SYSTEMATIC FUNCTIONAL DECOMPOSITION IN
A PRODUCT LINE USING ASPECT-ORIENTED
SOFTWARE DEVELOPMENT: A CASE STUDY*

TEGEGNE MAREW*, JUNGYOON KIM† and DOO HWAN BAE‡

Division of Computer Science,
Korea Advanced Institute of Science and Technology,
Daejon 305-701, South Korea

*tegegnew@se.kaist.ac.kr
†jkim@se.kaist.ac.kr
‡bae@se.kaist.ac.kr

Systematic configuration management is important for successful software product lines. We can use aspect-oriented software development to decompose software product lines based on features that can ease configuration management. In this paper, we present a military maintenance product line that employs such strategy. In particular, we applied a specific approach, feature based modeling (FBM), in the construction of the system. We have extended FBM to address properties specific to product line. We will discuss the advantages of FBM when applied to product lines. Such gains include the functional decomposition of the system along user requirements (features) as aspects. Moreover, those features exhibit unidirectional dependency (i.e. among any two features, at most one depend on another) that enables developers to analyze the effect of any modification they may make on any feature. In addition, any variations can be captured as aspects which can also be incorporated easily into the core asset if such variation is deemed to be important enough to be included in the product line for further evolution.

Keywords: Product line; aspect-oriented software engineering.

1. Introduction

Software product line has been recognized as reuse intensive software development methodology. It relies on capturing potentially reusable behavior as core assets and product specific ones as variations. Since software product lines heavily rely on software decomposition based on user visible functions (features) [1], the approach used to realize software product lines should possess such property. One candidate is aspect-oriented software engineering [2]. Aspects which were originally created for capturing “cross-cutting concerns” can also be used to capture basic functionality

*This work was supported by the Ministry of Information & Communication, Korea, under the Information Technology Research Center (ITRC) Support Program.
34 T. Marew, J. Kim & D. H. Bae

[3]. Such approach allows software product lines to profit from the advantages of aspect oriented software engineering.

This paper presents a case study where we applied an aspect-oriented software development approach to realize a product line for a maintenance software system for the military. The aspect-oriented approach which we used in the construction of the product line is called Feature Based Modelling (FBM) [4]. It was originally developed to enhance the trustworthiness of a software system by unidirectionally aligning features.

We have extended FBM, which was created originally to address development of single systems, so that it can handle specific features of product line. As, dependency management is at the heart of developing an easily evolvable product line, the extended FBM concerns dependencies between various core assets as well as variable features.

Among the many characteristics of software product lines, their ability to systematically manage the evolution of a specific product as well as the entire product line (configuration management) has paramount importance. The challenges of configuration management are numerous and very difficult to address. In this paper we discuss only two such challenges. The first is how do we know what part of the software to change (add, delete, or modify) to realize the evolution we want. The second is how do we know the effect of the change on the rest of the system. We will show how FBM is used to address these two specific challenges of software product lines.

This paper is organized as follows. Section 2 is devoted to the detailed discussion behind the motivation for this paper. In Sec. 6, we present the related work specific to configuration management of software product lines. FBM, the approach we used in the construction of the military maintenance system is discussed in Sec. 3. Then, the military system itself is presented in Sec. 5 where we also describe the application of FBM to the construction of the system. Section 8 concludes the paper by summarizing the main points and pointing possible future work.

2. Motivation

As described in the previous section, in this paper, we present a case study where FBM is applied to the construction of a product line for a maintenance software system for the military. The motivation behind choosing FBM, an aspect-oriented approach, is mainly due to the advantage of AOP over OOP [5]. Specifically, we believe that the ease to achieve transparency through features using AOP and the ability of FBM to manage dependencies among different software modules make it a better approach. These properties play a significant role in reducing the complexity in configuration management of a software product line.

