An Approach to Feature Based Modelling by Dependency Alignment for the Maintenance of the Trustworthy System *

Jungyoon Kim, Doo Hwan Bae
Division of Computer Science,
Korea Advanced Institute of Science and Technology, Daejon 305-701, Korea
{jkim, bae}@se.kaist.ac.kr

Abstract

Functionalties in a software system can be categorized as accidental or essential, where accidental functionalities are dependent on essential ones, but not vice versa. Distinguishing essential functionalities from accidental ones in the development can lead to the reduction of maintenance effort and to the trustworthy system. In this paper, we propose Feature Based Modelling (FBM) in which accidental functionalities are aligned with essential functionalities. The feature in this approach, as a logical unit, provides traceability from a functionality in the requirements to an implemented module with effective dependency management, thus the accidental features can be modified with minimal disruption to the essential features. FBM enables us to limit working boundary of maintenance activities, thus helps to keep the trustworthiness of the system against unskilful modification.

1. Introduction

Software maintenance is a significant part in the life cycle of large software systems [1]. However, it can be inferred that good maintainability of software comes mainly from the development, not from the maintenance itself[1, 2]. In this paper, we propose Feature Based Modelling (FBM) to support development process for effective maintenance and trustworthiness of the developed system.

Functionalties of a domain can be categorized as essential or accidental as in Figure 1, and accidental functionalities are usually dependent on essential functionalities, but not vice versa. The fundamental and strategic activities of a domain can be considered essential, and the subsidiary activities accidental. Martin characterized the essential as stable and accidental as unstable[3]. That means, accidental functionalities are frequently changing while essential ones are hardly changing. As Graham[4] also pointed, essential objects are stable in time and space. For example, in the telephone calling system [5], the calling operations are the basic operation of the system while the billing operations are dependent on the calling because the billing amount is calculated based on the calling records. The billing does not affect to the calling. The calling rate is more frequently changing than the basic calling operation. We call such dependency direction from accidental functionality toward essential functionality as unidirectional dependency.

If it is possible to map the functionalities to the implemented modules keeping the above unidirectional dependency, each module can be maintained by the different level of expertise or effort, in that a novice takes accidental modules and an expert takes essential modules. We call this levelled maintenance and is illustrated in Figure 2.

Levelled maintenance is motivated by military software systems [6] which are characterized as large scale, frequently changing, and trustworthy. Users (novices) in such domain are expected to modify frequently chang-
ing modules by themselves without the original developers' help. In the systems characterized above, it is common that the users do not have enough knowledge and skills to modify the software safely.

Also, such systems must be developed to be trustworthy. In addition, its trustworthiness must be maintained against modification during the maintenance phase. Berstein[16] listed six constraints that keep a system trustworthy. The constraints imply that human capability is limited and system defects are correlated with personnel practice. Thus, it is recommended to limit people's working boundary (i.e., module size) and their usage of language features. This limitation can be applied to the maintenance phase to keep the trustworthiness. Once such systems are delivered to the users (military), it is required to be maintained by the users without original developers' involvement. The users' maintenance activities to the system must not cause any severe problems. FBM provides a way to handle this situation by confining users' modification within limited part of a system: users (novice maintainers) can modify only the accidental parts and are prohibited to modify the essential parts.

Levelling maintenance, however, is not feasible when functionalities are scattered over the modules and the modules have mutual dependencies. A functionality implemented on scattered modules loses traceability [7, 9], and mutually dependent modules cannot keep unidirectional dependency property. Consequently, even if a user modifies an accidental module, it may affect an essential module. Implementation supports to overcome these obstacles are partially possible through language constraints such as AspectJ, but modelling support is still immature. The goal of our work is to provide modelling support for levelling maintenance.

Feature is a logical unit in FBM. It is traceable from the requirement to the corresponding implementation and can be aligned in unidirectional dependency through effective dependency management. FBM has three phases: 1) identify essential and accidental functionalities during the requirements analysis, 2) extract aligned features from Jacobson's use case[8] scenarios, 3) categorize each feature into essential or accidental. FBM is not a complete process, but a supplement to conventional development processes, such as Rational Unified Process [10].

