FOOM - Feature-based Object Oriented Modeling: Implementation of a Process to extract and extend Software Product Line Architecture

Patrick J. Tierney, P.Eng.  Projexis, Inc.  Carlisle, Ontario  LOR 1H2, Canada  patrick@tierney.ca

Samuel A. Ajila, Ph.D., MIEEE  Systems & Computer Engineering  Carleton University, 1125 Colonel By Drive, Ottawa, Ontario, K1S 5B6, Canada  ajila@sce.carleton.ca

ABSTRACT

FOOM is a synthesis of the FODA (Feature Oriented Domain Analysis) and Horseshoe models. It includes an Object Oriented approach to Product Line Family architectures. It focuses on identifying user-driven features throughout the product line architecture; organizing the architectural assets to lend them to substantial re-use; and instantiating multiple products from a single architecture. The Unified Software Development Process (USP) is used as a template for describing the architectural transformation from a base product to a product-line. Unified Modeling Language (UML) is used as a notation for the various assets developed in the FOOM process. This paper introduces the notion of a feature class and feature list. The feature list serves as contracts between the product and the features. A simple heuristic based on the definition of user visible objects and non-user visible objects is used in building the feature model. In its entirety, FOOM is a synthesis of FODA and the horseshoe models, but we are concerned with the aspects that integrate and extend the FODA methodology in this paper.

Key words: Object Oriented Model, Features, Domain Analysis, Software Product Lines, Design, and Analysis

1.0 INTRODUCTION

When a new Software project based on previous products is launched, a common belief is “we have built one of these before and we can build another, just bigger and better.” The intuitive idea is to “borrow” code and other assets from one application and use it in the new project. Ad-hoc Software re-use strategies like this often fail. What is needed is to step back, look at the product that is being built, abstract the details of the implementation and start to develop architectural patterns that will help in planning a whole family of products with a common architecture. Re-use without adequate planning is a time wasting venture. The benefits of re-use [8, 9, 11] include reduce time to market, reduce product development costs, improved process predictability, increased product quality, achieve large-scale productivity gains, and increase customer satisfaction.

A key challenge to taking a product line approach is that evolutionary methods of software development are required. In a single-product approach, the architecture is evaluated based on the requirements for that product alone. However, in a product line approach, the designer must consider the requirements for the family of systems, and the relationship between those requirements and the ones associated with each particular instance.

In this paper, we propose a Feature-based Object Oriented Modeling (FOOM) that will capture a product’s architecture so that we can logically transform it to one that is applicable to other products in the same family. FOOM focuses on identifying user-driven features throughout a product line’s architecture. The architectural assets are organized in such a way to lend them to substantial re-use. In addition, mechanisms are provided for instantiating multiple products from a single architecture.

The rest of this paper is organized as follows. Section two presents software product line engineering concepts and discusses the Feature Oriented Domain Analysis (FODA). FOOM – Feature-based Object Oriented Modeling is presented in section three, application of FOOM to Sonar Systems is given in section four, and a conclusion in section five.

2.0 SOFTWARE PRODUCT LINE ENGINEERING

A software product line [3, 5] is a group of software systems sharing a common (not identical), managed set of features that satisfy a well defined set of needs (market, special mission or otherwise) and that are developed from a set of common core assets for a specific application domain. Modeling product lines provides substantial reuse advantages. Software product line engineering is a superset of three constituent disciplines: domain engineering, software architecture, and software re-engineering.

2.1 Domain Engineering

Domain engineering is the systematic creation of domain models and domain-specific architectures and their use in building applications. The emphasis is on reuse. In fact, domain engineering is about developing assets for reuse. Reusable components must be designed to be easily tailored and easily testable. However, the size of the domain is very important. Domain engineering requires a deep and thorough understanding of the commonalities and variations inherent in the application domain. Domain engineering life-cycle is characterized by [1, 6]: Domain analysis – identifying, grouping, and representing relevant information in a domain based upon the study of existing systems; Domain design – developing a design model from the analysis; and Domain implementation – identifying reusable components from the domain design and application architecture.

2.2 Software Architecture

Software architecture is a set of decisions about the organization of a software system. It includes the structural elements, interfaces between these elements, collaborative behavior of the elements, and the composition of the elements into larger subsystems. In the context of product lines, software
architecture focuses on the representation, definition, and evaluation of software architectures and their use in engineering software-intensive systems in a particular domain. Software architecture represents a common vehicle for communication among a system’s stakeholders, and is the arena in which conflicting goals and requirements are mediated. It is also an abstract reusable model that can be transferred from one system to the next.