Aspect oriented software engineering is originally introduced to complement OO programming by allowing the developer to dynamically modify the static OO model to create a system that can grow to meet new requirements. Often, AOP
is used to capture non-functional requirements as they are usually “cross cutting concerns”. As a result, many case studies on applying AOP in product lines use AOP only to capture the non-functional requirements and variations in specific products [6–8]. However, classes are not ideal to capture functional requirements entirely as features of the software in a modular way [7]. Aspects, on the other hand, have been suggested as a good candidate for capturing such features [9, 10].

In software product lines, usually there is no specific structure between the modules of the core assets at implementation level. If we modify one module, it is not easy to find what and how that change affects the rest of the system. Some authors [15] have suggested to keep a list of required and affected modules along with each core asset which easily does not show the module/s that may have wide-reaching effect if modified. Such knowledge is important for a single product development as well as family of products. What we need is an approach that explicitly captures the dependencies among different modules. FBM, to be explained in Sec. 4, has such a property for improving the trustworthiness of a single product. In fact, this case study is conducted to see the application of FBM for a product line.

FBM has shown promising results as a complementary approach to other established software methodologies in the development of single products [4]. We believe three related characteristics of FBM makes it a good candidate for an approach to realize software product lines. These are:

- The ability to capture user requirements as features and realizing them as aspects. This will enable us to know what software module to modify in case of requirement changes either in the core assets or variations.
- The ability to create a dependency graph between features. The graph readily answers the question “Which modules or features are affected by a specific modification?”.
- The ability to create a lattice among features thus enabling unidirectional dependencies. We can easily know not only which modules affect which but also which ones have a wide ranging effect and thus systematically control the ripple effect caused by modification.

All these advantages become very handy during not only the configuration management of product lines but even during the construction of the software product lines. By mapping each user requirement to features realized as aspects, developers can enjoy better transparency during the different phases of development.

3. Feature Based Modelling

In large critical software system, the trustworthiness must be preserved during maintenance. Berstein [11] listed constraints (reliability, security, privacy, business integrity . . . ) that keep a system trustworthy. He points out that the human capability is limited and system defects are correlated with the personnel practice. To alleviate such a problem, it is recommended to limit peoples’ working boundary
(i.e., module size) and their usage of language features. In many cases, users are needed to maintain such software by themselves, because they cannot always afford to hire skilled experts (or original developers) for frequent or tedious maintenance. Moreover, some military systems need on-field modification for certain situations while most businesses are under a “time to market” pressure. The problem arises when the systems are maintained by users, unskilled maintainers, who might deteriorate the trustworthiness of the system. Such deterioration can be controlled by dependency alignment. It enables to confine the ripple effects caused by the users’ modifications within limited working boundaries. Such maintenance approach can be called “levelled maintenance”.

We devised Feature Based Modelling (FBM) that helps to apply levelled maintenance. FBM extracts features from use cases and aligns them in one direction. Since the features reflect functionalities in users-requirements, users can easily map the functionalities to the features in design and code. Some features of an application must be maintained by users without the help of software experts and still those maintenance activities must keep the trustworthiness of the software. FBM keeps the features in design and code, and with unidirectional dependencies, it also makes the levelled maintenance feasible. This modelling technique can be used as a supplementary for other conventional development processes, such as Rational Unified Process [13].

FBM process consists of the following four phases:

- **Functionality Alignment**: In functionality alignment, users involvement is indispensable, in that the dependency relationships among functionalities can be found only by systematic and iterative process with the knowledge of domain expert.

  The inquiry form as shown in Table 1 will be handed out to the domain experts. The requirements in the form are short sentences excerpted from original user problem statements. The left column lists requirements titles by one title in a row. Domain experts check requirements in the right column whether they are dependent on the requirement in the left column. Domain experts are asked a
simple question such as “If a requirement of the left column is removed (or modified), then what requirements in the right column will be affected?” Based on the inquiry results, the dependencies among all requirements of a system are determined. Such identification should also be based on factors like anticipation for change and change frequency. The result of the inquiry form can be represented in a directed graph.

- **Identify Features**: Features are extracted from the functionalities identified in the previous phase. Often the functionalities become feature themselves. In cases where functionalities share a sequence of activities (which is the result of analyzing their respective sequence diagrams), such shared activities become features themselves.

