependencies. In other words, we want to align dependencies in such a way that, in Figure 1, the working area 1 depends on 2, not vice versa, i.e. in one direction to keep the trustworthiness of working area 2 against the modification in working area 1. Similarly, layered architecture [4] addresses unidirectional dependencies, and so do many approaches including Aspect-Oriented Programming (AOP) [9] and feature refinement [10], suggesting a unidirectional dependency between the core (base) and refinements (aspect) of software. Yet, there is a serious lack of clear guidance from the requirements analysis phase to the coding phase to support the construction and maintenance of software. Guidance must be provided with alignment criteria to place parts of software, and thus leveled maintenance can be applied to the software after delivery.

§These are different according to whether it is peace time or war time.
We define the essentiality of a software part as the frequency of future changes of the part including two factors: users’ qualitative determinations of change anticipation, and dependency relations with other parts. It is inferred from Graham [11] that: essential parts are stable in time and space. It also reflects the stability terms of Yau and Collofello [12] and Martin [13]. We consider essentiality as the alignment criteria. Use cases [14] of a domain can be categorized as essential or accidental as shown in Figure 2, where accidental use cases are usually dependent on essential use cases, but not vice versa. We found that many of them have the tendency to unidirectional dependency. For example, in the telephone calling system [15], the calling operation is the basic operation of the system while the billing operation is dependent on the calling because the billing amount is calculated based on the calling records. Also, the billing operation is more subject to change than the calling operation.

Traceability is the modeling property to map from a functionality, a segment of requirements equivalent to the use case of the Rational Unified Process (RUP) [14], to an implemented module and vice versa. In leveled maintenance traceability from requirements to implemented code is required and managing dependencies unidirectionally is necessary across the development process. Figure 3 illustrates conceptually the development process for sustaining traceability and unidirectional dependency. Each column represents a development phase (domain analysis, requirements analysis, design and implementation), and each oval means a functionality. A horizontal arrow between functionalities represents traceability: both side arrowheads indicate the back and forth mapping in iterative development. A vertical arrow indicates the unidirectional dependency: each vertical arrow

---

Figure 1. Different working boundaries.

Figure 2. Unidirectional dependencies.

---

\[\text{In practice, the relation from modules to functionalities can be regarded as the ‘total subjective function’ (modules} \rightarrow \text{functionalities).}\]
points to the more essential functionality. The properties of traceability and unidirectional dependency in the software construction facilitates the leveled maintenance. To support such properties, we identify the following three issues to be addressed:

1. **Classification of functionalities based on essentiality.** At the initial phase of software development, the classification of functionalities can be decided by users, and later it will guide the alignment of dependencies among functionalities. The classification is similar to the prioritization of Graham [11]: before designing software we have to analyze business procedures in users’ requirements, so as to extract candidate functionalities. Because we may not include all of them in the software construction, we need to decide which functionalities will be included into the implemented software system according to the priorities of functionalities. This classification needs the users’ involvement at the requirements analysis phase.

2. **Functionality encapsulation.** This enables us to acquire traceability. In traditional Object-Oriented Programming (OOP), a functionality of a system may not be properly encapsulated, instead it may be scattered across structural elements such as classes [16]. In the leveled maintenance, when a functionality changes, maintainers must be able to identify the modules relevant to the changed functionality. If the functionality is not traceable across different phases, maintainers may fail to find and modify the corresponding modules. Since traditional OOP cannot successfully provide such functionality-based traceability [16,17], we need to encapsulate the scattered parts of a functionality. Another challenging point for encapsulated functionalities is how to keep the unidirectional dependency across development phases.

3. **Implementation support.** At the implementation level there are many dependencies to be considered and we therefore need to abstract them for alignment without losing the meaning of the original program [18]. As one of the tools that support such abstraction, AspectJ reverts directions of dependencies or removes dependencies in Java code (low-level implementation) without any semantic changes. This is a type of **dependency inversion** [13,19].

---

[1] Another type of dependency inversion is the inheritance in OOP. In traditional C programs, a main module is dependent on (calls) concrete submodules. However, the superclass in C++ is not dependent on the concrete subclasses.
This paper focuses on the second issue. The first classification of functionalities based on essentiality is an active research area in requirement engineering. In our approach, we empirically differentiate functionalities. In future work, a more robust technique must be developed. Previous approaches have already attempted to address this problem (see the languages listed on www.aosd.net).

We devised Feature-Based Modeling (FBM), a software development approach that facilitates the leveled maintenance. From use cases FBM extracts features, the logical units reflecting functionalities in requirements analysis, onto design and code and then aligns the dependencies of the features in one direction. Since the features reflect users’ requirements, developers or maintainers can easily map the functionalities to the features in design and code. To demonstrate the applicability, we applied FBM to the development of an aircraft maintenance information system, a military application. FBM keeps the features in design and code, and through unidirectional dependencies makes the leveled maintenance feasible. This technique can be used as a supplement to other conventional development processes, such as RUP [14].

