Feature-based Object Oriented Modelling (FOOM): Implementation of a Process to Extract and Extend Software Product Line Architectures

by Patrick J. Tierney, B. Sc. Eng., P. Eng.

A Thesis Submitted to
the Faculty of Graduate Studies and Research
in partial fulfilment of
the requirements for the degree of

Master of Science
Information and Systems Science

Faculty of Engineering
Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario, Canada, K1S 5B6
August 2002
©2002, Patrick J. Tierney

The undersigned recommend to the Faculty of Graduate Studies and Research acceptance of the thesis

Feature-based Object-Oriented Modelling (FOOM): Implementation of a Process to Extract and Extend Software Product Line Architectures

submitted by
Patrick J. Tierney, B. Sc. Eng., P. Eng.
in partial fulfilment of the requirements for the degree of

Master of Science
Information and Systems Science

Chair, Department of Systems and Computer Engineering

Thesis Supervisor

Carleton University
August 2002

ABSTRACT

Using a product line approach to software development and evolution requires much more than a reuse program: it requires the implementation of a common architecture across all members of the product family. FOOM represents a synthesis of the FODA (Feature Oriented Domain Analysis) , the Horseshoe model, the Unified software Development Process and the Unified Modelling Language (UML). It focuses on: identifying user-driven features throughout a product line's architecture, organizing the architectural assets to lend themselves to substantial reuse, and, instantiating multiple products from a single architecture.

DEDICATION

This work is dedicated to the memory of my son, Mark Daniel (1981-1983), whose brief time with us was filled with wonder and excitement as he discovered the world around him. His legacy has inspired me to look at each morning as a new beginning, to live each day to the fullest, to learn from all of life's experiences and to be at peace with the world as each day closes.

ACKNOWLEDGEMENTS

To my wife, and life partner for nearly 30 years, Cassie Kelly, whose constant and enduring love, support, and encouragement have helped me through many of life's challenges. Thank you. I love you.

To my daughter Sandra and my son David. Thank you for your understanding and patience during those many times when deadlines were looming.

To Samuel Ajila, for helping me complete this work. It has been a pleasure working with you.

To Lionel Briand, for introducing me to the finer points of software engineering and the study of software product lines.

To the faculty and staff of Systems and Computer Engineering, particularly Trevor Pearce, my academic advisor, Dorina Petriu, Murray Woodside, Daniel Amyot (University of Ottawa) and Tony Bailletti, my defense committe, and, Darlene Hebert and Judy Bowman.

To Computing Devices Canada, especially George Georgaras, Cindy Tutt, Simon Hebert, and John Moolenbeek, where I learned the best practices of Software Engineering.

To the men and women behind the National Research Council of Canada's O-Vitesse program, particularly Arvind Chhatbar and Hélène Biddiscombe for the opportunity to embark on my new career in Software Engineering.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

A software product line is a set of related products developed by an organization. These products share a common managed set of behaviors and attributes. Organizations are finding that a product line practice yields substantial measurable improvements in productivity and quality. First hand experience confirms that system development with an eye to sibling products substantially reduces the effort required for the design and implementation.

Typically, product-line development is characterized by processes and practices for developing an individual system and then creating variations of it. These variations on individual systems take continual investment in understanding new requirements, and in redesign, recoding and retesting. The result can be less than optimal designs, in terms of performance, quality and further evolution.

This work proposes a set of methodologies to design and build software families where members have a common architecture. The processes are based on providing value to the user. Documentation of the architectures uses industry standard notations and established software development practices.

1.1 A Definition of Software Product Line Engineering

Software product line engineering is a superset of three constituent disciplines: domain engineering, software architecture and reengineering. Each are required to adequately understand and build a product family.

1.1.1 Domain Engineering

Domain engineering is the systematic creation of domain-specific architectures and their use in building applications. The emphasis is on reuse: reusable components must be designed to be easily tailorable. For large systems, reuse of large, pre-integrated chunks is key [3].

Domain engineering requires a deep and thorough understanding of the commonalties and variations inherent in the undertaking. A process for domain engineering can be characterized by the following steps (Figure 1-1):

- *Domain analysis* is the process of identifying, collecting, organizing and representing the relevant information in a domain, based upon the study of existing systems, knowledge captured from domain experts, underlying theory, and emerging technology within a domain.
- *Domain design* is the process of developing a model from the products of domain analysis - requirements specifications, tables, models - and the knowledge gained from the study of software requirement/design reuse and generic architectures.

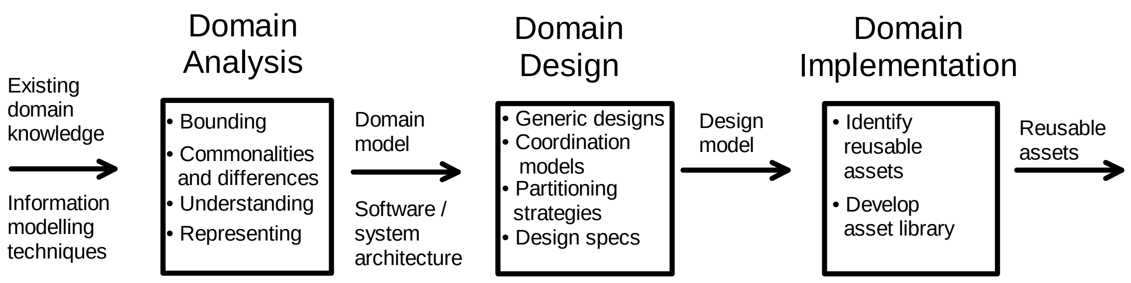


Figure 1-1 The SEI domain engineering process[3]

- *Domain implementation* is the process of identifying reusable components based on the domain model and generic architecture

1.1.2 Software Architecture

The architecture of a software system is the structure of the system comprising software components, externally visible properties of those components, and the relationships among them [24]. In an engineering context, it is also the set of requirements, plans and specifications that descibe the system in manner that designers can carry out the implementation of the system. In essence, it bridges requirements and code.

1.1.2.1 Architecture Terminology

Software architecture terminology varies depending on the level of abstraction and the intent of the particular design. Each type of architecture defines: element types and how they interact, mapping of functionality to architecture elements, and instances of architecture elements [9]. Following are architecture types used in this study:

- *Architectural pattern* defines generic element types and how they interact (i.e. client/server, peer-to-peer, single system).
- *Domain architectures* define element types and allowed interactions, but for a particular domain. These types define how the domain functionality is mapped to the architecture elements
- *Product line / product family architectures* apply to a set of products within an organization or company. They define element types, how they interact and how the product functionality is mapped to them. These architectures may also include mechanisms for identifying the commonalties and variabilities between individual products in the family.
- *Software system / software product architecture* applies to one system and describes the element types, how they interact, how functionality is mapped to them, and the instances of each element that exist in the system.

1.1.2.2 Architectural Views

The search for commonalties in various software architecture types has led to the evolution of four distinct views: conceptual, module, execution and code. Each view describes a different kind of structure. Between the views the structures are loosely coupled and address different engineering concerns. Figure 1-2 illustrates the intent of each view and the relationships between the different views.

- *Code view* - the organization of the source code, object code, libraries, binaries, which are then organized into versions, files and directories. The effectiveness of this organization can affect factors such as reusability of the code, build time for the system, etc.

- *Module view* - the decomposition of the system into major components, identification of interfaces and the partitioning of modules into layers.
- *Execution view* - allocation of functional components to runtime entities, handling of the communication, co-ordination and synchronization among those entities and mapping them to hardware.
- *Conceptual view* - description of the system in terms of its major design elements and the relationships among them.

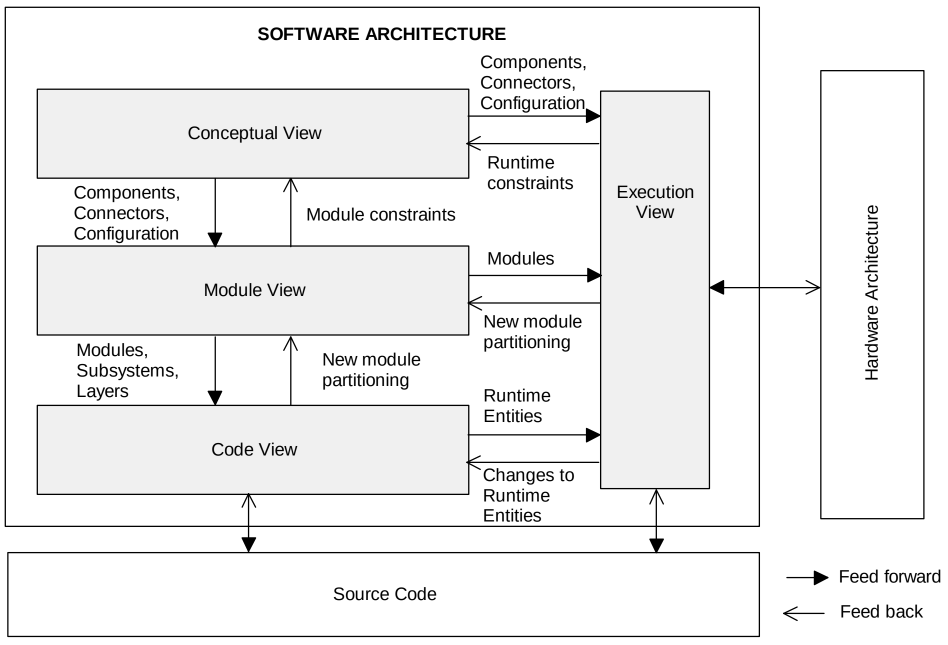


Figure 1-2 The four views of software architecture [9]

1.1.2.3 Architecture and Software Product Lines

A key challenge to taking a product line approach is that different methods of development are required. In a single-product approach, the architecture is evaluated with respect to the requirements of that product alone. Single products can be built independently, each with a different architecture. However, in a product line approach, the designer must also consider requirements for the family of systems and the relationship between those requirements and the ones associated with each particular instance.

In the context of product lines, a software architecture focuses on the representation, definition, and evaluation of software architectures and their use in engineering software-intensive systems in a particular domain. A robust software architecture applicable across the product line is critical.

Software architecture forms the backbone for building successful software-intensive systems. A system's quality attributes are largely permitted or precluded by its architecture. Architecture represents an abstract reusable model that can be transferred from one system to the next. Architecture represents a common vehicle for communication among a system's stakeholders, and is the arena in which conflicting goals and requirements are mediated .

Software architecture represents 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 [5].

1.1.3 Reengineering

Reengineering focuses on leveraging existing software assets[6] and the evolution of legacy systems, especially as a baseline for product lines[2]. Few systems start out as a "green field" development effort. A realistic approach for either migrating to a modern software architecture or developing a product line begins with analyzing legacy systems to understand the current architecture and developing a strategy for mining and reusing existing assets [6].

1.1.3.1 The Horseshoe Re-engineering Model

SEI's Horseshoe Model (Figure 1-3), as described in [19], presents a code-based approach for extracting a system's architecture. It is paradigm-agnostic, leaving its implementation to be defined on a system by system basis. This model identifies three basic reengineering processes:

- *Architecture Recovery / Conformance* - analysis of an existing system to recover a system's current architecture by extracting artifacts from source code. This recovered architecture is analyzed to determine whether it conforms 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.
- *Architecture Transformation* - The "as-built" architecture recovered in the previous step is re-engineered to become a desirable new architecture. It is re-evaluated against

the system's quality goals.