  The concept of a feature is partially borrowed from that of Eisenbarth’s [12]. 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 UML sequence diagram).
  - **Feature**: a sequence of computational units that can be triggered by multiple scenarios. A more formal definition can be found in [16].

- **Identify Shared Units**: This phase identifies any unit shared by more than one feature. The shared unit will be implemented in only one feature among all features that share the unit. For a simple example, assume two features sharing one unit. At first, this unit sharing does not indicate any dependency direction, but this association will be replaced by a dependency arrow when one feature is selected to have the unit implementation. The selection is decided through consideration for essentiality of the two features. An example of a shared unit is given in Fig. 9.

- **Decide dependencies among features**: This phase decides the dependency direction among features based on the result of the previous phases. Decisions are made according to the rules shown in Fig. 1. 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. For detailed information on FBM, please check the home page at “http://www.salmosa.kaist.ac.kr/~jkim”.

- **Implementation**: Since at the implementation level there are many dependencies to consider, we need to abstract them for alignment. Note that raising the abstraction level may lose the meaning of the original program. The abstraction
must not lose indispensable dependency information. As one of the tools that support such abstraction, AOP reverts directions of dependencies or removes dependencies (low level implementation) without any semantic changes. A set of aspects are used to implement a feature and the class they advise acts as a control object in the realization of the feature as shown in Fig. 8.

4. Extending FBM for Product Line

In Ref. 14, the authors identified a number of dependencies that could exist between features (as used in FODA [1]) in the product line. They went further than just structural relationships (e.g., aggregation and generalization) and configuration dependencies (e.g., required and excluded). The additional dependencies they identified are:

- **Usage Dependency**: A feature may depend on other features for its correct functioning or implementation. Usage can be further divided into
  - OR Dependency: A feature uses either one of two or more features.
  - Alternative Dependency: A feature uses exactly one of two or more features. This could be seen as an XOR dependency.
  - Optional Dependency: A feature may not use the other one. Here if the two features have this relationship, the actual product may not have the second feature.

- **Modification Dependency**: A feature (modifier) modifies another feature (modifyee). This occurs only when the modifyee is in activation (as explained below). Therefore, Modification dependency between two features means that the behavior of a modifyee may be modified by a modifier, while it is in activation.
Table 2. Feature alignment rules for product line.

<table>
<thead>
<tr>
<th>Feature Dependency</th>
<th>Feature Alignment Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>X has OR Usage Dependency with Y</td>
<td>X depends on Y</td>
</tr>
<tr>
<td>X has XOR Usage Dependency with Y</td>
<td>X depends on Y</td>
</tr>
<tr>
<td>X has Optional Dependency with Y</td>
<td>X does not depend on Y</td>
</tr>
<tr>
<td>X has Modification Dependency with Y</td>
<td>X depends on Y</td>
</tr>
<tr>
<td>X has EA Dependency with Y</td>
<td>X does not depend on Y</td>
</tr>
<tr>
<td>X has SA Dependency with Y</td>
<td>X depends on Y</td>
</tr>
<tr>
<td>X has CA Dependency with Y</td>
<td>X depends on Y</td>
</tr>
<tr>
<td>X has SeA Dependency with Y</td>
<td>X depends on Y</td>
</tr>
</tbody>
</table>

- Activation Dependency: A feature must be active before it can provide its functionality to users. An activation of a feature may depend on that of other feature. Activation dependency can be classified into four categories as
  
  — **Exclusive-Activation Dependency (EA):** Two features cannot be active at the same time.
  
  — **Subordinate-Activation Dependency (SA):** There may be a feature that can be active only while the other feature is active.
  
  — **Concurrent-Activation Dependency (CA):** Some subordinators of a superior may have to be active concurrently with each other while the superior is active.
  
  — **Sequential-Activation dependency (SeA):** Some subordinators of a superior may have to be active in sequence.