This paper is organized as follows. Section 2 presents a brief summary of related work that shares a similar insight with our work. Section 3 introduces the important concepts in our approach and discusses the process for applying our approach during software construction. In Section 4, we validate our work with a case study. Finally, Section 5 discusses the lessons learnt and concludes with future work.

2. RELATED WORK

Batory [20] suggests feature refinement as an approach to stepwise refinement [21]. In the feature refinement, a feature is a product characteristic that is a distinguishing behavioral sequence. The feature reflects mainly the user’s requirements, so thus the maintainers can easily and intuitively understand the unit. Many terms similar to feature are used, such as aspect [15], subject [17], and layers [10]. They usually encapsulate fragments of multiple classes, while the traditional module encapsulates sets of complete classes. Our work is concerned with feature refinement focusing on aligning features according to their essentialities. The following is a summary of approaches to feature refinement that share a similar insight with our work.

(1) Functionality-based software decomposition. Since traditional OOP decomposes software by its static structure, it loses traceability in that a distinctive functionality can be scattered during the development process. Subject Oriented Programming (SOP) [16,17,22] tries to overcome this ‘tyranny of the dominant decomposition’ and thus enhances traceability by software decomposition based on subjects that are defined as a specific user’s concerns, while HyperJ is the realization of SOP. We may use SOP to realize the leveled maintenance, but FBM needs to address functionality alignment to provide the guidance on how to manage a dependency. As another approach to functionality-based software decomposition, AOP [9] can provide such guidance with the property of unidirectional dependency in that an aspect is dependent on its base but not vice versa. Both SOP and AOP can provide implementation support for FBM, while FBM further extends such support to requirements analysis and the design phase to realize the leveled maintenance.

Feature Oriented Domain Analysis (FODA) [23] defines the feature as a prominent or distinctive user-visible aspect, quality or characteristics of a software system. However, FODA leaves room for life cycle support and does not intend to consider design/coding issues or dependency alignment.
The Use Case Driven Approach [24] has been successful for enhancing traceability, and it has been incorporated into RUPs [14]. Unfortunately, this approach is inherently based on OOP, so the use cases may disappear in the static structure and implemented code, easily losing traceability.

Feature Oriented Programming (FOP) [25] proposes a model for flexible composition of objects from a set of features that are services of objects. The feature in FOP is similar to the feature of FBM in that both are identified in the requirements phase and are visible to people (users, developers and maintainers) while the feature of FBM has larger granularity: visible means the ability to be understood by people. FOP uses Lifter to compose features. It does not overwrite methods whereas inheritance in OOP does. For example, Countered locker, where Counter counts the size of stack and Lock is a switch to allow/disallow modification, can be assembled by attaching two features, Counter and Lock using Lifter [25]. In comparison with FBM, FOP is not concerned with dependency alignment and FBM uses aspect as a glue to attach the links between features. FOP claims that an object with individual services can be created just by selecting the desired features. It may be useful in applications where a large variety of similar objects are needed, but such features (statical elements: one class with one functionality) may not practically exist in management information systems.

Theme [26] extends the concerns of AOP to the requirements analysis phase. Theme is an approach for showing the relationships between behaviors by identifying and isolating aspects in the requirements documentation. It models aspects with design languages. It is similar to our approach in that both support the software development process with software decomposition by a user visible unit (action in Theme and feature in FBM). It covers software development phases starting from action view for the requirements analysis, theme view for the preliminary design, then to theme/UML for UML design notation and finishes in augmented view for more refined design. However, Theme is not concerned with dependency, but is mainly concerned with how to identify aspects in requirements and connect them with the design and code.

(2) Unidirectional Dependency. Component Based Development (CBD) [5,6,27] provides strong encapsulation with well-defined interfaces, and its layered architecture realizes unidirectional dependency relationship among components as shown in Figure 4. Previously, because of their lack of dependency abstraction, OOPs have encountered difficulty in realizing such a layered architecture [5,28]. In Figure 4, the components in the upper layers are dependent on the components in the lower layers. It is also recommended that the components in the same layer should not depend on each other. Such unidirectional dependency in CBD is the relationship between services. The services are not directly mapped to requirements because CBD decomposes software by its structural entities as does OOP. Thus, traceability of CBD may be similar to that of OOP. Moreover, CBD does not intend to reflect the user visible functionality such as OOP. Collaboration-based development [10], in which layers are in unidirectional dependencies, enables layer level reuse. However, the layer is a collaboration across classes and can be extended by parameterized inheritance only (e.g. template in C++), and thus it can be applied only in some specific domains [29].

3. FEATURE-BASED MODELING

This section introduces FBM by explaining key concepts of FBM and the FBM process for software development.
3.1. Feature

The concept of the feature is partially borrowed from Eisenbarth et al. [30]. Eisenbarth defined a feature as a set of computational units (i.e., methods of classes). The feature can be triggered by multiple scenarios and is closely linked to the functionality described by the scenarios in users’ requirements. This enables the feature to be highly user visible. We redefined the feature and related concepts as follows.

- **Computational Unit**: an atomic executable part of a computational module.
- **Scenario**: a functional and temporal description of a use case (a user’s requirement that can be represented by a UML sequence diagram).
- **Feature**: a sequence of computational units that can be triggered by multiple scenarios**.