This paper is organized as follows: Section 2 presents the brief summary of related work, Section 3 presents the rationale of FBM, and Section 4 explains FBM in detail with its concepts and phases. Section 5 summarizes our work and suggests future works. We omit detailed explanations for the space limit, but further information (extended explanations with case studies) can be found at "http://www.salmosa.kaist.ac.kr/~jkim".

2. Related Work

FBM, based on separation of concern[11], succeeds the properties of Aspect Oriented Software Development (AOSD) to the modelling stage. The purpose of FBM is to complement traditional software development process such as Use Case Driven Approach [8]. This section briefly explains about AOSD that provides implementation level mechanisms for levelled maintenance, and Use Case Driven Approach that is a major traditional software development process.

Aspect Oriented Software Development: The existing AOSD work focuses on the implementation issues including language semantics[7, 9, 12] and weaving mechanisms[5, 13, 14]. However, few have focused on activities of early software development stages, such as requirement analysis and design. Although the levelled maintenance is feasible with the implementation mechanisms of Aspect Oriented Programming (AOP), we still need a modelling technique that utilizes the benefits fully throughout the software life cycle.

The key properties of AOP are 1) traceability, 2) unidirectional dependency, and 3) dependency inversion. First, the traceability of AOP can complement the shortcomings of traditional object oriented models that have poor traceability due to crosscutting concerns across classes [9]. In the object oriented models, a single requirement is implemented in multiple classes that participate in a single collaboration. On the contrary, in AOP models, a single collaboration can be implemented in a single aspect. Second, aspects in AOP models can be plugged in or unplugged because of the unidirectional structure[5, 14]. Aspects are dependent on the base object oriented model, but not vice versa, thus aspects over the base model can be removed without any disruption to the base model. Third, AOP inverts the direction of dependency in traditional object oriented models. Dependency inversion hides detailed dependencies in model, that are irrelevant for humans to manage. Raising abstraction level may bring about meaningfulness of a program [18], but AspectJ reverts dependency without losing its semantics [5, 13].

Use Case Driven Approach: Use Case Driven Approach [8] has been successful by enhancing traceability, and its practicality has been accepted by industry. Consequently, it incorporated into Rational Unified Process (RUP) [10]. Since use cases are the functionalities of a system and guide entire development process, the approach may enhance traceability to some extent from requirements to implementation. However, its traceability is not so feasible because the use case driven approach is based on object oriented paradigm. The functionalities extracted during requirement analysis phase will be scattered across system elements(classes) during design and implementation
phases. Use case model decomposes a system by requirements (functionalities) while classes decompose a system by structural view, thus Use Case Driven Approaches cannot keep the track between a requirement and a structural module.

Figure 3 summarizes Use Case Driven Approach. Development process starts with identifying use cases in which each use case can be mapped to a single functionality in requirements. The use case scenarios will be specified by collaboration diagrams (arrow 1), and static structure of program is extracted from the collaboration diagram (arrow 2). The static structure can be transformed to code by tools (arrow 3). One collaboration diagram may represent a single use case, but static structure do not keep the trace of use case. The use cases disappear in static structure and implemented codes.

3. The Rationale

This section explains the requisites and the properties required by trustworthy systems. FBM aims to meet the requisites by providing proper design mechanisms.

Levelled Maintenance:

Trusted systems must be developed to be trustworthy and the trustworthiness must be maintained against modification during the maintenance phase. According to the constraints for trustworthy system [16], it is recommended to limit peoples' working boundary and their usage of language features. This limitation can be applied to the maintenance phase to keep the trustworthiness. Especially in military, users are often expected to modify the system by the necessity of military matters. After such trustworthy systems are delivered to the users, the users’ maintenance activities must not cause any severe problems. This situation requires to confine users’ modification within limited part of the system.

The maintenance policy specified in the DoD Directive[17] categorizes three levels of maintenance as follows:

- **Organizational level**: simple and frequent jobs such as fuelling, replacing simple parts.
- **Intermediate level**: replacing major modules (engine, flight control computers...).
- **Depot level**: major overhaul, investigation of fuselage crack, and modification of aircraft for enhancing or adapting its missions.

This categorization of maintenance activities implies that i) maintenance activities can be divided by their complexities, ii) in each level, different level of the maintenance expertise is required for productivity. In its analogy to levelled maintenance, we can also consider that the depot level is essential and the organization level is accidental.