- *Architecture-based development* - instantiates the desired architecture. In this process, packaging issues are decided and interconnection strategies are chosen. Code-level artifacts from the legacy system are often wrapped or rewritten in order to fit into this new architecture.

The horseshoe model provides a road map for extracting an architecture from an existing system, transforming the architecture, say from functional decomposition to object-oriented, and then providing rules for instantiating the new architecture.

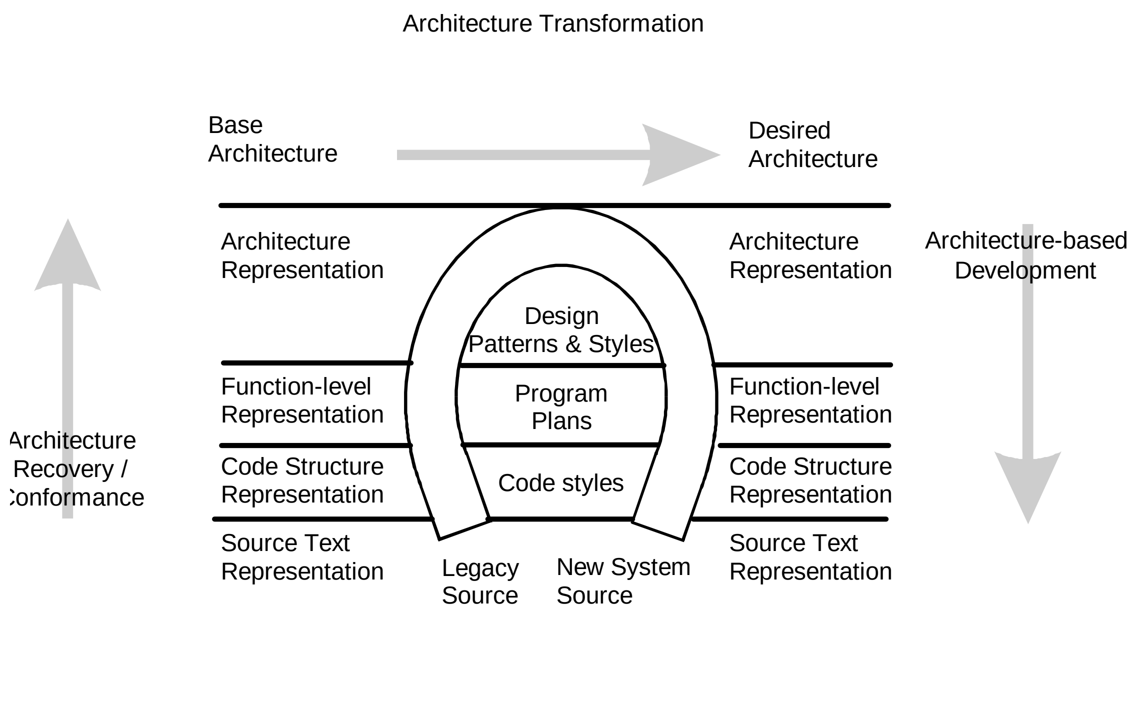

Architecture Transformation
Base Architecture
Desired Architecture
Architecture Representation
Architecture Representation
Architecture-based Development
Design Patterns & Styles
Program Plans
Code styles
Function-level Representation
Function-level Representation
Architecture Recovery / Conformance
Code Structure Representation
Code Structure Representation
Source Text Representation
Legacy Source
New System Source
Source Text Representation

Figure 1-3 The SEI horseshoe model for reengineering[19]

1.2 Feature-based Modelling

Current software architecture modelling technologies place a strong emphasis on capturing the user's requirements, but bury them inside constructs such as use cases[8]. A mechanism is needed for drawing more of the software developer's focus towards the final objective of any development project, the expectations and perceptions of the user.

1.2.1 What is a Feature?

In its simplest form a feature is an aspect of a software system, such as a behavior or an attribute, as perceived by the user. It represents a view of the system that is quite distinct from that of the software architect, hiding the details of the software from the user. Features can be used to group many requirements and their ensuing design artifacts into a single entity [26].

It is easy to think of a feature as an autonomous, atomic elements of a software system [26]. However, experience shows that, in any nontrivial system, this is not the case. Looking at a system from the feature level provides a macroscopic view of its static and dynamic structure as perceived by the user.

From a product line perspective, a feature can be considered to be an architectural pattern taken from several instances of a product family's siblings.

1.3 Motivation

User requirements can be captured in a very simple concept - features. These are the attributes and behaviors of a product, software or otherwise, that provide value. Modern software development processes place a strong emphasis on capturing the user's requirements, but bury them inside constructs such as use cases. A mechanism is needed for drawing more of the software developer's focus towards the final objective of any development project, the expectations and perceptions of the user.

1.4 Thesis Statement

The major contributions of this work are:

1. Feature model artifacts and mechanisms for its development and evolution are added to the Unified Software Development Process (USP).
2. FODA (Feature Oriented Domain Analysis) is extended beyond domain engineering to product line engineering by incorporating the extended USP (in 1 above) into its definition.
3. The horseshoe re-engineering model is extended to model multiple transformations, from a base product to domain, product-line and product architectures
4. The feature contract, which defines the rules for instantiating a product from the product line architecture, is introduced.
5. The resulting model - FOOM (Feature-based Object Oriented Modeling) is applied in a systematic way to a family of sonar systems.

1.5 Thesis Overview

In this first chapter, the subject of software product line engineering has been introduced and the reasons for embarking on this research project have been presented.

Chapter 2 provides a cursory review of the Unified Software Development Process and the UML.

Chapter 3 describes the state of the art in software product line engineering.

Chapter 4 identifies and discusses the strengths and weaknesses of the methodologies presented in Chapter 3. This discussion leads to the formulation of the problem addressed by this thesis.

Chapter 5 presents the details of the proposed model. It identifies the extensions to existing practices and introduces new processes that address the problem of modelling software product line architectures.

In Chapter 6 the new model is applied to an example application – a family of sonar systems. The concepts of sonar are introduced followed by a description of current and future members of the product family. From there, the new model applied systematically to the example.

Chapter 7 evaluates Feature Oriented Object Modelling (FOOM) against the methodologies in Chapter 3, recounts the contribution to research of this thesis and closes with the future direction to the work.

Chapter 2

Software Architecture with the Unified Software Process and the UML

Underpinning FOOM is the Unified Software-Development Process (USP) as described by [8][11]. It is used to describe FOOM itself, and as a template for developing and describing each of the architectures in the transformation from a base-product architecture to a product line architecture. An integral part of this standardization is the adoption of the Unified Modelling Language (UML) as the notation for the various assets developed in the process. The different phases are reviewed herein to provide a context for the model, but familiarity with the processes in [8][11] is required for a full understanding of how the USP is applied in the model.

2.1 Requirements Definition

The purpose of requirements definition is to identify a problem area and build a system specification that addresses the problem [11]. The end result of this work is a system specification: a natural language artifact. However, we do use the UML Use Case Model to assist us in understanding the activities the system will perform and the stakeholders, human and non-human, involved in the system (Figure 2-1). [11] suggests the following activities for requirements definition:

- *Identifying actors* - the different types of users the system will support
- *Identify scenarios* - concrete sets of interactions between one or more actors in the system.
- *Identify use cases* - generalized sequences of interactions between one or more actors and the system. This is captured in the Use Case Diagram (Figure 2-2) and documented in natural language. Table 2-1 is a suggested template.
- *Refine use cases* - elaborate use cases to include errors and exceptional conditions. The <<extends>> relationship is used for this.
- *Consolidate relationships among use cases* - eliminate redundancies.   The <<includes>> relationship helps to simply the number of use cases.

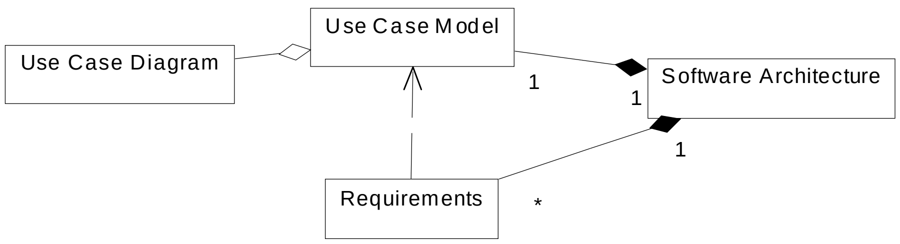


Figure 2-1 Relationship of the use case model to the USP

Use case name	AutomatedTellerWithdrawal
Participating actors	Customer, Keypad, Display, CashDispenser, Server
Precondition(s)	Customer has no overdraft
Flow of events	1. Customer inserts ATM card 2. Display message - request password 3. Customer enters password 4. Verify password 5. Request transaction type 6. Customers responds - WITHDRAWAL / AMOUNT 7. Verify transaction / adjust account balance 8. Dispense cash 9. Return card
Postcondition(s)	Customer has cash, account balance reduced
Special requirements	Customer has account with bank

Table 2-1 Example template for documenting a scenario - ATM withdrawal [11]

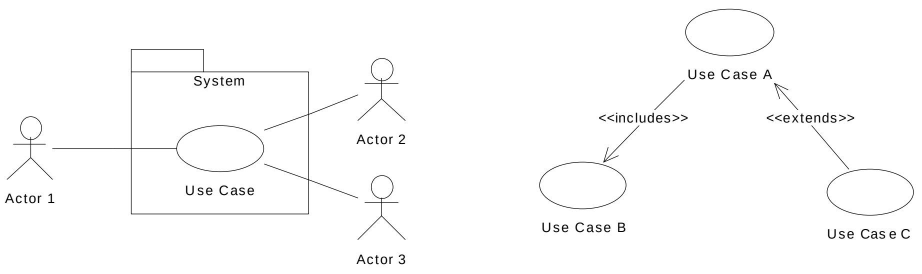


Figure 2-2 UML notations used for requirements analysis.

2.2 Analysis with the USP

The next step in the USP is analysis. Here we gain an understanding of the nature of the system by specifying behaviors and interactions, identify attributes such as quality and performance, and beginning to build a structure for the final product. The steps in the process are:

- *Define participating analysis objects* - Examines each use case and identifies candidate objects. The participating objects can be further broken down into entity, boundary, algorithm and control objects (Figure 2-3).
    - *Entity objects* represent persistent or long lived information traced by the system.
    - *Boundary objects* represent interactions between the user and the system.
    - *Algorithm objects* represent a task performed by the system.
    - *Control objects* aggregate algorithm, entity and boundary objects.

  Use these objects as the kernel for building the Analysis Class Diagram (Figure 2-4).