Software architecture is one of the key reusable assets that form the basis of a software product line. Different products in the product line usually share the same architecture, or are built using prescribed variations of a common architecture [1, 3].

2.3 FODA – Feature-Oriented Domain Analysis

The Feature-Oriented Domain Analysis (FODA) methodology [6, 9, 10] was founded on a set of modeling concepts and primitives used to develop domain products that are generic and widely applicable within a domain. It builds on three fundamental sub-processes: domain analysis, feature analysis and feature modeling. Domain analysis focuses on identifying a product that is believed will form the kernel of the product line. Feature analysis applies commonality and variability analysis to develop a list of common, required functions across the domain, as well as a top-level list of their differentiating characteristics. The result of feature analysis provides the constituent artifacts to populate the feature model. FODA method applies the aggregation and generalization primitives to capture the commonalities of the applications in the domain in terms of abstractions. Differences (variability) between applications are captured in refinements. The feature-oriented concept of FODA is based on the emphasis placed by the method on identifying prominent or distinctive user-visible features within a class of related software systems. These features lead to the conceptualization of the set of products that define the domain. However, many of FODA processes and products are from the non object-oriented (OO) world. They are based on functional decomposition, which makes FODA less straightforward to develop OO product family architectures. With the general adoption of the modern software notations, most of the artifacts of FODA can be instantiated with UML in order to extend them in OO context.

3.0 FOOM – Feature-based Object Oriented Modeling

In this section, we propose a model that will update and integrate the FODA methodology in order to accommodate product lines and object oriented design. Looking at the problem of architecture extraction, implementation and evolution from the perspective of features brings with it a strong aspect of traceability between user requirements and the development of domain, product line, and application architectures. Product development in some industries, such as telecommunications, can be organized completely around features.

3.1 FOOM Development process

Underpinning FOOM is the Unified Software Development Process (USP) as described in [4]. USP is used as a template for developing and describing each of the architectures in the transformation from base product architecture to product line architecture. An integral part of this standardization is the adoption of the Unified Modeling Language (UML) [2] as the notation for the various assets developed in the process. There are two distinct sets of activities in the USP: analysis and design. Analysis can be further broken down to identifying actors, identify scenarios, identify use cases, refine use cases, and relationships among use cases. The basic steps in design are: identify design goals, design initial subsystem decomposition, map subsystems to hardware and software platforms, manage persistent storage, define access control policies, select a control flow mechanism, and describe boundary conditions. Figure 3.1 illustrates the relationship between analysis and system design. It is an iterative-incremental process generating a number of refinements as the notion of how a system needs become clearer.

![Diagram](https://via.placeholder.com/150)

Figure 3.1 - Modeling elements used to describe a Software Architecture

3.2 An Object-Oriented Perspective of the Feature Model

The first building block of the feature model is the feature class. Considering FODA’s feature model, the feature class is specialized into three sub-types:

- **Required features** - must be present for the system to function as intended. There is normally only one version of a required feature for a given product.
- **Alternate features** - are subclasses of required features with their differentiator being that several versions of flavors of a particular feature are available, but only one version can be used to provide that feature’s functionality.
**Optional features** - are not required for the basic product to function, but rather they provide functionality supplemental to the required feature set.

The next set of artifacts in the feature model relates to how the features are assembled. In the first place, packages are used to consolidate products and features. Secondly, aggregation is used to illustrate the composition of products with multiplicity implying a sense of required, alternative and optional. Thirdly, stereotyping associations are used to refine the aggregation rules [2]. Lastly, the model is simplified with the introduction of a feature list, which is a contract between each of the products and the feature set of the product family. The feature list serves as an association class between the product and the features. The aggregation rules are the class invariant, written in the Object Constraint Language (OCL) [12]. Reducing the relationships to a few lines of OCL substantially reduces the maintenance efforts and keeps the model readable. Figure 3.3 shows the relationships between products and features in the Feature Model using contracts.