Figure 4 shows the dependency relations among levels in such maintenance activities. Performing depot level jobs accompanies intermediate and organizational levelled jobs. For example, to check the crack of the central fuselage with X-ray (depot), the engine must be unloaded before the crack check (intermediate), and a number of doors must be open before the engine down (organizational). Even though taking X-ray itself is simple, the whole job is complex because of the engine unloading. \(^1\)

**Traceability and Unidirectional Dependency**: To achieve levelled maintenance, we require two properties, traceability and unidirectional dependency, and one mechanism for dependency management. Figure 5 illustrates the development process for levelled maintenance. In Figure 5 each column represents functionalities in each development phase (domain analysis, requirements analysis, design, and implementation). An oval means a functionality. A horizontal arrow between functionalities represents the traceability: both side arrowheads indicate the back and forth mapping in iterative development. The vertical arrows indicate the unidirectional dependencies: each

---

\(^1\) For maintainer's view, fuselage is the most important part of an aircraft. Life span of an aircraft is determined by the crack growth of its fuselage. On the contrary, fuelling or replacing simple parts are trivial jobs that do not affect critical condition of an aircraft. We know that aircraft cannot fly without fuel, but fuelling is just a trivial job from maintainer’s view.
vertical arrow points to the more essential functionality.

The traceability is the modelling property to map from a functionality in the requirements to a module (an implemented functionality\(^2\)) and vice versa. In levelled maintenance, when a functionality changes, the user must be able to identify the modules affected by the change to the functionality. If the functionality is not traceable across the development phases, the user will fail to find and modify the target modules.

The unidirectional dependency must be kept across the development phases. If there exists any mutual dependency between two functionalities, the accidental functionality cannot be modified without affecting the essential functionality because of ripple effects to the essential functionalities. This is the failure of levelled maintenance.

Dependency management is required to hide detailed dependencies that are irrelevant for human beings to manage. For example in Java implementation level, there might be mutual dependencies. If we can have support from any automation tool, such as a language compiler which manages detailed dependencies, then we do not need to consider that detail and we can revert the dependency direction [13]. It means, without support of proper dependency management, we cannot keep the property of unidirectional dependency in the following development phases.

4. Feature Based Modelling

Feature Based Modelling (FBM) supports levelled maintenance, in that simple and frequent maintenance activities are assigned to users (as novice maintainers) and difficult activities to original developers (as expert maintainers). This modelling technique is not a complete process, but can be used as supplementary process for other conventional development process, such as Rational Unified Process[10]. FBM has the three phases that are summarized in Table 1. Each phase can support RUP's workflows shown in the 5th row of the table. This section briefly explains the key concepts and phases of FBM.

4.1. Key Concepts

This section introduces new concepts that are basis of FBM. They are feature, functionality alignment, feature alignment, and inertia.

4.1.1. Feature

The concept of feature is partially borrowed from the concept of Eisenbarth’s [15].

- **Computational Unit**: an atomic executable part of a computational module (i.e., a method of an object).
- **Scenario**: a use case, a user's requirement that can be represented by UML sequence diagram.
- **Feature**: a sequence of computational units, that is a part of a scenario.\(^3\)

Feature is a behavioral module that reflects a functionality (use case, an aspect in AOP, or a collaboration of objects). A feature maps a functionality to design/implemented module, providing dependency inversion mechanism. Figure 6 illustrates the feature examples. Each shaded part in the two

\(^2\) Practically, the relation from modules to functionalities can be regarded as "total subjective function" in logic(modules \(\rightarrow\) functionalities).

\(^3\) Usually, the feature invoked by a scenario is functional, and the feature shared(triggered) by multiple scenarios are non-functional that are not directly mapped to a user's requirement.
scenarios represents the shared part (Feature 3 in the figure).

4.1.2. Functionality Alignment This is the manual activity during requirement analysis for preliminary requirements categorization into essential and accidental. In this phase, the inquiry form as shown in Table 2 will be handed out to the users who submitted the requirements. Each cell in the left column is the title of each requirement. The users list requirements in the right column if those are dependent on the requirement in the left column. The result will be represented in a directed graph in Figure 7 (a). Some requirements will be merged until there is no cyclic (including mutual) dependency. The final result will show an acyclic directed graph as in Figure 7 (b), then each oval in (b) becomes a functionality ordered from essential to essential. In Figure 7 (b), we say that Req 6 is more essential than Req 3,5, and Req 3,5 is more essential than Req 1,2,4. This dependency relationship will be included into the original requirements in the form of use case scenarios.