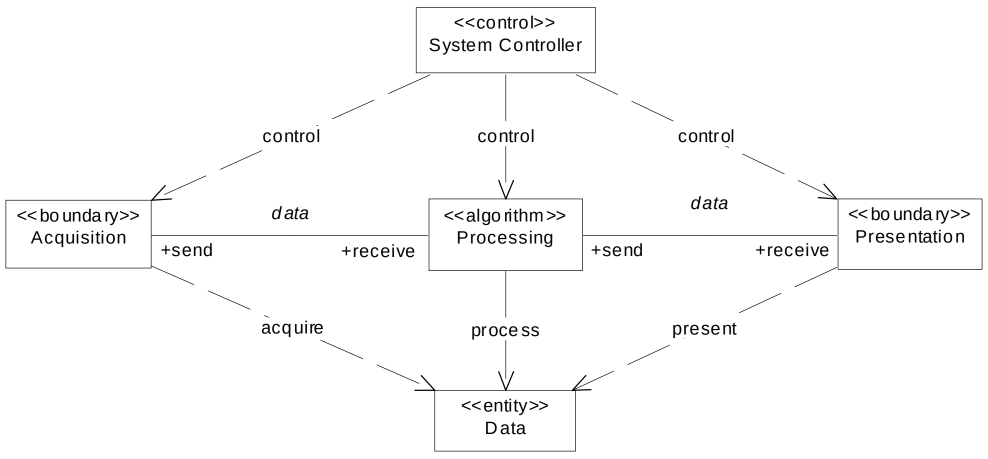
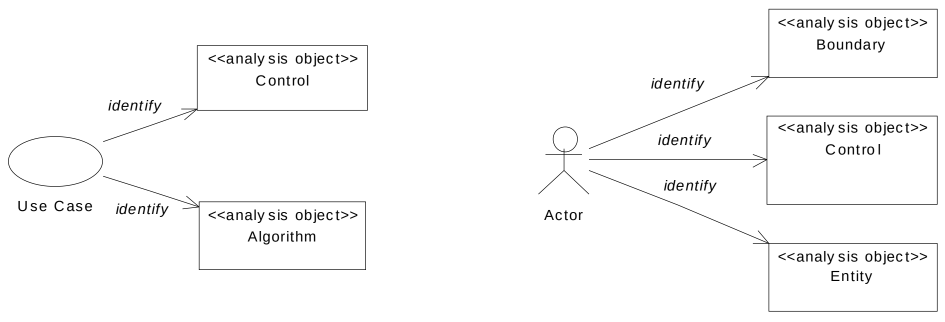


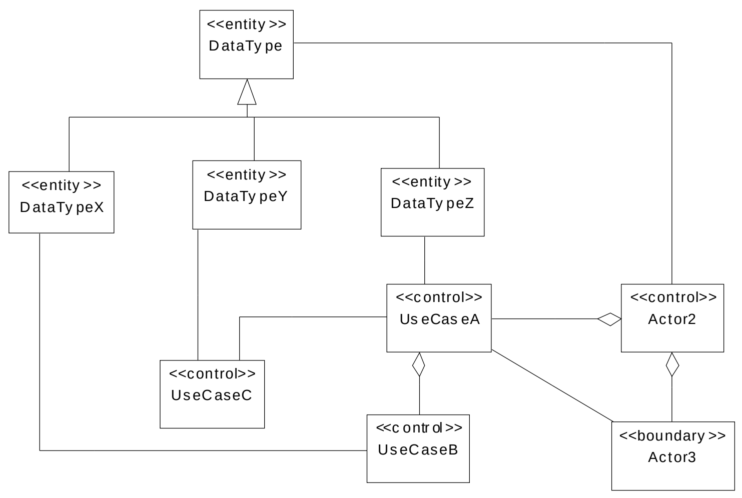
Figure 2-3 Relationships between the different analysis objects

- *Map use cases to objects and define interactions* - sequence diagrams tie use cases with objects. See Figure 2-4.
- *Model nontrivial behavior of objects* - in addition to sequence diagrams, which represent behavior from the perspective of the user, statecharts are used to represent the behavior of the system from the perspective of individual objects. Statecharts are constructed only for objects with an extended life span and nontrivial behavior. See Figure 2-5.
- *Define attributes* - named properties of a class defining a range of values an object can contain.
- *Define associations* - a relationship between two or more classes denoting possible links between instances of the classes; they have names and can have multiplicity and roles attached at each end.
- *Consolidate the model* - solidify the model by introducing qualifiers, generalization relationships and suppressing redundancies.
- *Review model* - the model is examined for correctness, consistency, completeness and realism.

Figure 2-6 illustrates the structure of the USP's analysis model.

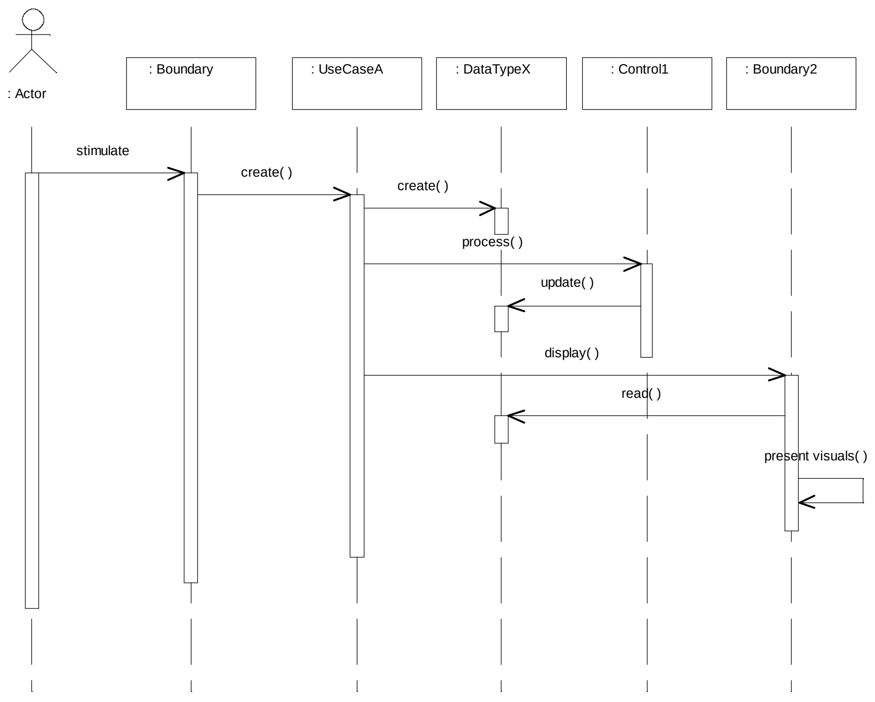


a)

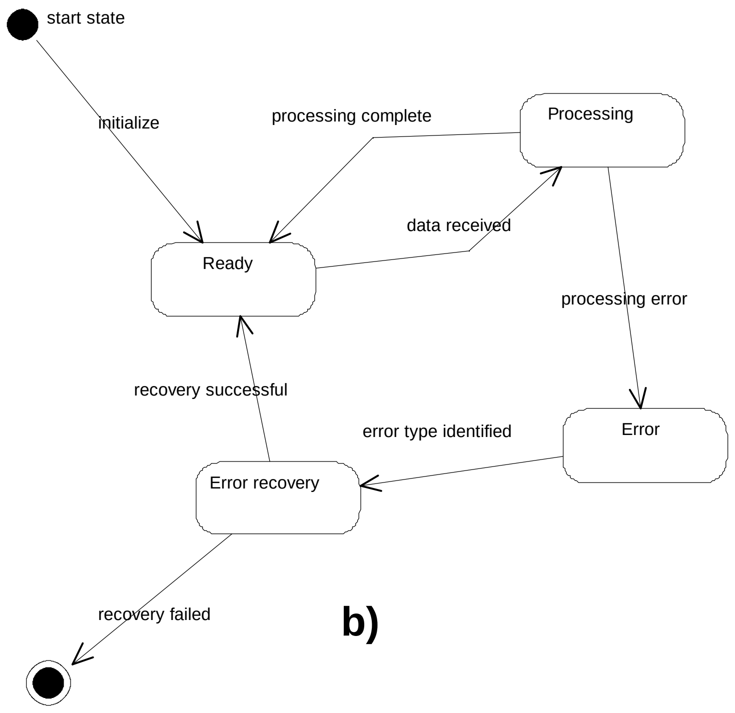


b)

Figure 2-4 UML notation for structural aspect of the analysis model: a) transform use case diagram artifacts to analysis objects b) Analysis Object Class diagram



a)



b)

Figure 2-5 UML notation for functional aspect of analysis model: a) Sequence Diagram b) Statechart

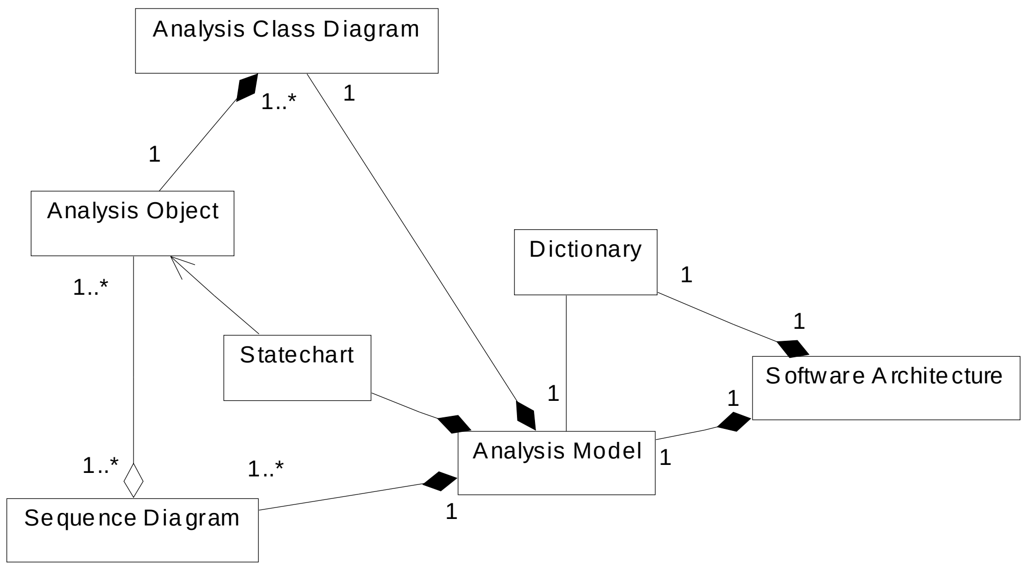


Figure 2-6 Artifacts comprising the analysis model in the USP.

2.3 Design with the USP

Once we are satisfied the Analysis Model adequately defines the behavior and basic structure of our system, we are ready to begin development of the Design Model, whose overall structure is illustrated in Figure 2-7. The basic steps in this process are:

- *Identify design goals* - non-functional requirements such as reliability, fault tolerance, security and extensibility.
- *Design an initial subsystem decomposition* - break the system down into simpler parts, each providing services to other subsystems. Document this work with the Subsystem Class Diagram (Figure 2-8).

- *Map subsystems to hardware and software platforms* - examine the allocation of subsystems to computers and the design of the infrastructure for supporting communication between subsystems and document with a Deployment Diagram (Figure 2-9).
- *Manage persistent storage* - identify the objects that need to be persistent and determine the most effective way of storing them
- *Define access control policies* - define for each actor in the system which operations they can access on each shared object
- *Select a control flow mechanism* - determine which actions - as previously defined in use cases) need to be executed for a given stimulus and the order in which they should occur.
- *Describe boundary conditions* - decide how the system is started, initialized and shut down.