![Unidirectional dependencies in CBD.](image)

Figure 4 shows the relationship of the feature and related entities using a UML class diagram. They can be formally defined as follows:

```plaintext
Feature ::= *CUnit
CUnit ::= ICUnit | PCUnit
PCUnit ::= type target [condition]
type ::= t | u | s
  t::= triggered
  u::= uses
  s::= shares
target::= Pointer to an ICUnit
condition::= condition of feature that triggers PCUnit
```

**Usually, a feature invoked by a scenario is functional, and the feature shared (triggered) by multiple scenarios is usually non-functional in that it is not directly mapped to the user’s requirement.**
A feature is defined as a sequence of computational units. A computational unit can be an implemented unit (ICUnit) that is actually implemented in a feature or a pointer unit (PCUnit) that is a proxy of an implemented unit. Each pointer unit has type, target and condition. The pointer (PCUnit) in a feature points to an implemented unit (ICUnit) in another feature. Pointer types are 'triggered' (t), 'uses' (u) and 'shares' (s). The 'uses' printer type represents a method call and is the basic style of dependency. The pointer (PCUnit) of 'uses' type uses the ICUnit to which the target points, and this relation is equivalent to the 'includes' relation between use cases. The other two types ('shares' and 'triggered') are derived from 'uses'. The 'shares' pointer type represents the indirect dependency that occurs between multiple calls to a method. Some pointers of 'shares' type share the ICUnit to which the target points to. The pointer of 'triggered' type is triggered when the condition ([condition]) of the ICUnit to which the target points is true. For example, in the phone calling system of Figure 2, the time recording (timing) is triggered whenever the calling operation starts. In Figure 5, the cardinalities between Scenario and Feature are both one-to-many (1...*), implying that a feature can be triggered by multiple scenarios and a scenario can be composed of multiple features. The pointer unit (PCUnit) is introduced to manage dependency inversion. Figure 6 illustrates an example of the feature. Both shaded parts in the two scenarios of Figure 6(a) represent the shared part and becomes Feature 3 in Figure 6(b). Each pointer unit in Features 1 and 2 points to the first unit of Feature 3.

Since users are mainly concerned about requirements change, they need to be able to find the design model and code relevant to the changed requirements. Thus the segment of software to be maintained must be user visible. We use the term ‘functionality’ that reflects such a segment of the users’ requirements, and each segment is a function or service that software provides to users. ‘Functionality’ is equivalent to the term ‘use case’ in UML, but functionality is used in this paper to emphasize that our work is an approach to functionality-based software decomposition [16].
which is different to structural-based software decomposition such as OOP and CBD. The feature is the module that reflects a functionality. The terms functionality and feature have separate meanings in our work. Functionalities are extracted from the users’ requirements, which can be specified by a UML sequence diagram. During requirements analysis, users order the functionalities giving priority. On the other hand, a feature reflects a functionality and is used as a logical unit to be designed and implemented. In short, both describe the same notion at different development phases: functionality is for requirements analysis and feature is for design and implementation. In addition, an accidental feature is an intrusive concern that watches what a more essential feature does but never disrupts. In the early phase of FBM, the possibility of dependency from an essential feature to any accidental feature will be identified and removed by functionality alignment.

3.2. Inertia

Many approaches have tried to link maintainability with complexity [2,31] and such linking is closely related to dependency among elements of software [13,12]. The main reason for such linking is the ripple effects caused by a code change. The ripple effects contribute to the increase in resistance to a change. Yau and Collofello [12] defined such resistance as stability. In our approach, essentiality mainly refers to this stability. Essentiality may mean more than stability, but only the stability of existing artifacts (design model, code) can be measured.

An empirical metric, inertia, is designed to capture the degree of feature essentiality and to represent the vulnerability created by dependencies among features. The order of features’ essentiality can be depicted on a feature dependency graph as shown in Figure 7. This graph shows the dependency relationship between features in a software system. The graph is composed of ovals, each of which represents a feature, and arrows that indicate the dependency directions. The feature from which an arrow starts is dependent on another feature to which the arrowhead points. A feature dependency graph visualizes the dependency relations in which inertia indicates the order of a feature essentiality.
Inertia in this paper is composed of the two different metrics: Martin’s Stability ($St$) [13] and Abbott’s Interface Size ($IS$) [32,33]. Martin provided a simple metric of stability defined as

$$St = \frac{D_i}{D_i + D_o}$$

where $D_i$ is the number of incoming dependencies to the feature, and $D_o$ is the number of outgoing dependencies from the feature. $St$ increases when the number of incoming dependencies to a feature increases, and this means that the feature must be more stable because more features depend on it.

The Interface Size ($IS$) has been introduced by Abbott as a heuristic but empirical metric, and Bandi validated it through a statistical experiment [33]. $IS$ measures the complexity of a class by calculating the size of methods and parameters of the methods. It is defined as

$$IS = K_1 \times p + K_2 \times s$$

where $p$ is the number of parameters and $s$ is the sum of sizes of parameters. The sizes of parameters are specified in Bandi’s work [33]. The constants $K_1$ and $K_2$ are tentatively set to 1, and they are subject to change through future validation.