We can easily reason that the above dependencies are not exclusive to Product Line. In fact, some of the rules we identified in Fig. 1 have some similarities with the above dependencies. Nonetheless, we decided to extend the rules of Fig. 1 by deciding how such dependencies in product line translate into feature alignment in FBM. The extended feature alignment rules are shown in Table 2.

The primary reason for keeping the potentially similar rules of the original as well as the extended FBM is the fact that the new rules are more specific to Product Line. For instance, Rule 3 in the original FBM and Usage Dependency has similar property. Yet, Rule 3 does not distinguish between the different kinds of Usage Dependencies because it does not have to. Therefore, we need all the rules, the original as well as the extended, to clearly describe the different dependencies in a Product Line: mainly the original rules can be used for specific product development while the new rules can be used during core asset development.

We did not include alignment rules for aggregation and inheritance because those are structural dependencies which are addressed differently in FBM. FBM uses aspects created from (a portion of) sequence diagrams. Therefore, features in FBM are more like collaborations of classes and thus such structural dependencies like aggregation do not apply directly to them. We also do not have rules for
configuration dependencies such as required and excluded because those specific configuration dependencies are coarse-grained compared to the ones we have developed. For instance, OR Usage Dependency can be considered one type of required dependency and Exclusive-Activation Dependency clearly is a more fine-grained exclusion dependency.

Still, it is possible to think of other dependencies between features of product line. We believe it would not be difficult to develop a feature alignment rule for such new dependency.

5. Case Study

The product line we are developing is a product line for a maintenance software system of a military. The overview of the system is shown in Fig. 2. Maintenance activities are generalized and grouped into seven major functionalities like tactical support and management data collection. Also each maintenance activity can be categorized into three levels according to the difficulties (organizational, field, and depot). The specific products are maintenance software system for eight departments such as aircraft and communication.

During the design of the product line, the first step is to identify the functionalities in the core asset. These are:

- **Maintenance activities**: are actual maintenance on equipment that might be troubleshooting (T), dissemble (D), assemble (A), removal from site (R), component removal (C), and component replacement (K).
Table 3. Categorization for levelled maintenance.

<table>
<thead>
<tr>
<th>Product</th>
<th>Maintenance Level</th>
<th>Activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aircraft</td>
<td>Organizational</td>
<td>T,R</td>
</tr>
<tr>
<td></td>
<td>Field</td>
<td>T,R,D,C,K,A</td>
</tr>
<tr>
<td></td>
<td>Depot</td>
<td>T,D,C,K,A</td>
</tr>
<tr>
<td>Ship</td>
<td>Organizational</td>
<td>T,R</td>
</tr>
<tr>
<td></td>
<td>Field</td>
<td>T,R,D,C,K,A</td>
</tr>
<tr>
<td></td>
<td>Depot</td>
<td>T,D,C,K,A</td>
</tr>
<tr>
<td>Fire Arms</td>
<td>Organizational</td>
<td>T,R,D,C,K,A</td>
</tr>
<tr>
<td></td>
<td>Field</td>
<td>T,R,D,C,K,A</td>
</tr>
<tr>
<td></td>
<td>Depot</td>
<td>Does not need</td>
</tr>
<tr>
<td>Missile</td>
<td>Organizational</td>
<td>T,R,K</td>
</tr>
<tr>
<td></td>
<td>Field</td>
<td>T,R,D,C,K,A</td>
</tr>
<tr>
<td></td>
<td>Depot</td>
<td>T,D,C,K,A</td>
</tr>
</tbody>
</table>

- **Maintenance plan:** plans maintenance activities for each equipment.
- **Technical standard:** provides standard for maintenance. Technical orders or manuals that describe procedures for maintenance activities.
- **Personnel management:** manages data of personnel such as maintenance skill level, licenses for specialties.
- **Tactical support:** provides maintenance data to tactical department such as current status of resources that can be required for combat situation.
- **Maintenance data collection/analysis:** gathers data of equipment during maintenance activities such as operation time, fault occurrence frequencies and analyze to produce data such as mean time between failure (MTBF).
- **Management data collection/analysis:** gathers data of management such as man hour.