Figure 2-10 illustrates the relationship between analysis and system design. It is an iterative-incremental process generating a number of refinements as the notion of how the system needs to be built becomes clearer.

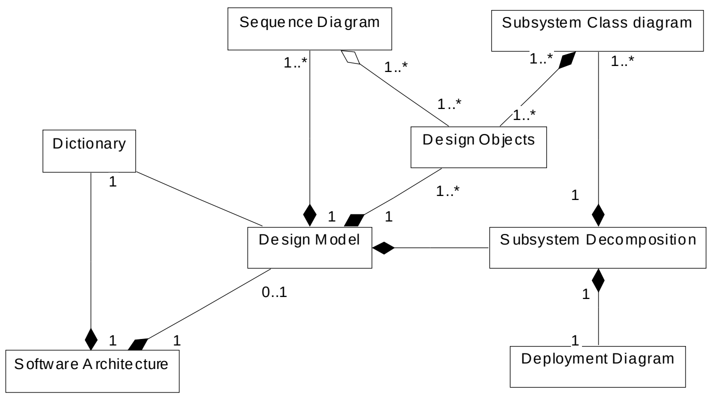
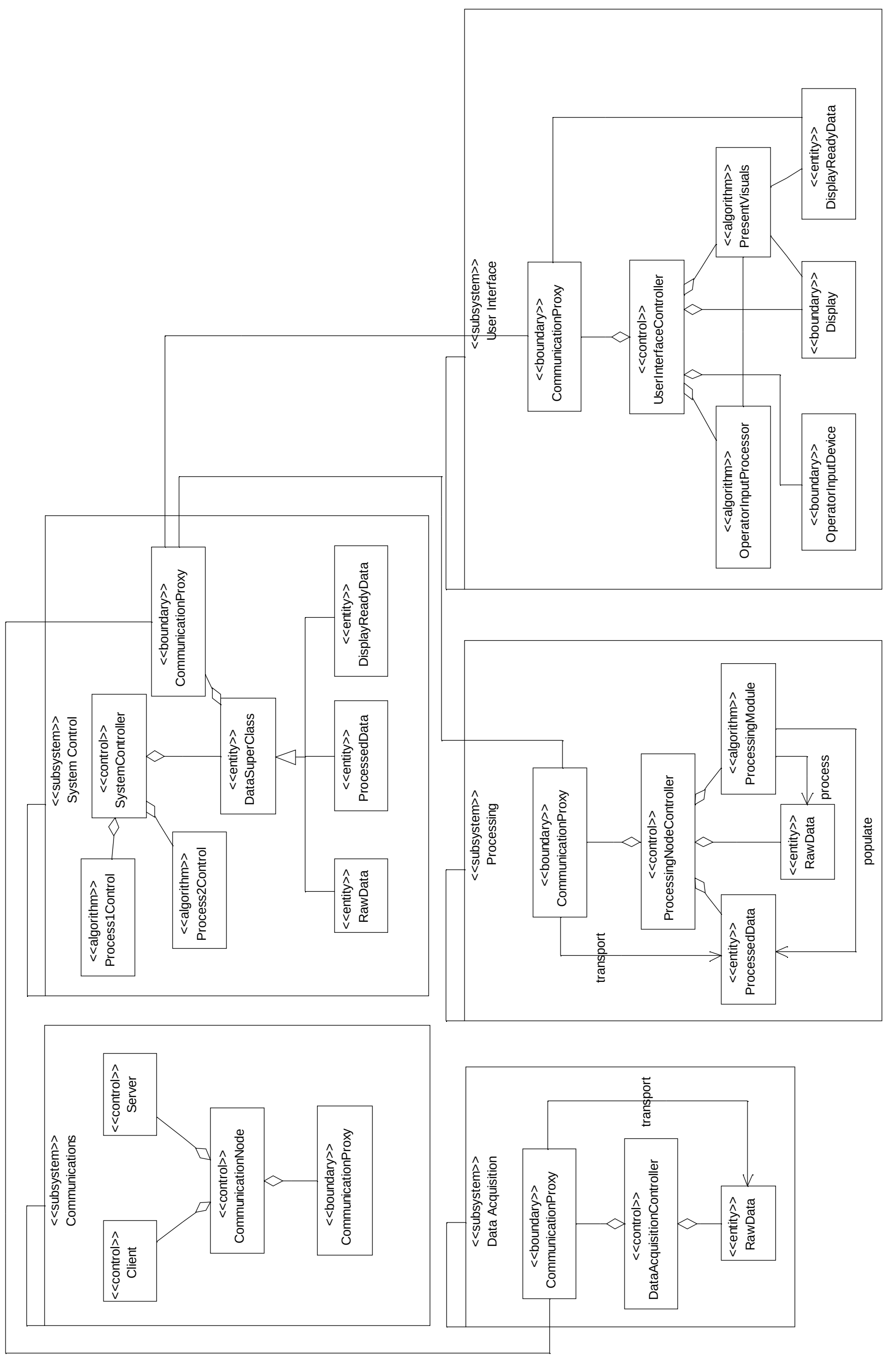


Figure 2-7 Artifacts comprising the design model in the USP.



<<subsystem>> User Interface
<<boundary>> CommunicationProxy
<<control>> UserInterfaceController
<<algorithm>> PresentVisuals
<<entity>> DisplayReadyData
<<boundary>> Display
<<algorithm>> OperatorInputProcessor
<<boundary>> OperatorInputDevice
<<subsystem>> System Control
<<boundary>> CommunicationProxy
<<control>> SystemController
<<entity>> DataSuperClass
<<entity>> DisplayReadyData
<<entity>> ProcessedData
<<entity>> RawData
<<algorithm>> Process1Control
<<algorithm>> Process2Control
<<subsystem>> Processing
<<boundary>> CommunicationProxy
<<control>> ProcessingNodeController
<<algorithm>> ProcessingModule
<<entity>> RawData
<<entity>> ProcessedData
process
populate
transport
<<subsystem>> Communications
<<control>> Server
<<control>> Client
<<control>> CommunicationNode
<<boundary>> CommunicationProxy
<<subsystem>> Data Acquisition
<<boundary>> CommunicationProxy
<<control>> DataAcquisitionController
<<entity>> RawData
transport

