The FOOM Method – Modeling Software Product Lines in Industrial Settings

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

Patrick J. Tierney, Pen.
Preexist, Inc.
Carlisle, Ontario
L0R 1H2, Canada

Abstract

This paper presents FOOM – Feature-based Object Oriented Modeling and its application to a family of sonar systems. FOOM is a synthesis of the FODA (Feature Oriented Domain Analysis) and Horseshoe models. The Unified Software Development Process (USP) is used as a template for describing the architectural transformation. FOOM focuses on identifying user-driven features throughout a product line’s architecture, organizing the architectural assets to lend them to substantial re-use, and instantiating multiple products from a single architecture. The paper introduces the notion of a feature class and feature list contracts between the product line and products. The application of FOOM to sonar system is done systematically starting with the identification of a suitable base product, then architecture extraction is done; domain analysis is then carried out in order to identify the current and future members of the product family, and finally, product line architecture is developed.

Keywords: Object Oriented, Features, Domain Analysis, Software Product Lines, Systems Design, and Evolution.

1. Introduction

A software product line [4, 8, 16] 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. The benefits of reuse includes [6, 7, 12, 13, 15] – reduced time to market, reduced 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 different methods of software development are required. In a single-product approach, the software architecture is evaluated with respect to the requirements of that product alone. However, a product line approach requires the software designer to consider requirements for a family of systems and the relationships between those requirements. The main goal of product line engineering is to reduce cost and increase customer satisfaction. To achieve this, both the common as well as the variable assets are dealt with explicitly instead of accidentally.

In this paper, we present the overview and the application of FOOM - Feature-based Object Oriented Modeling to a family of sonar systems. FOOM is a method that updates and integrates the FODA (Feature Oriented Domain Analysis) [6] and Horseshoe [7] methodologies in order to accommodate product lines and object oriented design. FOOM is a feature-based, object-oriented implementation of a process to extract and extend software product-line architecture. FOOM captures a product’s architecture so that it can be logically transformed to one that is applicable to other products in the same family. 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. A feature [4, 10] is a “behavioral concept” that captures user-visible aspects, such as functional and quality requirements of a software system. The purpose of feature modeling is to analyze the common factors and the differences within a family of software systems in terms of system features.

The rest of this paper is organized as follows. Section two presents the background to FOOM and the three-dimensional view of FOOM. Section three discusses the overview of FOOM methodology. The application of FOOM to sonar system is presented in section four and a conclusion is given in section five.

2.0 The Background of FOOM

FOOM synthesizes the FODA and Horseshoe methodologies. The Unified Software Development Process (USP) [11] is used to develop and describe 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) [3] as the notation for the various assets developed in the process.

2.1 FODA – Feature-Oriented Domain Analysis

The FODA methodology [6, 13] 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 to 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 feature modeling has several advantages [10] – variability control, requirements reuse, and support for configuration and development. The basic modeling concepts in FODA are abstraction – used to create domain products from specific applications in the domain, and, refinement – used to refine the generic domain products. 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. The domain analysis process consists of a number of activities, producing many types of models – context, feature, entity-relationship, functional, and architecture.

2.2 An Object-Oriented Perspective of the Feature Model

The first building block of the feature model is, of course, a 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. Aggregation is used to illustrate the composition of products with multiplicity implying a sense of required, alternative and optional. Stereotyping associations refines the aggregation rules [4]. The model is simplified with the introduction of a feature list: 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). Reducing the relationships to a few lines of OCL substantially reduces the maintenance efforts and keeps the model readable (figure 2.1).