The contract is built by creating the feature list for the product line. Its class invariant contains enumeration of the members of the product families plus enumeration of the required, alternative and optional features.

```
contract <Product Line>
    ProductType enum( Product1, Product2, ....,ProductN)
    RequiredFeatures enum(RFeature1, RFeature2,...,RFeatureN)
    AlternateFeatures enum(AFeature1, AFeature2,...,AFeatureN)
    OptionalFeatures enum(OFeature1, OFeature2,...,OFeatureN)
end contract;
```

From there, we can develop contracts for the feature lists of the individual products in the product line.

```
contract <Product>
    self.RequiredFeaturesList-> includes( RFeature1) and
    self.RequiredFeaturesList-> includes( RFeature2) and
    ... self.RequiredFeaturesList-> includes( RFeatureN);
    self.AlternateFeaturesList->
```

Figure 3.3 - Using contract to simplify the relationship between products and features in the Feature Model

The inheritance relationship between the product line and family members' contracts is shown in Figure 3.4. Use of enumeration of products and features and Boolean expressions allows for a very concise definition of an individual product's features.

### 3.3 Feature Discovery

The feature model is built using a simple heuristic can be – the user visible objects: **boundary objects and algorithm objects** are mapped into **features** and this is stereotyped as **<algorithm>**. The non-user visible objects: **control objects** are mapped into **functions**, which is stereotyped as **<control>**. These mappings are consistent with the definitions of the various types of objects. This simple heuristic proved invaluable in providing a good first estimate of the feature models for the architectures at all levels.

### 3.4 Integrating Feature Model

Now that the feature model has been explored, its relationship to the other parts of USP must be determined. The feature model [5] is a mechanism for refining the requirements elicited during analysis and how these requirements are related to the design assets (figure 3.6a). Our model extends this view, incorporating the constituent artifacts of the object model (figure 3.6b).

The next step is to determine the relationship of the feature model vis-à-vis other parts of the USP. From a stakeholder perspective, the feature model is a mechanism for refining the requirements elicited during analysis and thus serves as an intermediate step between analysis and design (cf. figure 3.7).

The resulting model (figure 3.8) is multidimensional. It is entirely supported by an object model. The relationship between the analysis and design models remains. There is also an independent relationship between the analysis and feature models, where evolution may occur without the need to build the design model. Similarly, the feature and design models may evolve independent of analysis, although, this should not be done until we are building individual architectures from our product line architecture.

![Figure 3.4 – Relationship between product line and product feature](image-url)
4.0 Applying FOOM to a family of Sonar Systems

In this section, we present how we apply FOOM to an example application – a family of sonar systems. Sonar is a system that captures transmitted and reflected sound in order to detect and locate underwater objects. Active sonar captures the reflected waves of transmitted acoustic energy (the characteristic "ping"). Passive sonar listens to the background acoustics of the marine environment. The primary function of any sonar is to capture acoustic energy from the marine environment, digitize the analog input, process the data, and display results on the operator console. Modern sonar uses combinations of complex hardware and software to perform their designated tasks and control the systems themselves. A transducer is used to capture the acoustic energy, whether it is background noise or reflected from a ping originating from a transmitter. The transducer also performs the analog to digital conversion. The digitized information is then forwarded to a digital signal processor that performs real-time algorithmic analysis to identify acoustic features. The features are displayed on the operator console. The whole system is controlled by the sonar controller, which also provides an interface to the command and control systems of the ship. Figure 4-1 provides an overview of this process. In this analysis, we will examine four types of sonar systems, which are deemed sufficiently diverse as to represent a substantial portion of the sonar domain – Hull Mounted Sonar (HMU), Variable Depth Sonar (VDS), Towed-Array Sonar (TAS), and Mine Hunting Sonar (MHS).

Figure 4.1 – Sonar system overview

The application of FOOM to the family of sonar systems is done in a systematic fashion, starting with the identification of a suitable base product. If the base product’s architecture is not defined in a form consistent with the processes outlined in FOOM, then some architecture extraction work is required. This is the case with our sonar example.
To do this, we start by selecting the top 5-20% of the use cases that will identify all the major software components [3]. A new analysis model is developed using these use cases and by modifying the stereotyping notation. Starting with the new analysis model, the feature model is constructed by transforming the analysis classes and control objects to functions and features. Table 4-1 lists the use cases that were selected for the HMS.

Figure 4.3 – Base product feature model

To build the Feature Model in accordance with the heuristic suggested in FOOM, we create objects based on the actors that are not directly user visible and designate them with the stereotype <<base-product function>> and objects based on user-visible boundary objects are assigned the stereotypes <<base-product feature>>. In addition, more objects – based on the use cases are created and they are designated <<base-product feature>>. Aggregation/composition associations are shown as <<includes>> relationship or as an implication that a particular feature is known only to be part of one system function (figures 4.2 & 4.3).