The complexity of a feature should reflect the essentiality of a feature considering stability, anticipation of change, internal complexity and dependency relations. Since it is not practical to consider a complete set of factors, we only consider two factors (Abbott’s interface size as internal complexity and Martin’s stability for dependency relations). The complexity of a feature ($C$) can be interpreted as the partial order of $IS$ and $St$, that can be defined as

$$C = IS \times (St + k)$$

where the constant $k$ is set to 1 for convenience (note that $St$ is a normalized value). This partial ordering places a weight on the $IS$ (giving priority on interface size). Finally, the inertia ($In$)\textsuperscript{††} reflects external dependency including not only directly neighboring features but also indirectly related features by summing them whereas $St$ only reflects the direct feature. The inertia $In_e$ of the feature $e$ in a feature

\textsuperscript{††}In physics, the moment of inertia of a body is not only related to its mass but also the distribution of the mass throughout the body.
Inertia values updated by dependency inversion assuming each IS value of all features is 1.

dependency graph is

\[ In_e = \sum_{i \in D_e} C_i \]  

(4)

where \( D_e \) is the set of features dependent on the feature \( e \) and \( C_i \) is the complexity of a feature \( i \). For example in Figure 7, the \( In_e \) is the sum of the values of \( C_i \) of the shaded features, \( a, b, c \) and \( e \).

Inertia indicates the (lattice) order among features on a feature dependency graph. In short, a feature \( A \) dependent on a feature \( B \) cannot have a larger \( In \) than that of \( B \). Inertia will be used for feature comparisons and will be repeatedly updated whenever any related dependency direction is newly determined or reverted (dependency inversion will be explained in Section 3.3). Figure 8 shows an example of inertia updates that are caused whenever dependency is reverted or newly determined. The example assumes each IS value of all features is 1 for simplicity. In Figure 8(a) F4 has the largest inertia value, but after the updates of the dependency directions among F1, F3 and F4, now F1 in Figure 8(b) has the largest value. Inertia can be regarded as a simple vector space model [34] such as the combination of the depth on the feature dependency graph and internal complexity [depth, IS] where depth is the distance from the most essential feature. To decide which feature is more essential when using a vector space compare the depth space of the two features and if the depths are the same then compare the IS space values of the features (note that the depth is more significant than IS since feature \( A \) cannot have larger \( In \) than that of \( B \)).

### 3.3. Feature alignment

Features are aligned according to the alignment rules that guide developers/maintainers to determine the dependency direction among features. The rules can be inferred from the testing approaches that address various types of ripple effects [35,36]. The four rules summarized in Figure 9 are as follows.

- **Rule 1: Mutual Dependency.** If Feature 1 calls Feature 2, and vice versa, both become one feature. This is typical cyclic dependency. The cyclic relation is normally removed at the initial stage of software development.
- **Rule 2: Read-write Dependency.** Both features share data and Feature 1 can read and write, but Feature 2 needs only to read. In such a case, we determine Feature 2 to be dependent on Feature 1.
Users describe the dependencies in Rules 1 and 2 during the requirements analysis. Suppose we have two dependent functionalities. If one functionality is decided as essential and the other as accidental, then the dependency direction will be decided as the accidental feature to be dependent on the essential feature. For example, in the phone calling system of Figure 2, ‘timing’ records the calling event on database, but ‘billing’ only reads the records (read-only).

- **Rule 3: Dependency Inversion.** Only Feature 1 calls Feature 2, not *vice versa* and there is no return value. The dependency direction can be inverted if Feature 1 is more essential than Feature 2. In the case of a use case inclusion or extension, the base use case‡‡ is dependent on the included use case while extended use case is dependent on the base use case. In the design and coding phases, the part for the base use case must call other parts for both included or extended use cases. This means that the base use case is dependent on both included and extended use cases. To keep the unidirectional dependency, we have to invert the dependency directions in that the included or extended use cases are dependent on the base use case, similar to the dependency inversion mechanism of AspectJ, as shown in Figure 10(a). However, the dependency direction between base and included use cases may not always be inverted: if the included use case returns any value to the base use case, the base use case must be dependent on the included use case. In this case, it is considered that the included use case is more essential. In reality, the included use case is usually shared by multiple use cases. For example, ‘user authentication’ can be shared by multiple transaction use cases, such as ‘withdrawal’ and ‘deposit’.