From now on we will follow the steps of FBM (augmented with the extension ideas) to develop the product line.

### 5.1. Functionality alignment

We need to understand what features are needed by each application which can be used as a basis for differentiating core asset features from those of variations. Such knowledge is captured in Table 3. To understand the table, we use the letter codes introduced in the previous list (e.g. T refers to trouble shooting). From the table, we can understand the only maintenance activities that could take place in relation to aircraft at organizational maintenance level are trouble shooting, and removal from site whereas at depot level there is no removal from site but there is disassembly, assembly, removal of component and replacement of component. Ship and aircraft have virtually the same maintenance activities at each level whereas firearms do not need any maintenance activity at depot level since these are mostly small relatively
simple equipment that could be handled by personnel at organizational and field level.

After use case analysis of the functionalities, we have the use case diagram for the core assets as shown in Fig. 3. As we see, there are three actors: the planner who takes care of the scheduling and the assignment of maintenance jobs, the technician who actually does the maintaining and finally the administrator who takes care of the personnel management. We have identified the functionality Personnel Management should be decomposed into two use cases: Login and Registration, because users need log in each time they use the system but need only to register once. Moreover almost all use cases need Login so it is included virtually by every use case. Since the planner needs to have information regarding the standards (both technical and tactical) before planning a specific maintenance activity, we decided the maintenance plan to include both use cases related to standards. Finally, the management data collection extends the maintenance data collection since the data collected and analyzed by the management is a special case of total data collected by the maintenance data collection.

If we were developing a single application using FBM, the use case would be enough for extracting features. However, we are developing a product line. Therefore, we need to distinguish the features in the product line. Here the features are required for FBM because we have identified new feature alignment rules for product line features. The features of the product line are given in Fig. 4. We have almost identical features in this diagram as use cases in Fig. 3. This is not necessarily
always true though. Moreover Fig. 4 has additional information. First we have the XOR Usage Dependency between standalone and client/server. These are two versions of the same feature to be used in different application architectures. They have XOR (alternative) relationship because if one version is available then the other is not necessary. Second is the OR usage dependency between the features that are available at the three maintenance levels. Their dependency is OR dependency because one or more of them can coexist in the same application.

So far, the activities we carried out enabled at features from the domain expert point of view. The final step in this phase is to align these features. Once we identify the features, we inquire domain experts to give us their thought on the dependency of features. The result is shown in Table 4.

The important observation from this table is only the features in the use case are used. It is because the alternative or the OR features (e.g. assembly, client/server) are always included in the features in the use case. Thus the dependencies between the features in the use case (Fig. 3) also shows dependencies between all the features in the feature diagram (Fig. 4). For instance, the fact that maintenance activity depends on technical standards implies that if we implement a server/client version with only assembly feature as the only maintenance activity, this feature depends on the client/server version of the technical standard feature. Therefore, from now on our discussion only concentrates on use case features.
Table 4. Dependency among features according to domain experts.

<table>
<thead>
<tr>
<th>Requirements</th>
<th>Requirements that are dependent on the requirement in the left column</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registration</td>
<td>none</td>
</tr>
<tr>
<td>Login</td>
<td>Registration</td>
</tr>
<tr>
<td>Technical Standard (TeS)</td>
<td>Log in</td>
</tr>
<tr>
<td>Tactical Standard (TaS)</td>
<td>Log in, TeS, TaS</td>
</tr>
<tr>
<td>Maintenance Plan</td>
<td>Log in, Maintenance Plan, TeS, TaS, Maintenance Activities</td>
</tr>
<tr>
<td>Maintenance Activities</td>
<td>Log in, Maintenance Activities</td>
</tr>
<tr>
<td>Management Data Collection</td>
<td>Registration, Log in, Maintenance Activities</td>
</tr>
<tr>
<td>Maintenance Data Collection</td>
<td>Registration, Log in, Maintenance Activities</td>
</tr>
</tbody>
</table>