![Diagram of feature model with contracts]

**Figure 2.1 – Using contracts to simplify the relationship between products and features in the Feature Model.**

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 Features>
ProductType enum{ P1, P2, ...., PN}
RequiredFeatures enum{ RF1, RF2, ..., RFN}
AlternateFeatures enum{ AF1, AF2, ..., AFN}
OptionalFeatures enum{ OF1, OF2, ..., OFN}
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( RF1) and
self.RequiredFeaturesList->includes( RF2) and
... self.RequiredFeaturesList->includes( RFN);
self.AlternateFeaturesList->includes( AF1) and
self.AlternateFeaturesList->includes( AF2) and
... self.AlternateFeaturesList->includes( AFN);
self.OptionalFeaturesList->includes( OF1);
end contract
```

The inheritance relationship between the product line and family members’ contracts is shown in Figure 2.2 Use of enumeration of products and features and Boolean expressions allows for a very concise definition of an individual product’s features.

![Diagram of inheritance relationship between product line and feature lists]

**Figure 2.2 – Relationship between product line and product feature**

The feature model is built using a simple heuristic – the user visible objects: boundary objects and algorithm objects are mapped into features; non-user visible control objects are mapped into functions. These mappings are consistent with the definitions of the various types of objects [4]. This provided a good first estimate of the feature models for the architectures at all levels. 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 2.3).

![Diagram of relationship between feature, analysis, and design models]

**Figure 2.3 – Relationship between feature, analysis, and design models in FOOM**

The resulting model (cf. figure 2.4) is two-dimensional, supported entirely 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 [1] 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 building the individual architectures from the product line architecture.

![Figure 2.4– Bi-dimensional relationship between analysis, feature, and design models in FOOM](image)

### 2.3 Extending the Horseshoe methodology

The next building block in FOOM is the Horseshoe model. The SEI Horseshoe model [7, 14] presents a code-based approach for extracting system’s architecture. The model identifies three basic reengineering processes: 

**Code and architecture recovery / conformance evaluation** - analysis of an existing system to recover the architecture by extracting artifacts from source code. The conformance recovery is used to compare the recovered architecture to the “as-designed” architecture. The discovered architecture is also evaluated with respect to a number of quality attributes such as performance, modifiability, security, or reliability.

**Transformation** - The recovered architecture is reengineered to become the new architecture. This new architecture is also evaluated with respect to the system’s defined quality goals. It is also subjected to new requirements and some business objectives.

**Development** - In this process, the new architecture is instantiated. In addition, packaging issues are decided and interconnection strategies are chosen. This process may also include re-wrapping and recoding of legacy code in order to fit-in with the new architecture.

The horseshoe model provides an excellent concept for extracting architectures from an existing system, transforming the architecture, and providing rules for instantiating it. However, even the updated perspective [14] only looks at transforming the architecture for a single system or product. FOOM builds on the approach, focusing on migrating multiple architectures (i.e. -from a base product through to domain, to product line, and to product architectures) at the conceptual level only.

![Figure 2.5 – Modified Horseshoe model for a product line architecture.](image)

Once a base product has been identified and its architectural assets deemed to be suitable, a series of transformations are performed in order to transform the architecture, first to domain architecture that is sufficiently abstracted to represent all products in the domain. From there, variability analysis provides attributes that differentiate one product from another, leading to product line architecture. The final step is to develop the rules for deriving single-system architecture from the product line architecture. Figure 2.5 provides a pictorial description of this process. A UML perspective of the same logical transformation is shown in figure 2.6.

![Figure 2.6 – Relationships of the different stages of architecture transformations](image)
3.0 Overview of FOOM methodology

When the extensions of each of the supporting methodologies (FODA and Horseshoe) are aggregated in the new model, a new set of processes arises that generate product line architectures that are entirely object oriented. A feature model at each of architecture allows the evolution of each feature to be tracked. FOOM provides, in a single model, the ability to develop road maps for future evolution of product (cf. figure 3.1).

Figure 3.1 – Architecture transformation in FOOM

Figure 3.2 shows the three dimensional view of FOOM. There is a close relationship between all the different stages of the model. The same artifacts are used to describe the architecture at each step in the transformational process. All sub-models are internally consistent with one another in each architectural level.

In order to provide a very clear traceability mechanism, a series of stereotypes are used. The general form of the stereotype is <<architecture + artifacts>>, where architecture is the current phase in the logical transformation process and artifact is the object type.

Table 3.1 Examples of stereotype labeling for traceability of architecture elements

<table>
<thead>
<tr>
<th>Label</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>architecture</td>
<td>base product, domain, product-line and product</td>
</tr>
<tr>
<td>artifacts</td>
<td>feature, function, entity, boundary, control, subsystem</td>
</tr>
</tbody>
</table>

4.0 FOOM and the Sonar Systems

This section presents systematic application of FOOM to a family of sonar systems. Sonar 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 (figure 4.1).

Figure 3.2 Object model developed by with FOOM

Figure 4.1 – Sonar system overview

Four types of sonar systems that are deemed to represent sufficiently a substantial portion of the sonar domain will be examined:

- **Hull-Mounted Sonar (HMS)** – An all-purpose type of sonar excels in detecting submarines in
Variable Depth Sonar (VDS) - This incorporates essentially the same functionality as an HMS. Its distinguishing feature is that its wet end is in a separate winch-controlled to body, which is pulled behind the ship.

Towed-Array Sonar (TAS) - This is a passive type system with computing hardware similar to active systems, but requires only the software for passive Digital Signal Processor (DSP) and display.

Mine Hunting Sonar (MHS) – This is similar to HMS, but operates at much higher frequencies than other active sonar.

The application of FOOM to the family of sonar systems is done systematically starting with the identification of a suitable base product. If the base product’s architecture is not consistent with FOOM processes, then some architecture extraction work is required. Once the base product architecture is defined, extraneous use cases are removed, notations are modified to conform to the traceability rules through stereotypes, and a feature model is developed. Domain analysis is then carried out in order to identify the current and future members of the product family as well as the features for each product. Finally, product line architecture is developed and variability analysis is done to associate features with their respective products, enabling contracts for each family member to be developed to formalize their feature content. The focus here is on the development of the feature model for each of the architectures.

The HMS was selected as the base product because its use cases should identify all the major software components. 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.

Table 4-1 Primary hull-mounted sonar use cases

<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>

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 and 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 (cf. figure 4.3).

Once we have a good understanding of the base product as represented by the architectural artifacts (cf. table 4.1 & figures 4.3), we can proceed with the development of the domain architecture.

The next step is to build both the product features and product functions matrix [5] (cf. tables 4.2 & 4.3). These tables provide a solid foundation for domain analysis. From these tables, we can analyze both the commonality and variability in the domain in order to construct the product family feature model. Table 4.2 identifies the base product feature. Examination of this table reveals that active and passive detection features are not common across all products. These features are abstracted to Detect Acoustic Features, which is sufficiently general to be applicable to all products. Table 4.3 lists the products in the domain and the components required to implement them.
Table 4-2 Summary of features for the various sonar types

<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 Array</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>

In addition, transmitter is removed from the component list for the domain because passive systems do not have transmitter. It is not abstracted to a common description similar to the development of the Detect Acoustic features use case; instead, for the purposes of the domain model, the exact source of incoming acoustic energy is ignored and considered to be an outside stimulus. With these simplifications, the domain functions are reduced to Transducer, Digital Signal Processor, System Controller, Operator Console, Track Processor and CCS Interface. With this background, the use case diagram (not shown) and the domain feature model (cf. figure 4.4). Now that the feature model for the domain has been developed, the next phase of the logical transformation is to develop the product-line feature model. The process begins with revisiting the product family members, identifying their use cases and actors. From there, we use variability analysis to enumerate each product's features and functions.

Our goal is to build a hierarchical feature model derived from a series of use cases. The domain use cases become the abstract superclass for the whole subfamilies of features and functions found at the product line and product levels. The process starts with building a table of...
Figure 4.4 Sonar domain feature model

all the scenarios and use cases for all products in the family. Each use case is examined to determine if it, or a more generalized form, can be found on another product. If so, we assign it the <<product line use case>> stereotype. In turn, use cases found on specific products are designated with the <<product use case>> stereotype. This process must be done methodically to ensure that we capture the true relationships between the use cases throughout the product family. Table 4.4 lists the major use cases in the product family and indicates the products that would use them. This table is used to build the use case diagram (not shown) for sonar product family. The product-line use case diagram includes use cases for features of individual products to provide a mechanism for correlating the products back to the product family (cf. figure 4.5). Two areas in the use case diagram show multiple levels of specialization going from the domain to individual products, rooted in the use cases Detect Underwater Features and Display Underwater Features. Similarly, using the relationships between the different objects, we can build the product-line architecture feature model.

Now that we know the products in the family and the features available for each, we begin the process of building the product contracts.

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 Object Constraint Language (OCL), a component of UML. The contract for the product line includes the enumeration of all the features and functions available across all products, as well as the attributes for all common functions and features.

The sonar product-line contract class invariant is:

```
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 to members of the product line
  ProductFeatures enum{ ActiveSubmarineDetection, PassiveSubmarineDetection, DetectMines, DetectTorpedoes, StealthMonitor, InitializeSystem, DisplayAcousticFeatures, UpdateTracks, `
A typical contract for one of the family members identifies the features and functions required to instantiate the architecture for that product. For example, the feature contract for the Hull-Mounted Sonar (HMS) is:

```plaintext
contract <HMS Product>
-- Specify the product's functions
    self.FunctionsList->includes(
        SonarController ) and
    self.FunctionsList->includes(
        SignalProcessor ) and
    self.FunctionsList->includes(
        OperatorConsole ) and
    self.FunctionsList->includes(
        TrackProcessor ) and
    self.FunctionsList->includes( Transducer ) and
    self.FunctionsList->includes( Transmitter ) and
    self.FunctionsList->includes( CCS Interface );
    -- Specify the product's required features
    self.RequiredFeaturesList->includes(
        ActiveSubmarineDetection ) and
    self.RequiredFeaturesList->includes(
        PassiveSubmarineDetection ) and
    self.RequiredFeaturesList->includes(
        InitializeSystem ) and
    self.RequiredFeaturesList->includes(
        DisplayAcousticFeatures ) and
    self.RequiredFeaturesList->includes(
        UpdateTracks ) and
    self.RequiredFeaturesList->includes(
        SetDisplayFormat ) and
    self.RequiredFeaturesList->includes(
        SetActiveMode ) and
    self.RequiredFeaturesList->includes(
        SetPassiveMode );
    -- Specify the optional features to be included with
    -- this instance of the product
    self.OptionalFeaturesList->includes(
        DetectMines ) and
    self.OptionalFeaturesList->includes(
        DetectTorpedoes );
end contract;
```