Figure 2-8 Subsystem decomposition class diagram

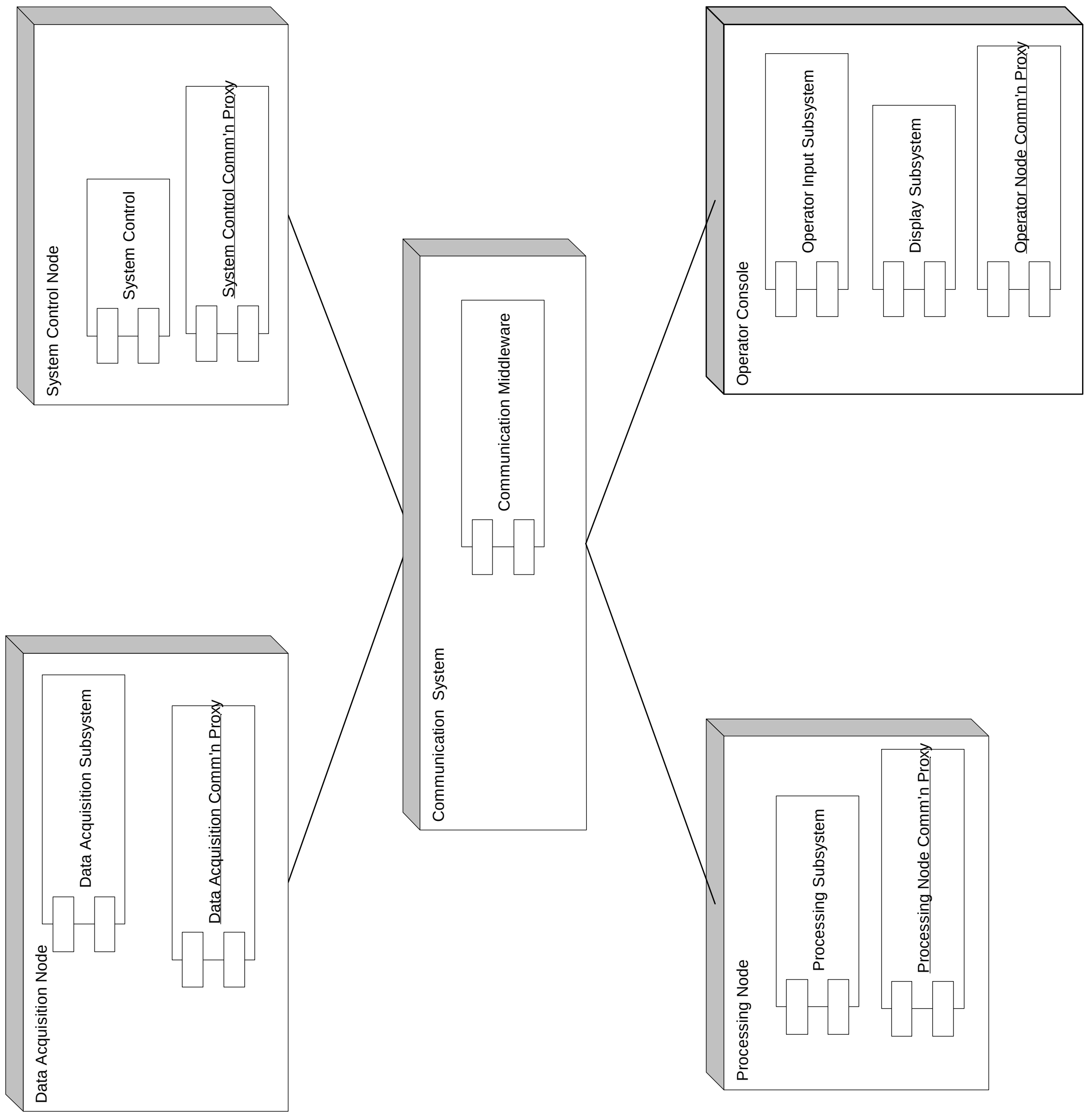


Figure 2-9 Design model deployment diagram

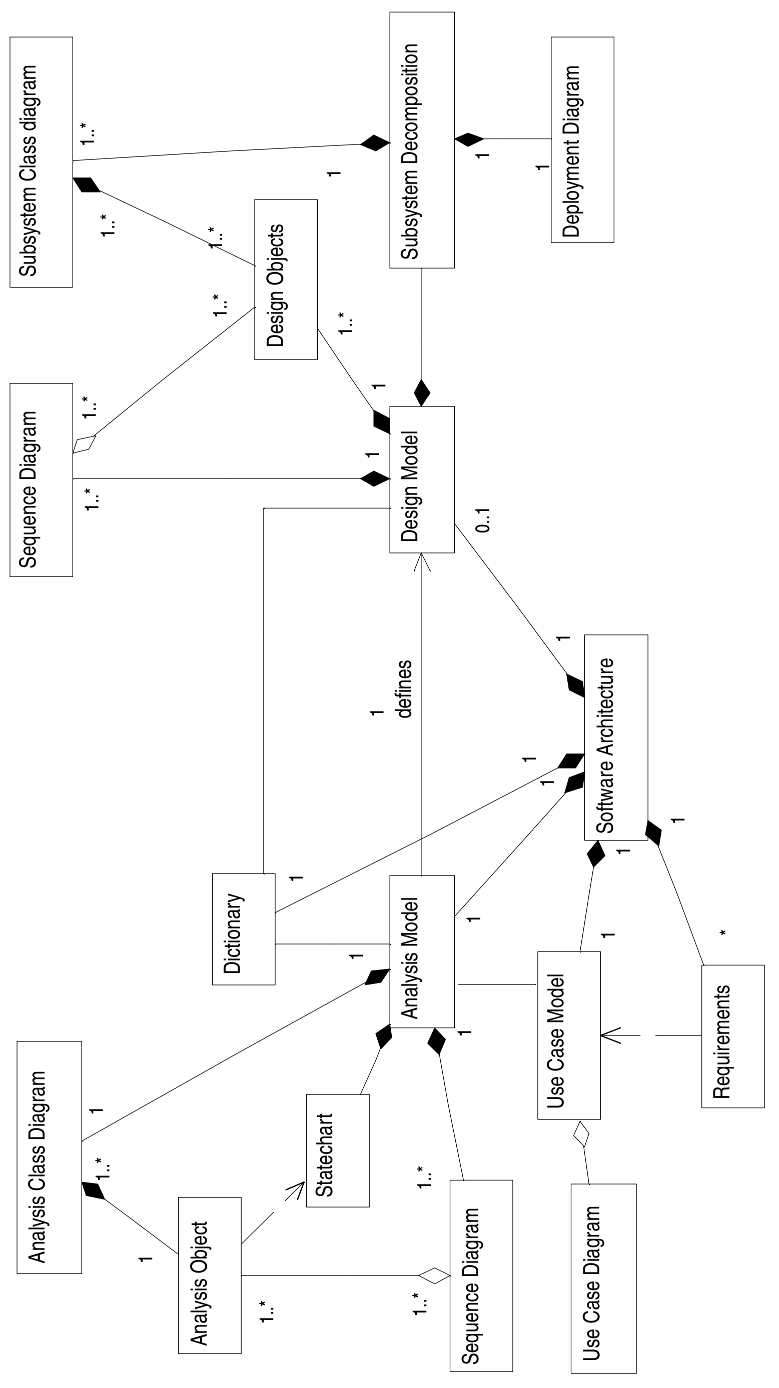


Figure 2-10 Modelling elements used to describe a software architecture

Chapter 3

Current Practice of Software Product Line Engineering

Most new methods are built upon existing processes and models. In this chapter, current software product line engineering practices are examined. The goal is to expose weaknesses from which the basis for a new process can be identified, and, to highlight strengths that can be carried forward into the new methodology.

3.1 FODA - Feature-Oriented Domain Analysis

The Feature-Oriented Domain Analysis (FODA) methodology [14] resulted from an in-depth study by the Carnegie Mellon Software Engineering Institute (SEI) of other domain analysis approaches. FODA focuses on the concept of a feature - an aspect of a system as perceived from the user's point of view. Successful applications of various methodologies pointed towards approaches that focused on the processes and products of domain analysis. FODA processes are:

- establishing methods for performing a domain analysis
- describing the products of the domain analysis process
- establishing the means to use these products for application development

The FODA methodology was founded on a set of modelling concepts and primitives used to develop domain products that are generic and widely applicable within a domain. The basic modelling concepts are [14]:

- *Abstraction* - used to develop a domain architecture from the specific applications in the domain. This architecture abstracts the functionality and designs of the applications in a domain, generalizing "factors" that make one application different from other related applications. The FODA method advocates that applications in the domain should be abstracted to the level where no differences exist between them. This is done to expose the underlying common architecture, or, if one doesn't exist, to facilitate its development.
- *Refinement* - used to both refine the domain architecture back into applications.

> Specific applications in a domain are represented as refinements of the domain architecture by using the general abstraction as a baseline and selecting among alternatives and options to develop the application (i.e., those factors that have been abstracted away must be made specific and reintroduced).

Application abstraction / refinement is accomplished by using the modelling primitives of: aggregation/decomposition, generalization/specialization and parameterization. The FODA method applies the aggregation and generalization primitives to capture the commonalties of the applications in the domain in terms of abstractions. Differences between applications are captured in refinements.

An abstraction can usually be refined (i.e., decomposed or specialized) in many different ways depending on the context in which the refinements are made. Parameters are defined to uniquely specify the context for each specific refinement. The result of this approach is a domain product consisting of a collection of abstractions and a series of refinements of each abstraction with parameterization. Understanding what differentiates applications in a domain is most critical since it is the basis for abstractions, refinements, and parameterization.

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 (Figure 3-1). These models are used to develop applications in the domain:

- The context model is used by a requirements analyst to determine if the application required by the user is within the domain for which a set of domain products is available.
- The feature model identifies mandatory, alternative, and optional features. The feature model is a better communication medium since it provides this external view that the user can understand.
- The entity-relationship model can be used by a requirements analyst to acquire knowledge about the entities in the domain and their interrelationships. An understanding of the domain will help the analyst to deal with the user's problems.
- The analysis can determine if the functional model, consisting of the data-flow model and the finite state machine model of the domain products, can be applied to the user's problems to define the requirements of the application. If the user's problems are all reflected in the feature model, then the requirements may be easily derived from the models. Otherwise, new refinements of the abstract components may have to be made.
- The architecture model is used by the designer as a high-level design for the application. If the user's problems are reflected in the feature model, a design may be easily derived from the architecture model. If the problems are not represented, then

the architecture model should be further refined from the other domain products

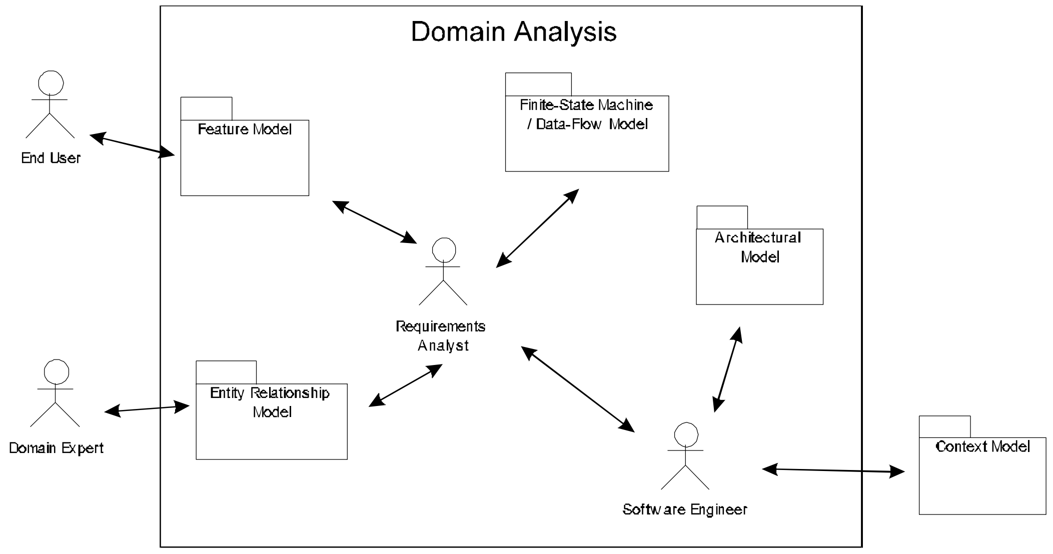


Figure 3-1 Use of Domain Analysis Products in Software Development [14]

3.2 FAST - Family-oriented Abstraction, Specification, and Translation

FAST was developed at Bell Laboratories (Lucent) in an attempt to balance the requirements for rapid production of software and the ever-present need for careful engineering. In essence, it is a pattern for software production based on three sub processes:

- *Qualifying the domain* - families which are deemed important enough to the business to warrant further investment are identified;
- *Engineering the domain* - infrastructure for the purpose of generating product family members is developed;
- *Engineering applications* - using product generation infrastructure to produce family

members rapidly.

Figure 3-2 illustrates the relationship between these three sub processes.

3.2.1 Domain Engineering and FAST

The purpose of domain engineering in the context of FAST is to make it possible to generate members of a family. To accomplish this, domain engineers must [1]:

- Define the domain (or family of products)
- Develop a language for specifying the family members (the application modelling language)
- Develop an environment for generating family members from their specifications
- Define a process for producing family members using the environment

The artifacts produced during the FAST domain engineering process include:

- An economic model of the domain
- A definition of the family identifying standard terminology and any assumptions that characterize the commonalties and variabilities of individual family members (see the following section for a further understanding of commonality and variability analysis)
- A description of the decision model for the domain
- The tools, code libraries and documentation required to build and use an application engineering environment

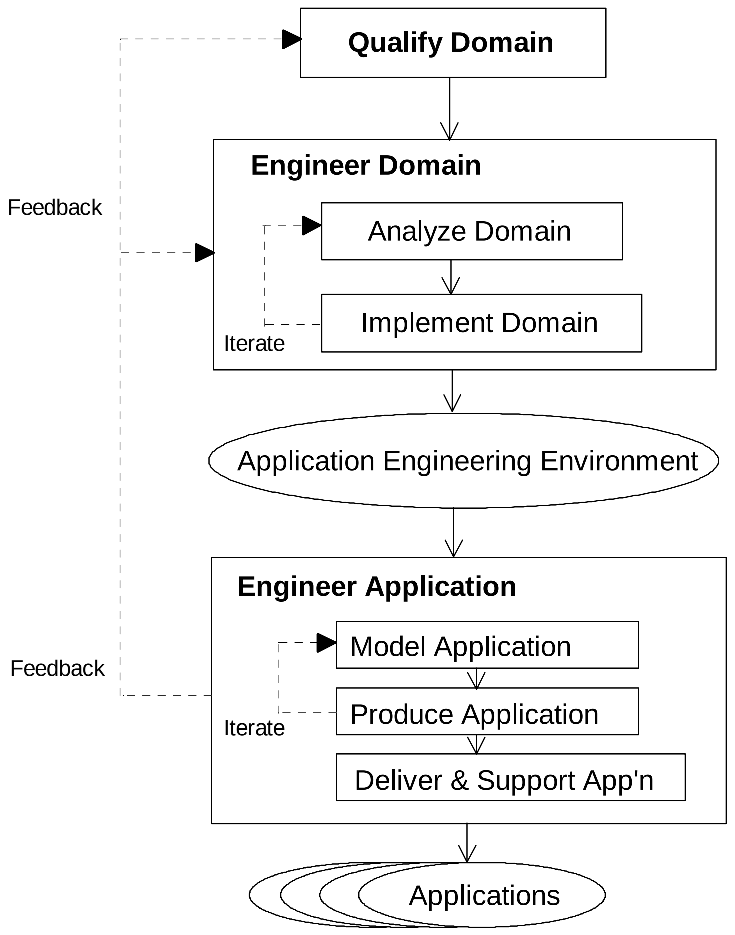


Figure 3-2 The FAST process pattern [1]

3.2.2 Commonality and Variability Analysis in FAST

A central element of the FAST domain engineering process is the commonality analysis. This analysis contains a list of assumptions that are true for all family members. These assumptions form the set of requirements that hold true for all members of the product line. Also part of this analysis is an identification of product variabilities - the aspects that will vary from product to product in the family - and the range of values for each. Table

3-1 summarizes the artifacts generated by the commonality analysis. Refer to [1][16][17] for a detailed discussion of the FAST Commonality Analysis process.