4.1.3. Feature Alignment: Feature alignment aligns the features by alignment rule that decides the dependency direction among features. We have devised the four rules as listed in Figure 8:

- **Rule 1: Mutual Dependency**: Feature 1 calls Feature 2, and vice versa. Both become one feature.
- **Rule 2: Read-Write Dependency**: Both features share a system state, such as database, and Feature 1 can read and write, but Feature 2 has read only. Feature 2 is dependent on Feature 1.

Users describe the dependencies in Rules 1 and 2 at Phase 1 in Table 1. The cases of Rule 2 can be analyzed during the requirements analysis. If any two features are aligned by this relation, the two will be used as the basis for other features' comparisons.

- **Rule 3: Uni-Directional Dependency**: Only Feature 1 calls Feature 2, not vice versa. Feature 2 does not return any value. The dependency direction can be inverted if the Feature 1 is more essential than Feature 2: "more essential" means, for instance, Feature 1 is originated from Req 6 while Feature 2 is originated from Req 3,5 in Figure 7 (b). The Dependency inversion mechanism of AspectU [5] is shown in Figure 9 (a). As another example in the case of a use case inclusion or extension, the base use case is dependent on included use case while extended use case is dependent on base use case. If such dependencies are handed over to design model, the encapsulated base use case must call both included and extended use cases. That means the base use case is dependent on the included and extended use cases. To keep the unidirectional dependency, we have to invert the dependencies in that the included and extended use cases are dependent on the base use case.

---

4 Base use case: a use case that includes other included use cases or extends to other extended use cases.
- **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. At first, unit sharing does not indicate dependency direction, but this association will be replaced by a dependency arrow when one feature is selected to have the unit implementation. This is shown in Figure 9 (b). The shared units can be methods or attributes of a class. In Figure 9 (b), shared unit \( m \) can be implemented in either one of features and the other feature will have a proxy of \( m \). The implementation code of \( m \) will be placed in the feature that is more essential.

These rules may not be complete. There could be write-write, or read-read dependencies, but write-write dependency can be treated as similar to the case of Rule 1, and read-read can be ignored because they do not affect each other. The rule set can be extended, or some rules can be replaced (or changed) according to the domain context.

### 4.1.4. Feature Dependency Graph
This graph shows the dependency relationship between features in a system. The graph is composed of ovals similar to the UML use case, and arrows that indicate the dependency direction. A feature from which an arrow starts is dependent on another feature to which the arrowhead points, as illustrated in Figure 10.

### 4.1.5. Inertia
Any pair of features without any reachable relationship on the feature dependency graph cannot be compared. Thus, we need a weighted graph with the proper measurement to compare such incomparable features. The new metric, inertia\(^6\), is based on Abbott’s *inertia*\(^5\), and Martin’s *stability*(\(St\))\(^3\). We use the term *essentiality* to represent how a feature is essential or accidental in the relations of the unidirectional dependency. The inertia is a metric to indicate the essentiality.

Inertia is the *Partial order of Internal and External Complexities*. The complexity (\(C\)) of a feature is defined as, by composing the feature’s *IS* as internal complexity and *St* as external complexity:

\[
C = IS \times (St + k)
\]

where the constant \(k\) is set to 1 for convenience (Note that *St* is normalized value). *IS* and *St* are heuristically used to indicate the partial order of features placing weight on *IS* (giving priority on interface size). Inertia(\(In\)) reflects external dependency including not only the directly neighboring features, but also the indirectly related one (unlike *St* that only reflects the direct one). The inertia \(In_e\) of the feature \(e\) in a feature dependency graph is

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

where \(D_e\) is the set of features dependent on the feature \(e\), \(C_i\) is the complexity of a feature \(i\) in \(D_e\). For example in Figure 10, the \(In_e\) is the sum of \(C_i\)s of the shaded features, \(a, b, c, d\), and \(e\).\(^6\) Inertia reflects the (lattice) order among features on a feature dependency graph. In other words, any feature \(A\) dependent on a feature \(B\) cannot have bigger \(In\) than that of \(B\). Inertia will be used for feature comparisons during Feature Extraction process and Level Categorization in Table 1.