‡‡Base use case: a use case that includes other included use cases or extends to other extended use cases.
A.\textsubscript{ma} calls B.\textsubscript{mb}

\begin{itemize}
\item \textbf{Rule 4: Unit Sharing}. Both features share a computational unit. The shared unit will be implemented in only one feature among all features that share the unit. For example, assume two features sharing one unit. At first, this ‘unit sharing’ does not indicate any dependency direction, but when one feature is selected to implement the unit, the dependency is apparent. The identification of the feature that implements the ‘shared unit’ is decided by observing the essentiality of the two features. The selection is decided through consideration of the essentiality of the two features. In Figure 10(b), the shared unit \textit{m} can be implemented in either one of the features and the other feature will have a proxy of \textit{m}. The implementation of \textit{m} will be placed in the feature that is more essential.
\end{itemize}

In addition to the rules explained above, there could be write–write, or read–read dependencies. However, write–write dependencies can be treated by Rule 1, and read–read dependencies can be ignored because they do not affect each other. The above rule set may not be complete. The rules are developed heuristically through several project implementations. Also, it is possible to extend or enhance the rules as needed.

3.4. The FBM process

The FBM process has four steps to extract and align features, as shown in Figure 11. The first step, ‘Functionality Alignment’, identifies the essentiality of each functionality with users’ help. The essentiality will reflect not only the dependency relations among functionalities but also the change in anticipation/frequency and criticality. The result will be a use-case diagram with dependency relations. The second step identifies features from the functionalities. The third step identifies shared computational units among features, and finally determines the dependency directions by considering preliminary dependency directions in the functionalities and the alignment rules given in Section 3.3.

A simplified version of the Hotel Reservation System (HRS) is used for illustrating the FBM process in the following explanation. The HRS has three functionalities. First, a customer tries to reserve a room. Second, the customer needs to check the availability of a room, prior to the room reservation. Finally, if the reservation cannot be successful because the room is not available, then the customer
requests their name to be placed on the waiting list. Note that the term ‘use case’ is used as it is used in RUP [14] while the equivalent term ‘functionality’ is used as defined in FBM, an approach to functionality-based software decomposition [16].

**Step 1. Functionality Alignment**. Step 1 categorizes the functionalities identified at the requirements analysis phase according to their essentiality. The inquiry form shown in Table I is handed out to the users who have documented the requirements. In Table I, each cell in the left column is the title of each functionality. The users list certain functionalities in a cell of the right column if those are dependent on the requirement in the left column in the same row. The result of the inquiry form can be represented in a directed graph as shown in Figure 12(a). Existing circular dependencies among functionalities can be removed by inverting any possible dependency according to Rule 3 to break the cycle. If such inversion is not possible, then the functionalities in the cyclic relation must be merged because those functionalities are closely related and can be regarded as a single feature (Rule 1). The final result will show an acyclic directed graph as shown in Figure 12(b), where each oval becomes a *functionality* ordered from accidental to essential. In Figure 12(b), we say that S6 is more essential than S3, 5, and S3, 5 is more essential than S1, 2, 4. This dependency relationship will be included into the original requirements in the form of use-case scenarios.

During Step 1, users will align functionalities by essentiality considering anticipation for changes and change frequency. Note that the dependencies identified by inquiry form, as in Figure 12(b), are not always consistent with users’ opinions. For example, even if the inquiry form indicates S3, 5 is dependent on S6, users may insist that S3, 5 is more essential when they consider anticipation to change. In such a case, we recommend following users’ opinions.

Figure 13 shows an example of Step 1 for the HRS. The use-case diagram in Figure 13(a) shows the relations among the functionalities. The inquiry form in Figure 13(b) shows the dependency relations among functionalities. The dependency relation can be represented as shown in Figure 13(c).

<table>
<thead>
<tr>
<th>Scenarios</th>
<th>Scenarios that are dependent on the left scenario</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>Subset of {S2, S3, \ldots, SN}</td>
</tr>
<tr>
<td>S2</td>
<td>Subset of {S1, S3, \ldots, SN}</td>
</tr>
<tr>
<td>\ldots</td>
<td>\ldots</td>
</tr>
<tr>
<td>Si</td>
<td>Subset of {{S1, S2, \ldots, SN}/{Si}}</td>
</tr>
<tr>
<td>\ldots</td>
<td>\ldots</td>
</tr>
<tr>
<td>SN</td>
<td>Subset of {S1, S3, \ldots, S(N - 1)}</td>
</tr>
</tbody>
</table>
In this example, the alignment is decided as follows: since S1 calls S2, the base use case (S1) is dependent on the extended use case (S2). By Rule 3 in Section 3.3, the dependency direction of ‘extends’ will be reverted so that the original dependency in the use-case diagram (extends) will not be changed in the design and implementation phases: now, the extended use case (S2) is dependent on the base use case (S1). On the other hand, the dependency relation between S1 (base use case) and S3 (included use case) will be kept because S3 ‘check room availability’ must return a value to S1. S3 cannot be dependent on S1 as explained by Rule 3 in Section 3.3 (the case where dependency inversion is not possible). On the other hand, let us consider the case where S3 is ‘logging’, with no return value. In this case S3 can be dependent on S1, and thus the original dependency direction of the ‘includes’ relation can be reverted.