Artifact	**Description**
Dictionary of terms	A standard set of key technical terms used to describe the product family and its members
Commonalities	A structured list of assumptions that are true for all members of the family
Variabilities	A structured list of how family members may vary
Parameters of variation	Quantification of the variabilities, specifying the range of values for each one
Issues	A record of the alternatives considered for key issues that arose while analyzing the domain

Table 3-1 Artifacts generated during the FAST Commonality Analysis

3.2.3 Application Engineering and FAST

The purpose of application engineering in FAST is to quickly explore the space of requirements for an application and then to generate the application with the infrastructure developed during domain engineering. The idealized FAST application engineering process consists of analyzing a customer's requirements for a product line member, engineering and generating the application, delivering the code and documentation to the customer for validation and acceptance, and providing sustaining support. The artifacts generated or refined during this process include: a model of the application, code for the application and support documentation. FAST recognizes that the application engineer and the customer rarely establish satisfactory requirements on the first try, thus engineering the application becomes an iterative process that makes heavy use of the analysis and generation tools previously developed.

A key component of the FAST application engineering environment is the application modelling language that is used to specify family members. FAST does not specify a particular language, but instead leaves the details to the domain engineers and system architects.

3.3 KobrA [29][30]

The KobrA method - an object-oriented version of PuLSE[25][27][28] - revolves around component-based software engineering: the development of work products - interfaces, subsystems, use cases, classes, templates and test cases, etc. - that are designed to be reusable [7]. The thinking behind KobrA is that component-based systems within a given domain will share many similarities and will use many of the same components. The variabilities between systems in a family will thus likely revolve around a relatively small number of critical components. Instead of assembling every system in the family from scratch, KobrA builds a framework which hardwires the common aspects of the family, and allows the variable components to be plugged-in as needed.

Work products are oriented towards the description of individual components. This means that, as far as possible, there are no global or system-wide assets. Instead they are defined to carry information only related to their particular component. The intention is to allow components to be separated from their development environment and be more readily reused independently. Figure 3-3 illustrates these concepts. .

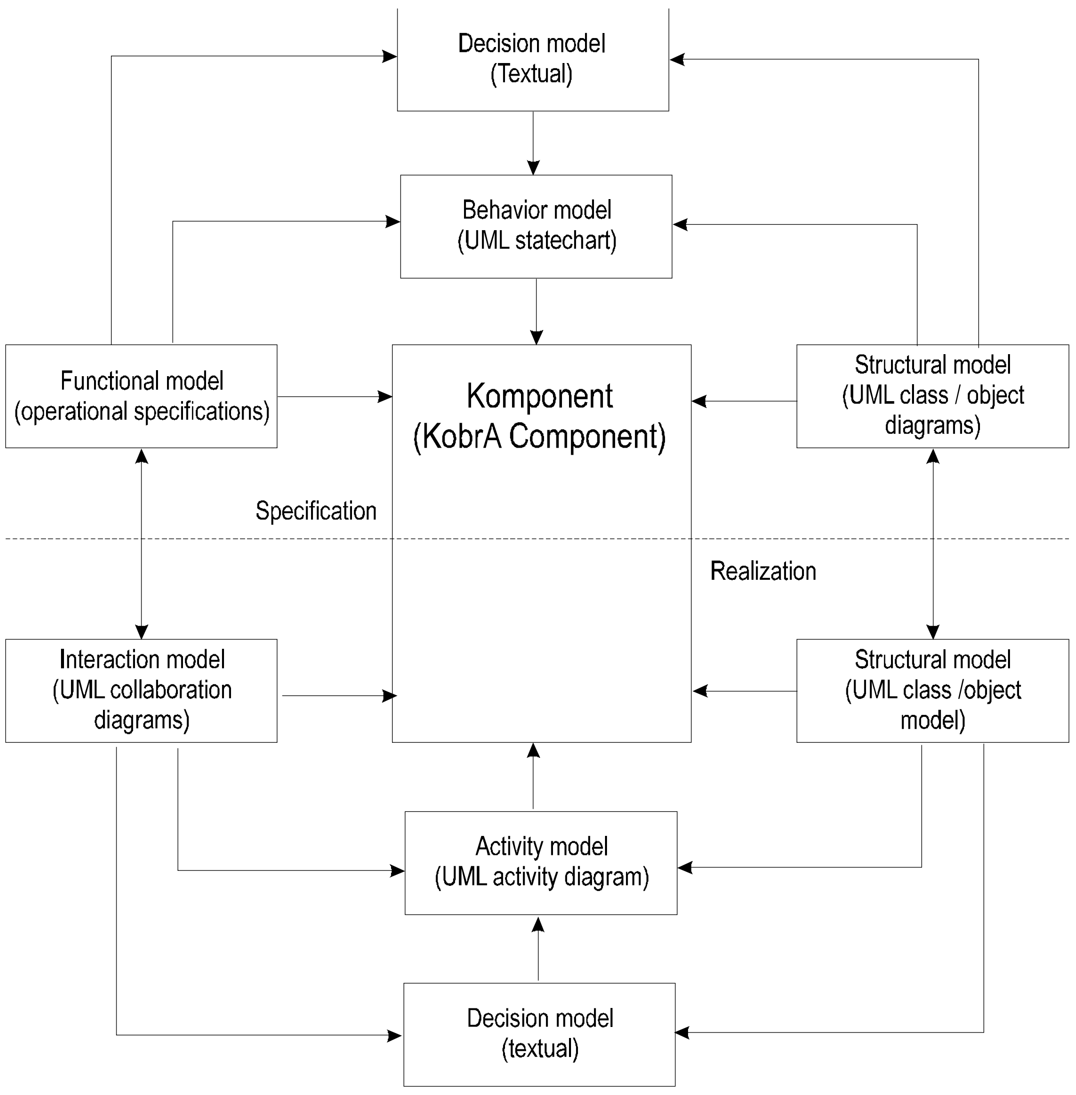


Figure 3-3 Artifacts used for the specification and implementation of a KobrA component

KobrA has fully embraced the UML in documenting the work products created in the development of components, eliminating the need to learn new notations and build tools to support the processes. It also employs some aspects of the USP focusing on microscopic rather than macroscopic development.

The KobrA method is broken down into two constituent sets of activities: framework engineering and application engineering. The purpose of the framework engineering activity is to create, and later maintain, a generic framework that aggregates all product variants that make up the product family, including information about their common and disjoint features. Application engineering activity instantiates this framework to create individual members of the product family. The goal is to use a single framework to instantiate multiple products / applications.

Chapter 4

Problem Definition

The methods in the previous sections have been proven to assist software architects in understanding product domains, extracting and re-engineering the architecture of existing systems, and in presenting these concepts in standardized models. Unfortunately, most of these methodologies have either not adopted modern modelling techniques. Those that have embraced change have chosen to ignore other developments in software engineering such as the importance of architecture-centric development. With the rapid adoption of object-orientation and its associated modelling languages, these tried and true processes need to be revisited to see if some currency can be added.

4.1 Perspectives

Each of the approaches in Chapter 3 has demonstrated their value in supporting the development, maintenance and evolution of many diverse product lines.

4.1.1 Comments on FODA

FODA mainly distinguishes itself by its feature analysis, whose purpose is to capture, in a model, the end user's understanding of the capabilities of applications in the target domain. However, many of its processes and products are from the non object-oriented (OO), functional decomposition world, making it less straightforward to develop OO product family architectures. With the general adoption of the modern software notations within the software community, most of the artifacts of FODA can be instantiated with UML and merged with the processes in [8] to extend FODA in an OO context.

With that said, there is still one aspect of FODA that does not have an equivalent in OO modelling techniques - the feature model. Although there have been examples of a one to one mapping of the feature types to an OO context, such as in [21], incorporating the notion of a feature as defined previously affords the opportunity to leverage the USP to gain an understanding of the dynamics of a feature not otherwise possible with a simple hierarchical representation.

Languages such as the UML place a strong emphasis on capturing the user's requirements, but bury them inside constructs such as use cases. We need a mechanism for drawing more of the software developer's focus towards the final objective of any development project, the expectations and perceptions of the user.

4.1.2 Comments on FAST

FAST is as much a pattern for processes as it is a set of processes. Although this makes it very flexible, this also makes it prone to misinterpretation when instantiated. Modern software development demands that processes be more of a cookbook than a somewhat more ambiguous design pattern.

FAST also centers on the creation of a whole suite of application generation tools. Presumably, these tools are custom-built by the development team, which means that they must also support them. This can seriously defocus the development team from their primary function: application development. Third-party vendor tools are always preferable when available because they are focusing on their primary function: tool development.

There are two aspects of FAST that would appear to have some direct applicability to developing a generic method of describing software product lines. First, the commonality analysis process appears to be extremely useful: it is sufficiently process-agnostic allowing it to be ported to other methods. Second, the concept of an application modelling language permits developers to use open standard notations, such as UML, which are supported by third-party tools complete with code generation capabilities.

4.1.3 Comments on KobrA

KobrA has made significant advances in bringing the UML, and to a lesser extent, the USP to software product line engineering. It has leveraged this de facto industry standard notation to make its daily use more readily adoptable. It also advocates the use of many object oriented techniques, especially frameworks. Having system and application level behaviors encapsulated by a framework permits software engineers to adorn a prescribed architecture with application-specific components. The intent is to promote large-scale reuse.

Unfortunately, KobrA has chosen to significantly reduce the role of software architecture, basically hardwiring the product line's architecture very early on in the process. But how many times have we seen a software project run into severe difficulty due to the absence of an architecture altogether, or the presence of one that is not sufficiently malleable. This diminution of the role of architecture brings with it risk of increased, not decreased, development risk. Further, the view of the product line at the architectural level is what provides us with the best perspective of the product family evolution road map.

4.2 Problem Definition

Table 4-1 lists strengths of each of the product line methodologies examined previously which can be used to support a new process for modelling software product lines, and, deficiencies which any new process should address.

Methodology	Re-usable Aspects	Deficiencies
FODA	• Feature based, providing a user perspective of the product domain • Well-established domain engineering processes and work products	• No formal definition of a feature from a modelling perspective. • Does not specifically address architecture • No formal processes for migrating domain analysis to product lines
FAST	• Well understood commonality and variability processes	• Designers spend a lot of time developing tools. • Application generation language not defined • Not object-oriented
KobrA	• Incorporates object-oriented notations and processes	• Role of architecture not adequately addressed

Table 4-1 Summary of current software product line practices

In this work, a feature-based, object-oriented implementation of a process to extract and extend a software product line architecture will be developed. To accomplish this, it will:

- Present an object-oriented version of FODA's feature model using UML and integrate it into the Unified Software Development Process.
- Provide a mechanism for recovering architectures from base products, evolving those architectures to the product domain, product line and products.
- Use non-proprietary software tools.