5.2. *Identify features*

Now, we are ready to extract features. Unlike approaches like Feature Oriented Domain Analysis (FODA) [1], our approach extracts features that are not necessarily user-visible but are still shared by a number of functionalities that are identified during use-case analysis. Often, use cases themselves end up being complete features but that will be decided after we draw the sequence diagrams of each use case. A portion of use cases (actually, part of their respective sequence diagrams) that occur in more than one sequence diagram is considered as a feature. This is in-line with our intent to use aspects as decomposition units. The original functionalities are still achieved since these shared features (later realized as aspects) are weaved to achieve the original functionality.

In our product line, such phenomenon is observed in the use cases for technical standard and tactical standard whose sequence diagrams are shown in Figs. 5 and 6. The rectangles in both sequence diagrams show the shared sequences among those diagrams. We create a new feature called “selection” which represents the shared sequences. By extracting such shared units, we guarantee modification is consistent
and less time-consuming which are important in configuration management.

We describe such feature sharing as follows. Let S1 and S2 be the two sequence diagrams for technical standard and tactical standard respectively with A, B, C, D, and E describing the classes. When we say A.ma1, we mean the first method in class A. We can now describe the situation as shown in Fig. 7. It shows how two sequence diagrams give rise to three features that are realized as aspects. In the first feature, F1, the aspect F3 is used at the beginning indicating F3 will be weaved into F1 during execution as in Fig. 8 (the \{s:F3\} in F1 indicate that F1 depends on an instance of F3). The same is true for F2. Therefore F1 and F2 are dependent on F3. Since the sequence diagrams of the other use cases do not have such property, no shared feature extraction is performed. F1 and F2 will be the new technical and tactical standard features while F3 is the new feature “selection” mentioned in the previous section.
5.3. Identify shared units

We have reached the phase where we identify shared units. Unlike shared sequences, we are not going to have a new feature. Instead, we select one of the features that share the unit to implement the feature and the other features that share the unit to have the proxy of that unit. During the analysis of the sequence diagrams of maintenance activity, management data collection and maintenance data collection we find such an instance. To perform their functionality, all three features need to specify the scheduled maintenance work they want to work on. As we can see from the features for each use case (Fig. 9), they all have G.mg1 to indicate such selection of a particular scheduled maintenance work (the features are derived from their respective sequence diagrams (not shown) as the previous case). We chose maintenance activity (F1) to implement the G.mg1 while the other F2 and F3 have the proxy of G.mg1. As a result features F2 (for management data collection) and F3 (for maintenance data collection) are dependant on F1 (for maintenance activities).
5.4. **Decide dependencies among features**

Finally we need to sort out all the dependencies based on the previous phases and the rules in Fig. 1. The following are the dependencies among the features of the core asset.

- **LOGIN** feature has a read/write dependency with **REGISTRATION** since login needs the data collected during registration.
- **TECHNICAL STANDARD** as well as **TACTICAL STANDARD** are dependent on **SELECTION** feature as we extracted it from them because it is shared among them.
- **MAINTENANCE PLAN** depends on both **TECHNICAL STANDARD** as well as **TACTICAL STANDARD** since it includes them (in the use case sense of ≪includes≫).
- **MAINTENANCE PLAN** depends on **LOGIN** since Login is included in Maintenance Plan.
- **MAINTENANCE ACTIVITY** has a read/write dependency **MAINTENANCE PLAN** since the latter assigns the activities which the former maintains.
- **MANAGEMENT DATA COLLECTION** and **MAINTENANCE DATA COLLECTION** depends on **MAINTENANCE ACTIVITIES** since they share a unit and we chose **MAINTENANCE ACTIVITY** to implement that unit.
- **MANAGEMENT DATA COLLECTION** depends on **MAINTENANCE DATA COLLECTION** because the former extends the latter (in the use case sense of ≪extends≫).