<table>
<thead>
<tr>
<th>Use Case Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>InitializeSystem</td>
<td>Manage power-up sequences and initial configuration of subsystems</td>
</tr>
<tr>
<td>ActiveDetection</td>
<td>Detect objects in the surrounding water</td>
</tr>
<tr>
<td>PassiveDetection</td>
<td>Process and display income passive acoustic features</td>
</tr>
<tr>
<td>DisplayAcousticFeatures</td>
<td>Displays acoustic features, tracks and other visuals on the operator console</td>
</tr>
<tr>
<td>ProcessOperatorCommand</td>
<td>Process a command from the operator console</td>
</tr>
<tr>
<td>UpdateTracks</td>
<td>Analyze acoustic features to detect potential targets</td>
</tr>
<tr>
<td>ProcessCCSMESSages</td>
<td>Process (Rx/Tx) messages between sonar and ship’s command and control system</td>
</tr>
</tbody>
</table>

Table 4-1 Primary hull-mounted sonar use cases

Once we have a good understanding of base product as represented by the architectural artifacts (table 4.1, figures 4.2 & 4.3), we can proceed with the development of the domain architecture (the details of this are beyond the scope of this paper). As stipulated at the beginning of this paper, FOOM is a synthesis of FODA and the Horseshoe models of SEI.

The next step is to build both the product features and product functions matrix [7] (cf. tables 4.2 & 4.3). From these tables, we can analyze both the commonality and variability in the domain in order to construct the product family feature model.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Hull Mounted</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>Variable Depth</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>Towed Depth</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>Mine Hunting</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
</tbody>
</table>

Table 4-2 Summary of features for the various sonar types

Now that we know the products in the family and the features available for each, we can build the product contracts. The product contracts are the invariants for association classes defining the relationship between the family members and the feature set for the entire product line. These contracts are written in UML. Using an inheritance relationship between the product line feature contracts and the product contracts, we pass the enumeration to the sub-classed objects, via the class invariants. For the sonar family, the product-line contract class invariant is:
### Product Functions / Subsystem Controllers

<table>
<thead>
<tr>
<th>Product Functions / Subsystem Controllers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transmitter</td>
</tr>
<tr>
<td>Hull Mounted</td>
</tr>
<tr>
<td>Variable Depth</td>
</tr>
<tr>
<td>Towed Array</td>
</tr>
<tr>
<td>Mine Hunting</td>
</tr>
</tbody>
</table>

Table 4-3 Summary of Product Functions for various types of sonar

```small
contract <Product Line>
-- Identify the products in the family
ProductType enum( HMS, VDS, TAS, MHS);
-- identify the current set of functions
-- available to members of the product line
ProductFunctions enum( SonarController,
  SignalProcessor,
  OperatorConsole,
  TrackProcessor,
  Transducer,
  Transmitter,
  CCS Interface
);
-- identify the current set of features
-- available members of the product line
ProductFeatures enum( ActiveSubmarineDetection,
  PassiveSubmarineDetection,
  DetectMines,
  DetectTorpedoes,
  StealthMonitor,
  InitializeSystem,
  DisplayAcousticFeatures,
  UpdateTracks,
  SetDisplayFormat,
  TxRxCcsMessages,
  SetActiveMode,
  SetPassiveMode
);
end contract;
```

### 5. CONCLUSION

In this paper, we have established a framework of processes to assist architects in extracting product line architectures from a single product. That architecture, by its very nature, is intended to be evolved over time and to generate multiple products and multiple variants of each member of the family. This framework is based on the extension of FODA. Next, we have shown that the UML is suitable for describing abstractions at levels as high as whole product families, and, at the same time, is capable of describing software in detail. We have used standard mechanisms like stereotyping to provide traceability at every point of evolution in the life of a software system and its siblings, parents and descendants. Further, we have reinforced the notion of use cases as architectural patterns. The patterns can be applied across many or all of the products in a domain. This brings with it a substantial reduction in the cost and effort of developing a new product. In a time-to-market environment, this is the primary driver. Finally, of most importance, is the addition of a feature model to the Unified Software Development Process (USP). The purpose of the feature model is to formalize the relationship between user's perceptive of a system and how developers go about its design and implementation.

### BIBLIOGRAPHY

3. Jan Bosch, Design & Use of Software Architectures – Adopting and evolving a product-line approach, Addison-Wesley, 2000