The preliminary inertia values of each scenarios are calculated on the dependency graph in (b) as follows. Since the factors of $IS$ have not been identified, the $IS$ values are tentatively set to 1 for each scenario. The $Sr$ values of functionalities in Figure 13(c) can be calculated using Equation (1) as 0 for S2, 0.5 for S1 and 1 for S3. The $C$ values can be calculated using Equation (3) as 1 for S2, 1.5 for S1 and 2 for S3. Finally, the inertia values are initially calculated using Equation (4) as 1 for S2, 2.5 for S1 and 4.5 for S3.
Step 2. Feature Identification. This step identifies features from the output of Step 1, as illustrated in Figure 14. Originally there are three functionalities in Figure 13, and those are described by the three scenarios of sequence diagrams S1, S2 and S3 in Figure 14. Two functionalities, S1 and S2, in Figure 14 share the end part of their sequences, and the shared parts become a new feature F4. The remnant parts of the two functionalities become two features F1 and F2. The functionality S3 itself becomes a feature F3.

In this step, inertia values of features F1, F2 and F3 are the same as the inertia values of the functionalities, S1, S2 and S3, while values of the newly extracted feature F4 has not been calculated because the dependency relation of F4 has not been determined. However, a feature in which the parameters of methods are identified can have its interface value. The following example calculation

---

Figure 14. An example of feature extraction (Step 2) for HRS.
using Equation (2) is the IS of F2 ‘handle waiting list’ in Figure 14 under the supposition that the first method has two input parameters (ID and Date), the second method has one output parameter (availableRooms) and the third method has two parameters (ID and Date). Note that the two classes, G and H, are not involved in F2.

<table>
<thead>
<tr>
<th>Handle waiting list</th>
<th>Interface Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: WaitingWindow.requestWaitingList(ID, Date)</td>
<td>2 + (3 + 3) = 8</td>
</tr>
<tr>
<td>2: Room.checkRoom():availableRooms</td>
<td>1 + (6) = 7</td>
</tr>
<tr>
<td>3: WaitingList.placeRoom(ID, Date)</td>
<td>2 + (3 + 3) = 8</td>
</tr>
<tr>
<td>4: WaitingWindow.confirm():</td>
<td>0 + (0) = 0</td>
</tr>
</tbody>
</table>

\[ \text{total} \quad 23 \]

The calculations are inspired by Bandi et al. [33] who indicate the sizes of parameters (i.e. Array is ‘3’ and Object is ‘6’). The method ‘1: WaitingWindow.requestWaitingList(ID, Date)’ has two parameters, ID and Date, that make the first ‘2’ (the number of parameters). ID and Date are considered as a character array, and thus they make the second ‘3’ and the third ‘3’. The interface size of the method is now ‘8’. In the same way, the IS for the other methods are calculated. The total interface size of the feature is 23 by summing up all the sizes of methods.

**Step 3. Shared Unit Identification.** This step identifies any unit shared by more than one feature. Figure 15 shows the four features identified in Step 2. Each oval represents a feature and each rectangle represents a computational unit in which a capital letter represents a class and the following sequence of lower characters with digits represents a method in the class. The computational unit B.mb1 inside of the dotted circle is shared by the features F1 and F2. The St values are calculated using Equation (1) as 0 for F2, 0.5 for F1 and 1 for F3, which are the same as the values of the functionalities S2, S1 and S3 in step 1 because the features and functionalities can be mapped one to one. However, the new feature F4 has no stability because the direction of the dependency will be determined at Step 4 after considering other possible relations among features.

In this step, the interface size of each feature is no longer 1 because the computational units in the features are realized as shown in Figure 16: the IS values are 21 for F1, 23 for F2, 14 for F3 and 8 for F4.
The \( C \) values are calculated using Equation (3) as 23 for F2, 31.5 for F1 and 28 for F3. The \( C \) value of F4 has not been determined because it has no stability value. Finally, the inertia values are calculated using Equation (4) as 23 for F2, 54.5 for F1 and 82.5 for F3. Again, the new feature has no inertia value yet.

Step 4. Dependency Determination. This step determines the dependency direction among features based on the result of the previous steps (according to Rule 4 in Section 3.3). In this step, the types of dependencies are determined and specified using the feature definition given in Section 3.1, as shown in Figure 17(a). The relations ‘uses’ between F1 and F3, and ‘shares’ between F1 and F2 are the same as in the previous step. The relation ‘triggered by’ is newly specified for F4 (logging) and is...
triggered when a target unit meets a condition to be triggered (log when a certain method is called). For the relation between F1 and F2, F2 is triggered when F1 is not available (if a room is not available for reservation, then ‘Handling waiting list’ is triggered). The inertia values will be updated at each dependency determination by Rule 4. Finally, the feature dependencies are extracted and specified as shown in Figure 17(b).

At this step, all inertia values are determined. The finally updated stability ($St$) values are calculated using Equation (1) as 0.5 for F2, 0.7 for F1 and 1 for F3. The value 0 is assigned for F4 because it is now considered as the topmost feature indicating no feature is dependent upon it. The $C$ values are recalculated using Equation (3) as 34.5 for F2, 35.7 for F1 and 28 for F3. The $C$ value of F4 is 8. Finally, the inertia values are also recalculated using Equation (4) as 8 for F4, 42.5 for F2, 78.2 for F1 and 106.2 for F3.