There are several reasons for developing this new approach:

- It has its basis in processes that have been demonstrated to be of value (SEI's FODA, FAST and the horseshoe re-engineering model), but have not, as yet, been formalized with newer technologies such as Object-Oriented Analysis and Design (OOAD) and the Unified Modelling Language (UML). By leveraging the experiences gained with these older technologies, we short-circuit the need to "reinvent the wheel" in terms of underlying processes and give them currency that might not otherwise be obvious.

- Legions of software developers are being trained with a solid understanding of the Unified Software Process (USP). Basing the extended processes on the USP further leverages skills of designers, reducing even further their "learning curve", accelerating their productivity.
- 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. The method can readily integrate in such an organization.
- Designing a process within the scope of the existing range of computer aided software engineering (CASE) tools produces a process that is more readily usable "out of the box".

Chapter 5

FOOM - Feature-based Object Oriented Modelling

FOOM is an extension of SEI's FODA[22] and the elaboration of the Unified Software Process described in detail in [8][11]. These processes and methodologies have proven to be effective in their respective focus areas of domain analysis and object-oriented analysis and design. FOOM represents a model that combines complementary aspects from each into a process and a set of artifacts suitable for modelling software products based on a family approach.

5.1 Expanding Feature-based Modelling

Languages such as the UML place a strong emphasis on capturing the user's requirements, but bury them inside constructs such as use cases.

Recall that, for this work, features are used to group many requirements and their ensuing design artifacts into a single entity [26]. From a modelling perspective, a feature could include a primary use case plus its <<extends>> and << includes>> counterparts, associated analysis and design objects, their associations, and, model elements such as diagrams and entries in the data dictionary (Figure 5-1). This provides a very powerful mechanism for capturing the essence of a system at a level of abstraction above that of a standard USP / UML model.

It is easy to think of a feature as an autonomous, atomic elements of a software system [26]. However, experience tells us that, in any nontrivial system, this is not the case. Looking at a system from the feature level provides a macroscopic view of its static and dynamic structure.

From a product line perspective, a feature can be considered to be an architectural pattern taken from several instances of a product family's siblings. As will be shown in subsequent sections, these patterns take shape as a products' commonalties and variabilities are discovered.

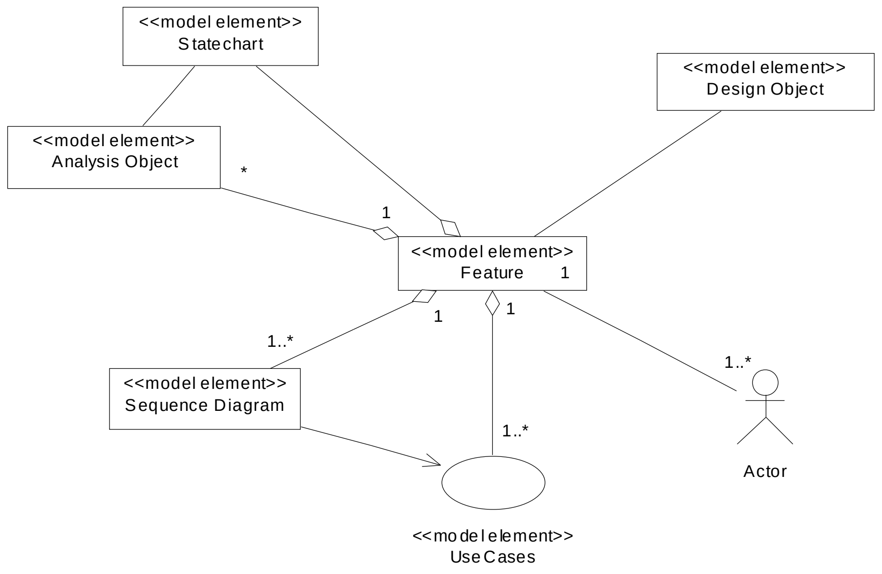


Figure 5-1 A feature encapsulates structural and dynamic aspects of a product

5.1.1 Building a Feature Model Based on FODA

FODA is the progenitor of most modern domain and analysis modelling methodologies. It builds on three fundamental sub processes: domain analysis, feature analysis and feature modelling. Domain analysis focuses on identifying a product that is believed will form the kernel of the product line. It also lists current and future sibling and descendant products. 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 results of feature analysis provide the constituent artifacts to populate the feature model.

5.1.2 An Object-Oriented Perspective of the Feature Model

The first building block of the feature model is a feature class. Considering FODA's feature model, the feature class is specialized into three subtypes:

- *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 or 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 relate to how the features are assembled and the rules that guide their merger into a product. Packages are used to consolidate Products and Features. On a first pass, aggregation can be used to illustrate the composition of products with multiplicity implying a sense of required, alternative and optional. [21] introduces the use of stereotyping associations to further refine the aggregation rules. Figure 5-2 illustrates these relationships.

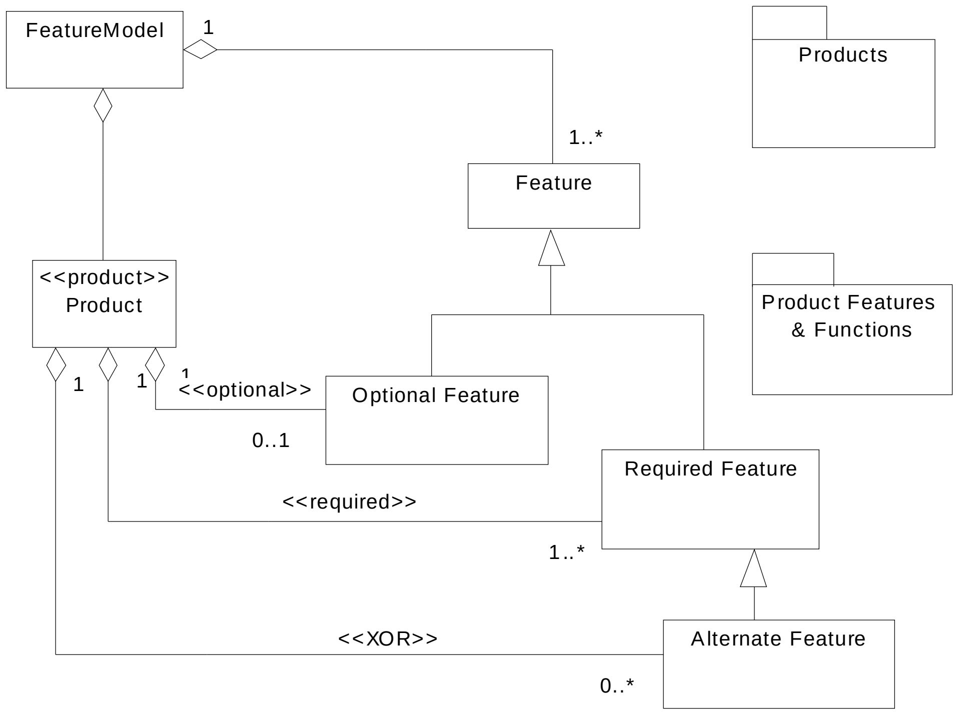


Figure 5-2 Building the feature model with basic aggregation and multiplicity techniques

5.1.3 Adding Precision to the Feature Model

Although these basic constructs are sufficient to build a complex feature model, the maintenance efforts as the product line evolves could be substantial. The model can be simplified with the introduction of a *feature list*, which is a contract between the product and the feature set of the product family. It becomes 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 ongoing maintenance and keeps the model readable. Figure 5-3 illustrates the relationship.

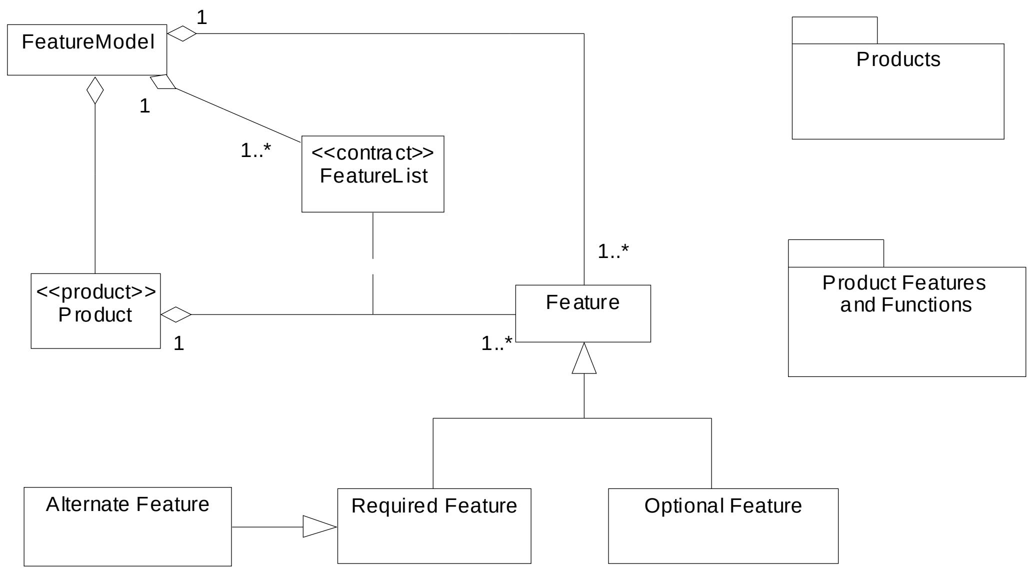


Figure 5-3 Using a contract to simplify the relationship between the Products and Features in the Feature Model.

Start 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->includes( AFeature1) and
      self.AlternateFeaturesList->includes( AFeature2) and
      ...
      self.AlternateFeaturesList->includes( AFeatureN);