### 4.2. FBM Phases
This section further explains the three FBM phases in Table 1.

\(^5\) The term inertia in the context of software maintenance usually means the resistance to change by users\(^1\). By contrast, the inertia in our work reflects the characteristic of software model, including complexity of a unit and ripple effects of changes to a unit.

\(^6\) This concept of the inertia may be closer to the concept of moment of inertia 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.
4.2.1. Phase 1: Functionality Alignment In functionality alignment, users’ involvement is indispensable, because the essentiality of the functionalities can only be found with help of users’ knowledge.

Figure 11 shows an example related to use case, as explained by Rule 3 in the section 4.1.3. In Figure 11, requirement 1 is the base use case, requirement 2 is the included use case, and requirement 3 is the extended use case. By Rule 3, requirement 1 is dependent on requirement 2 and this dependency will be reverted. On the contrary, requirement 3 is dependent on requirement 1 and this will be kept.

4.2.2. Phase 2: Feature Extraction Extraction Process extracts a feature dependency graph based on the output of functionality alignment in Figure 7. The process has procedure\(^7\) that can be automated. The input of the process is a set of use case scenarios and the output is a weighted dependency graph with calculated inertia. The output will be used for categorizing features for leveled maintenance.

Figures 12 and 13 illustrate Phase 2. Figure 12 (a) shows the extracted features from use cases in Figure 11 assuming that requirement 3 mapped to two features (the feature 3.1 and 3.2) and requirement 1 and 2 mapped to feature 1 and 2, respectively. The computational units are preliminarily identified as shown in Figure 12 (b) from the use case scenarios, then the inertia values of features are calculated. Let us assume that both features 3.1 and 3.2 have the same \( IS \). The inertia values will be updated at each dependency decision by Rule 4. Finally the feature dependency graph is extracted as shown in Figure 12 (c). The feature 3.2 has been dependent on 3.1 because 3.1 has bigger inertia (same \( IS \) but bigger \( St \): if a feature has more features that are dependent on it, it has bigger stability).

Figure 13 shows a more refined description of the feature dependency graph. The classes and sequence diagrams are arbitrary examples for simple explanation. A UML package in Figure 13 represents a feature in Figure 12 (c). In Figure 13, the dependency relationships are described by dotted line arrows that may be annotated with UML comments. The arrows explain that the class from which arrow starts must be implemented in the class at which the arrowhead points. For example, the class A in feature 1 is shared by feature 3.1 and feature 2, thus it tells that the class A will be implemented in the requirement 2. The arrows between the sequence diagrams explain that the message from which the arrow starts will be triggered by the message at which the arrowhead points.

4.2.3. Phase 3: Level Categorization Inertia reflects the order of the feature’s complexity. Thus the features in feature dependency graph can be categorized into a several levels according to the order specified by the inertia values. Note that the inertia value is a heuristic guideline.

Figure 14 (a) illustrates the example of the implementation. Each feature has number of classes and the classes in a single feature are partial definitions. Those partial definitions will be composed into Figure 14(b) by automated tools such as AspectJ or HyperJ. Level categorization phase enables the levelled maintenance in that a novice can work only on upper features in 14 (a), while an expert can work on all. This confinement according to the level of maintainer’s skill is required to keep the trustworthiness of a sys-

---

\(^7\) The pseudo code of the algorithm can be found at “www.salmoa.kaist.ac.kr/~jkim.”
5. Conclusion and Future Works

We introduced FBM for levelled maintenance of the trustworthy system, that requires modelling properties including traceability, unidirectional dependency, and effective dependency management. The feature in FBM makes such requisites feasible. As a result, we can effectively assign maintenance effort for a feature according to its essentiality level. Feature Based Modelling contributes to a better allocation of human resource during the design and implementation phases, in that users (novices) take charge of accidental features and developers (experts) take essential features. This confinement of working boundary helps to keep a system trustworthy.

The ongoing research includes the following works. Tool development: a tool that extracts the feature dependency graph from use case scenarios and calculates the inertia. Investigation for the applications for large system: The small example in this paper must scale to real life software systems. Metric validation: We need to verify that inertia has empirical usefulness through experiments.

References