3.5. Feature-based implementation and leveled maintenance

Figure 18(a) illustrates the features of the HRS as sets of class fragments. Each feature has a number of classes and the classes in a single feature are partial definitions. Those partial definitions are composed into Figure 18(b) by the languages such as AspectJ or Hyper/J that supports feature refinement*. We chose AspectJ over Hyper/J due to the simplicity and unidirectional dependency constraint support. However, we believe that Hyper/J can also be used for implementation. In Figure 17(b), each feature can be implemented as follows using AspectJ. For instance, F2 in Figure 18(b) has four computational units that make the sequence of a functionality ‘Handling waiting list’. The first unit E.me1{t.F1.A.ma1[cma1]} can be implemented as the following code:

```java
public aspect WaitingRequest {
    pointcut reserveFailed(UI ui): target(ui) &&
        call(void UI.reserve());
    after(UI ui) : reserveFailed(ui) {
        int room = ui.getRoom();
        System.out.println("In reserveFailed, room = "+room);
        if (room < 0){
            System.out.println("Now, putting on waiting list...");
            WaitingWindow waiting = new WaitingWindow();
            waiting.requestWaitingList();
        }
    }
}
```

The pointcut points to another feature UI and UI’s unit reserve(). reserveFailed() indicates the condition [cma1]. t indicates if reserve() failed then call E.me1 which is WaitingWindow.requestWaitingList(). The aspect WaitingRequest acts like the glue of the two classes (E and A) in the two different features (F2 and F1).

*Originally AOP aims to modularize crosscutting concerns that can be unplugged from the base program.
The leveled maintenance can be performed as follows: the features are categorized into differently leveled groups. The boundaries of the groups can be decided by the policy of the software acquirers’ organization, which may need the consent of the developers. The features in the groups of higher inertia values are relatively more essential than others, while the features in other groups of lower inertia values are more frequently changing and less essential. The maintainers who are the personnel in the software acquirer’s organization can only maintain the groups of the features of lower inertia values. However, the maintenance of the features of higher inertia values usually requires experts because the features are usually the fundamental features of the software, such as critical engagement or logistics rules that are not likely to change frequently in the future. The benefits of this idea are that we can save money to hire experts for frequently changing features, and we can reduce the possibility of trustworthiness deterioration by unskilled maintainers.

4. CASE STUDY

For the validation of our approach, we applied FBM to a military information system ‘Aircraft Maintenance Information System (AMIS)’ developed in Java [38].
4.1. Scope and result

AMIS is a mission application operating on top of the Defense Information Infrastructure Common Operating Environment (DII COE [37]). AMIS has 262 functionalities of 104,000 lines of code. In the previous project [38], six functionalities, as shown in Figure 19, were chosen and implemented as component-based software to show the feasibility of the common operating environment [37], a layered architecture. For this case study, the part of AMIS has been redesigned and re-implemented by four graduate students to show how the redesigned part meets our rationale for FBM. The students took six months for design and coding of the seven features. The part originally had 2813 lines of Java code. The size of the redeveloped part became 3092 lines of AspectJ code. During most of the development time, the students had to learn AOP with AspectJ programming and the FBM approach. They refactored the original Java code into AspectJ by adding eight aspects and removing some of the original code to realize the features. However, we realize that more effort is needed for requirements analysis, especially to identify the initial order of functionalities’ essentiality and also to extract/align features preserving the unidirectional property from design to code. The evaluation in this section focuses on the comparison of two versions, which are based on CBD (OOP) and FBM, respectively.

Redesigned AMIS shows the unidirectional dependency among the functionalities. Figure 20 shows only three functionalities as a part of AMIS. The methods in each class, represented as rectangles in Figure 20, are the computational units. The units of shaded classes are invoked by three different functionalities, and they become a new feature shared by the three features. The final inertia values of the features are shown in Table II. The features are ordered by the inertia values, and can be used as guidance for maintainers. The features in the order are grouped into different levels that tell whether maintainers can work on or not, as explained in Section 3.5.

4.2. Evaluation

We have examined the code in this case study with the evaluation metrics for OOP, as in previous experiments [39,40] which used object-oriented metrics [41] for evaluating AOP, since no metric

---

†DII COE is originally the specification of the U.S. Department of Defense. The DII COE in this paper is the Korean version.
has been established yet for evaluating AOP. We believe the code can be examined by the object-oriented metrics because it is based on OOP. We chose three metrics, Weighted Methods per Class (WMC), Coupling Between Object (CBO) and Lack of Cohesion of Methods (LCOM), and followed the approach of Tsang et al. [40]. In the evaluation, an aspect in AspectJ is considered as a class and an advice block as a method. We do not consider inheritance metrics, such as Depth of Inheritance (DIT) and NOC (Number of Children), because features (or implemented aspects) do not have inheritance.

The HRS, which has been used in Section 3.4 to show the example of the FBM process, is also evaluated. The results of the evaluation for AMIS and HRS are shown in Table III. The first metric, WMC, is a measure of the number of methods and advice blocks implemented within a class or an aspect.