Now that we have a feature model for the known members of the product family and a set of contracts for each product, we can begin the task of examining each subsystem with respect to the role it plays for each product. When completed, the design model will represent the detailed design for an entire family of products. In this paper, we will limit our discussion to the Operator Console subsystem. This subsystem is the most complex in terms of features because this is the system with most interaction with the users. Figure 4.6 shows the Operator Console subsystem class diagram. The two most obvious components are the operator input devices and the video display. The input devices are given base class because of reuse. The video also presents many possibilities for reuse, particularly in the various display formats – the combination of graphic primitives used to convey information in a meaningful form. Several of the sonar systems have active and
passive submarine detection displays. Others have displays such as mine-hunting and stealth monitoring.

5.0 Conclusion

Developing product lines requires an evolutionary approach. In a single product approach, the software architecture is evaluated with respect to the requirements of that product alone. In product line, the system designer must consider the requirements for a family of products and the relationships between those requirements. In this paper, we have presented a framework of processes to assist software architects in extracting architecture from a base product, examining it in the context of that product’s domain and extending it to capture a family of products in the same domain. That architecture by its nature is intended to be evolved over time to generate multiple products and multiple variant of each member of the family. Next, we have shown that the UML is suitable for describing abstraction at levels as high as whole product families. We have used standard mechanisms like stereotyping to provide traceability at every point of evolution in the product line. Further, we have reinforced the notion of use cases as architectural patterns and these can be applied across the product lines. Finally, the addition of feature model to the Unified Software development Process (USP) is very important. The purpose of a feature model is to formalize the relationship between the user’s perceptions of the system, how the developers go about the designing, and the implementation of the product. It also provides a tighter coupling between the analysis and design.

A complete description of FOOM and its application to family of sonar systems will be too voluminous for this paper, so we have had to limit ourselves to presenting the essential parts and omitting some details as well as limiting the number of figures. Furthermore, work on the application of FOOM to sonar systems is continuing especially in the area of system design and the extension of the product line architecture. In addition, the feature list contracts need to be extended to capture not only known products but future products as well.
References