The unidirectional dependency achieved by applying FBM is shown in Fig. 10. Here we observe that every feature is reachable from one another. Thus, it is possible to achieve unidirectional dependency. However, if two features are not reachable, the original FBM has a notion of inertia to compare the two features. Inertia is a metric that can be used to measure the stability of a feature by taking into account the features that depends on it, the features it depends upon, the methods it has, and the parameters its methods have. More detailed discussion is available at [www.salmosa.kaist.ac.kr/~jkim](http://www.salmosa.kaist.ac.kr/~jkim).

5.5. **Implementation of features**

As discussed in Sec. 5.2, we implement features as aspects. However, care was taken in anticipation of future evolution of the product line. Specifically, we employed the suggestions of [14] on how to weaken the dependency of components. For instance, when we designed almost all features, we have a “stand alone” and a “client/server” version resulting in when a feature A depends on a feature B, we have four versions to keep track (two versions for each feature). Instead of worrying about version-based dependency, we shield each feature as shown in Fig. 11 per [14]’s suggestion. As a result, the technical standard feature being shielded by the Log-in Factory component uses only the Log-in Interface. The Log-In Factory component based on the needs of the Technical Standard feature (whether it is stand
alone or server/client) creates the appropriate Log-in version.

For developing a specific product, we just instantiate the product line features with the specific product preferences. For example the dependency graph for the specific application with the property:

- the equipment is aircraft
- the maintenance level is organizational
the application uses client/server architecture
is shown in Fig. 12.

For instance, when we say Log in[Client/Server], we mean the feature that is a realization of the sequence diagram in Fig. 13, which can be compared with Log in[stand alone] whose sequence diagram is depicted in Fig. 14. Both features are available in the core assets and we choose the appropriate one for the application based on the specific product’s characteristics.
To see how useful our dependency graph for the core assets is, we decided to add new equipment with the following property:

- Fire Control (for controlling fire breakouts): has all the features except it does not need the feature tactical standard.
- Precision measurement: has all the features except the feature management data collection.

By looking at our core asset dependency graph, it is clear that if Fire Control does not need technical standard, then its feature maintenance plan is different.
from the one in the core asset since the one in core asset depends on technical standard. Because of this difference all the features below maintenance plan are also different. Thus, if we want to incorporate this product into our product line, we need to make a substantial modification in the existing core assets. In case of Precision measurement, lack of management data collection feature does not have any effect on the core assets since no feature depends on it. Therefore, including Precision management in the product line is a very straightforward task. Thus, our dependency graph enables us to make a reasoned judgement on whether to include a new member of a product line based on the potential modification we may need to perform on the core assets. The new core assets dependency graph is shown in Fig. 15.

6. Related Work

There have been researches conducted investigating the merit of aspect-orientation in product line development. There are case studies that use aspect-oriented programming in realizing software product lines. As we discuss below, such work mainly uses aspects either to capture non-functional requirements or introduce variations in specific products.

M. Anastasopoulos and D. Nutig [3] have conducted a case study for a hypothetical mobile company that employs aspects in the implementation stage of a mobile phone software product line. They identified a group of criteria to systematically analyze the advantage of aspects. After using those criteria in their case study, they concluded AOP is especially suitable for variability across several components. The same conclusion is reached in [17]. In our work, we contend aspects can also be used effectively during core asset development.

W. Gilani et al. [6] have presented another case study involving aspects in implementing a product line. This time it is for adaptable middleware product lines. They used aspects to capture such middleware behaviors as security, transactions, and naming. These are commonly known as non-functional requirements and aspects are ideal to capture those requirements. Yet, we can also use aspects to capture functionalities as we will show in our case study.