---

Figure 20. Example functionalities in AMIS: (a) scenario: input flight time; (b) scenario: input part time; (c) scenario: input idle time.
Table II. Ordered features in AMIS.

<table>
<thead>
<tr>
<th>Order</th>
<th>Features</th>
<th>In</th>
<th>Order</th>
<th>Features</th>
<th>In</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Check part time</td>
<td>12</td>
<td>5</td>
<td>Set flight time</td>
<td>28</td>
</tr>
<tr>
<td>2</td>
<td>Check flight time</td>
<td>12</td>
<td>6</td>
<td>Update part time</td>
<td>61</td>
</tr>
<tr>
<td>3</td>
<td>Input idle time</td>
<td>14</td>
<td>7</td>
<td>Input flight time</td>
<td>113</td>
</tr>
<tr>
<td>4</td>
<td>Input part time</td>
<td>21</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table III. Evaluation for the example systems HRS and AMIS with Chidamber and Kermerer metrics [41].

<table>
<thead>
<tr>
<th>Difference between refactored and original (FBM-OOP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>WMC</td>
</tr>
<tr>
<td>LCOM</td>
</tr>
<tr>
<td>CBO</td>
</tr>
</tbody>
</table>

According to Tsang et al. [40], AOP does not change the WMC value so much because advice blocks take the place of removed methods. In our work, the WMC is decreased by 0.63 in HRS code and 0.60 in AMIS code. The complexity in terms of WMC decreased while the total number of methods remained unchanged because methods were just repositioned into newly added aspects. Some aspects are added while the number of removed methods and added advice blocks are almost the same. The second metric, LCOM, is the degree to which methods within a class are cohesive (the percentage of references on irrelevant data per total references: a low percentage means a high cohesion). This value is decreased because FBM factors out some irrelevant calls, as AOP factors out aspects from classes. As an example of the aspect to be factored out, logging is an irrelevant functionality that is involved in operations of some other functionalities. The third metric, CBO, is the count of couplings between classes, and a higher CBO value usually means poor design. The result shows that there is no change. This means, in terms of CBO, that there is no improvement by FBM. However, modularization has been improved because multiple method calls scattered across classes have been replaced by a single advice block (note, method calls are placed in multiple classes but links are gathered in one advice block).

In the redesigned part of AMIS, the ripple effects caused by a modification are unidirectional, but it lost the modularization for classes because the classes are scattered across multiple features. This may be a disadvantage from the view of traditional OOP, but it is expected that during the maintenance phase maintainers are able to modify more efficiently focusing only on the parts that they are concerned with. Therefore, although FBM is not validated quantitatively, FBM can improve the maintainability without losing modularity.
5. DISCUSSION AND CONCLUSION

We propose an approach to software construction by aligning dependencies unidirectionally for improving the maintainability of a software system. In our approach, FBM supports the properties of traceability and unidirectional dependencies so as to lead to safer maintenance by confining novice maintenance activities within a limited working area.

It is not easy to show the property of unidirectional dependency. However, a similar property has been explained by Martin [13]: a system is composed of stable parts and unstable parts. More abstracted languages such as AspectJ enable us to design and implement such properties, and the aligned features in FBM extend the property to facilitate the leveled maintenance.

As a basic requirement for maintaining trustworthiness, the re-factored part of AMIS must guarantee consistency among features. Since a feature is a set of partial class definitions and is added to other features in the manner of stepwise refinement [20], the addition must guarantee the consistency of all features. Liscov and Wing [42] provided notions of strong consistency (refinements are totally independent of each other) and weak consistency (refinements are dependent only on one direction). It can be inferred from the work of Colyer et al. [43] that weak consistency is achieved in AOP in that aspects are dependent on the base program but not vice versa. We conclude that FBM also ensures weak consistency because FBM is based on AOP: FBM does not allow a method to be overwritten by newly added features and requires complete analysis of read and write dependency relations among the features.

Leveled maintenance is consistent under the assumptions that functionalities are well aligned and features are properly implemented reflecting the functionalities. In the example of the HRS, suppose a part of the users’ requirements is changed and F2 in Figure 18 needs to be modified for the change. Any modification on F2 by maintainers must not affect F1 and F3. The AOP language mechanism guarantees that there is no effect in code. Moreover any effect on program behaviors can be avoided by prohibiting maintainers from modifying other more essential features. This is possible because all features are the reflection of the aligned functionalities; thus there is no reason to modify more essential features when modifying less essential features. The pointers can be ‘one triggers another’ or ‘shares’ as illustrated in the rules, and are implemented by the pointcut of AspectJ. Either way, any modification to the less essential feature cannot overwrite the more essential feature: when we try to add a method in an aspect that exists in the base class, the AspectJ compiler creates a compilation error. As a result, the software built by FBM provides greater security against maintainers’ modifications.

For future work, we plan to apply FBM in a software product line in which capturing commonalities and variations of software are major concerns. We believe that features with higher essentiality will become common assets while accidental features become variations. We also plan to revise the inertia metric to reflect maintenance frequencies and anticipation to change systematically.

REFERENCES