D. Batory et al. [10, 17] has introduced and used feature oriented programming (FOP) in product lines. FOP has greatly simplified (both conceptually and in practice) program construction and understanding. Nonetheless, the notion of feature in their work is a little different from the one we have in FBM in the following ways. A feature in FBM can be:

- a user requirement
- a portion of the sequence diagram shared among sequence diagrams. This feature differs in FBM from the feature in FOP
- a non-functional requirement whereas a feature in FOP is only a functional requirement.
This difference leads to the following distinguishing characteristics between FBM and FOP:

- FBM, through the lattice produced, help us gauge the effect of modification. In FOP, we have to go through each product equation to see if a modification of a feature has impact on other features. Even then, it would be difficult to analyze how severe the impact is on the other features.
- FBM is primarily built to achieve traceability of user requirements among different phases of software development. In FBM, one can easily follow a use case A to analysis of A, design of A, and finally to the feature that implements A. There if a developer wants to modify, test, reuse use case A, all the related artifacts in other phases can easily be traced. Such analysis is not easily accomplished in FOP.

7. Discussion

Our basic objective when we tried to extend FBM for product line development was to create a dependency graph between the features of the product line like that shown in Fig. 16. Our case study shows how adding a product that requires modification of features at lower part of the graph is much easier than adding those that requires modification of features found at the upper part of the graph. This same reasoning can be used during maintenance of the product line where we need to modify an existing feature to accommodate changes that affect the entire product family. Thus our approach is indispensable during cost, time and risk analysis of modifying/evolving a product line.

Our case study revealed that not all the dependencies that could exist between features of a product line exist in every product line. Specifically, the features in our case study only possess the XOR and the OR Usage Dependency. This is not surprising since the other dependencies do not seem that common. For instance, Activation Dependency needs two features who execute depending on whether the other is executing.

Even though we have not used it in our case study, the original FBM has a notion of “Inertia” to address the dependency between features that are not dependent. We did not use it in our case study as all features which we have turned out to be

![Cost, time, and risk of modification](image-url)
dependent on each other. We think it is appropriate to conduct another case study to involve inertia calculations and observe its usefulness.

The primary advantage of our work, as pointed in the Introduction, is demonstrated during configuration management (CM). Both the conceptual discussion about extended FBM (in Sec. 4) as well as the case study (in Sec. 5) shows

- Extended FBM is very helpful during evolution of a product line feature in both identifying which features will be affected and how “big” the effect will be. Besides, the uni-directional dependency of features in the lattice as shown in Fig. 17, in contrast with features in OO system as shown in Fig. 18, greatly simplifies the management of feature modification.
- Section 5.5 has explained how FBM’s reliance on AOP has enabled it to reap the fruits with regards to the shielding (to a degree) of feature modification. The features when implemented in AOP following the guidelines for product line feature implementation of [14] will have less dependence on each other thus easing feature evolution.
The traceability of a user feature to its implementation is one of the strengths of FBM. Specifically, testing of a user requirement is made quite easy. We can follow through from requirement to analysis, design and finally to implementation, a particular requirement to trace if everything in every step is according to the developers’ design.

8. Conclusion and Future Works

We have presented an ongoing case study that shows the application of FBM to the construction of a military product line. We have shown how aspects can be used to represent functionalities considering whether they are (i) functional or non-functional requirements, and (ii) user-visible or not (extracted features). By applying FBM, we succeeded in identifying dependencies among features more clearly. We also extended the original FBM to include dependencies that are peculiar to product lines and have shown how those dependencies can be incorporated.

Often feature modelling [1, 7, 10] is confined to domain engineering. As a result, there is no clear traceability between identification of features and their implementation. Because of such lack of traceability, even if we know the dependency among features in the domain, we have difficulty identifying the dependency at the implementation level. Our approach, by using aspects, solves that problem. Thus, we have addressed one of the challenges of configuration management: which software module must be modified to accommodate a modified/new user requirement.

Finally, we have shown, using FBM, how a dependency graph can be constructed among features which can be used to show how modification of a given feature affects other features. Therefore, we can manage the ripple effect caused by the modification. We also learned during the development of specific products, how such graphs can be used to show the difficulties associated with creating each product. Therefore, our approach deals with another challenging point in configuration management: i.e., what is the effect of modifying a specific software module on the rest of the system.

In the future, we plan to finish the case study (including the implementation). We will also try to evolve the product line as a whole as well as each product to see how our approach helps us with configuration management.

References