      self.OptionalFeaturesList->includes( OFeature1);
end contract;
```

The inheritance relationship between the product line and family members' contracts is shown in Figure 5-4. Use of enumeration of products and features and Boolean expressions allows for a very concise definition of an individual product's features.

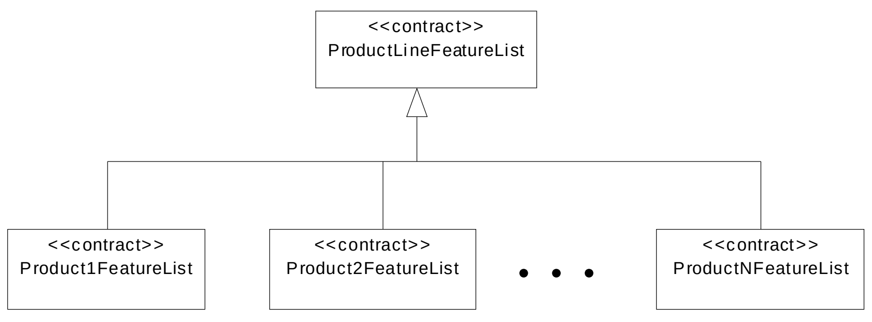


Figure 5-4 Relationship between product line and product feature contracts

5.1.4 Feature Discovery and Propagation

When building the Feature Model, it may not be obvious exactly how a product's features are identified. Work on the application described in the next chapter provided a means of formalizing this process.

The first pass at feature discovery started with the user-visible behaviors and outcomes[3]. Commonality and variability analysis of the system behaviors and attributes served to subdivide the features. One set was still directly visible to the user - that is observable via one of the system "boundaries" - and also served to distinguish one product from another. Work on the sample application gave rise to the notion that the features discovered in domain analysis would become the use cases around which the target architectures would be built.

Another set of features was found to be common to all or most of the products, but they were not directly visible to the user. These features were reclassified as *functions* as they provide the infrastructure on which the features operate or "function". They also serve to aggregate sub components of different features. When building target architectures, there was a strong correlation between them and the "actors" in the analysis. For this reason, and their correlation with the features, they remain part of the feature model.

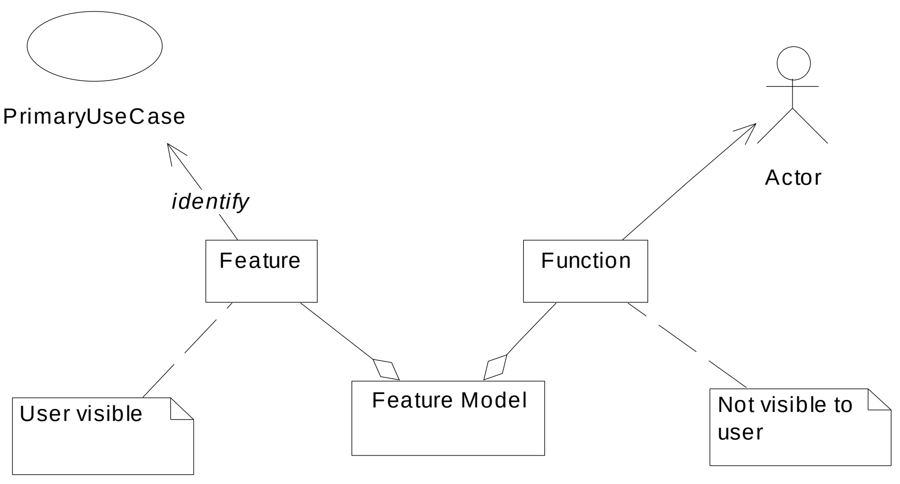


Figure 5-5 Correlation of the features and functions to the USP Analysis Model

5.1.5 Integrating the Feature Model

Now that the internal structure of the Feature Model has been explored, its relationship to the other parts of USP must be determined. [20] discusses the relationship of the feature model from a stakeholder perspective: the feature model is a mechanism for refining the requirements elicited during analysis and is related to the design assets as illustrated in Figure 5-6a. Figure 5-6b extends this view, incorporating the constituent artifacts of the object model.

Next, recall the UML description of a feature (Figure 5-6c). Examination shows a correlation between its constituents and the USP Use Case, Analysis, and design Models. Building on the relationship between features and the object models (analysis, design) , it was concluded that the feature model bridges the analysis and design models. Features populate these models (Figure 5-7), from which modelling continues.

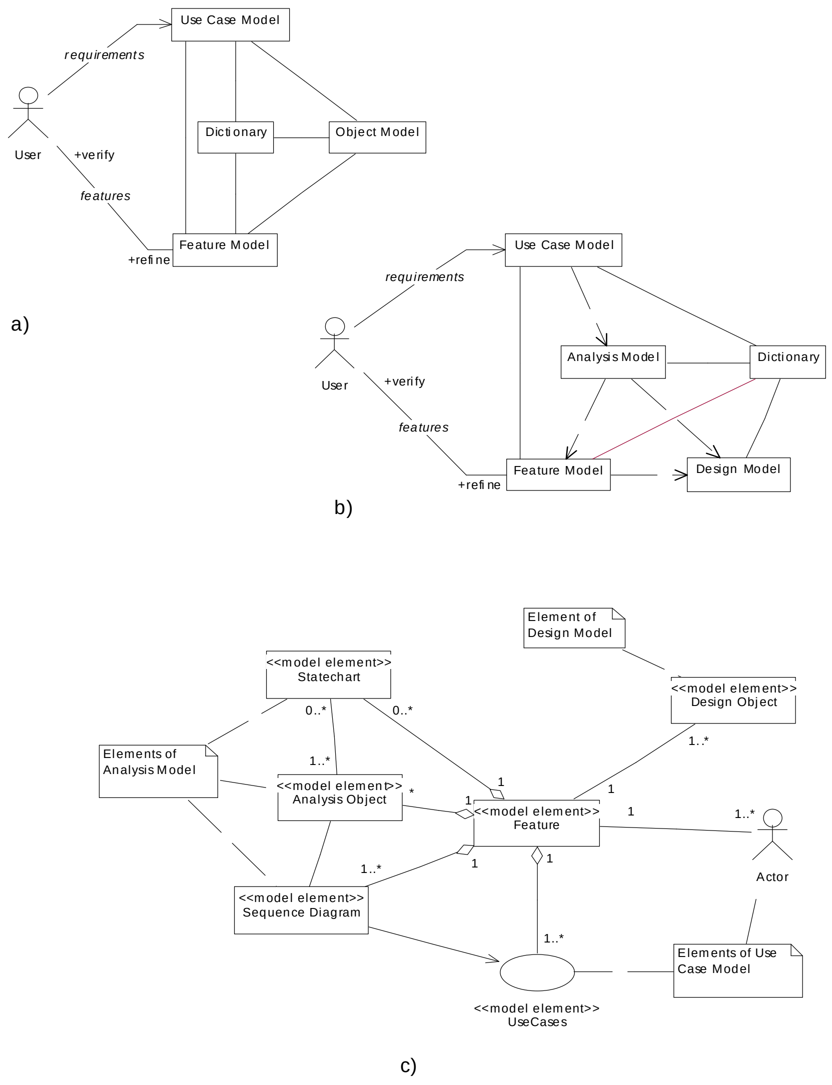


Figure 5-6 The feature model bridges the analysis and design models.

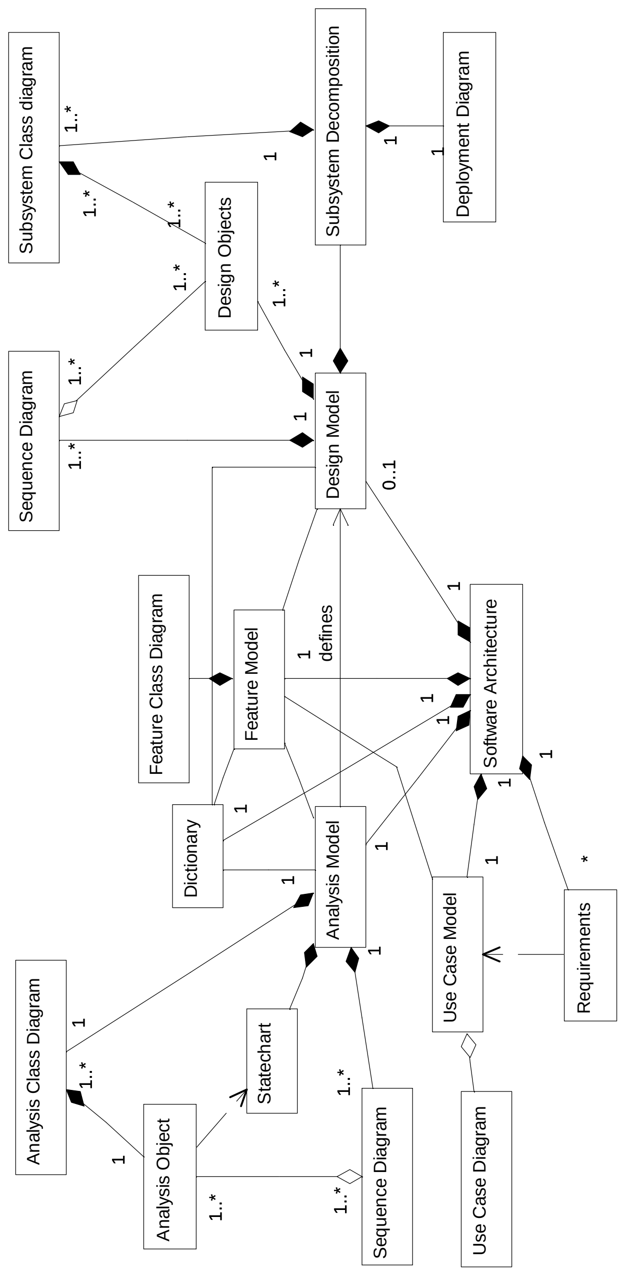


Figure 5-7 Relationship between the Feature Model and the Analysis and Design Models

The resulting model (Figure 5-8) is also multidimensional, and, still 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 also build the design model. Similarly, the feature and design models may evolve independent of analysis. However, this should not be done until individual product architectures are being developed from the product line architecture.

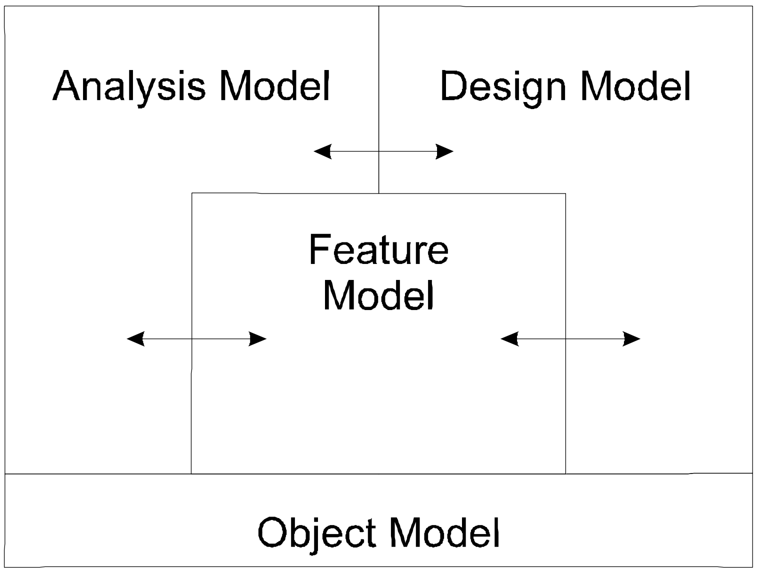


Figure 5-8 Multidimensional relationships between the analysis, feature and design models in FOOM.

5.2 Architecture Transformation and Evolution

The transformation process for FOOM is based on the horseshoe re-engineering model and its latest form CORUM II [19]. It involves re-engineering an architecture from an abstract level - plans and specifications rather than code - giving rise to a product architecture that can be transformed to a new paradigm. FOOM builds on this approach, focusing on migrating architectures from the base product through to domain, product line and product architectures from the conceptual level only.

There is an assumption that there exists a base product from which downstream architectures. If that is not the case, then the domain architecture will have to serve as the starting point for the product line. It is also assumed that the base product's architecture is well defined. If that is not the case, an architecture must be extracted before the transformation can proceed.

Once a base product has been identified and its architectural assets deemed to be suitable, a series of transformations are performed to migrate the architecture, first to a domain architecture, sufficiently abstracted to represent all products in the domain. From there, variability analysis provides attributes that differentiate one product from another, leading to a product line architecture. The final step is to develop the rules for deriving a single-system architecture from the product line architecture. Figure 5-9 provides a pictorial description of this process. Figure 5-10 provides a UML perspective of the same logical transformation.

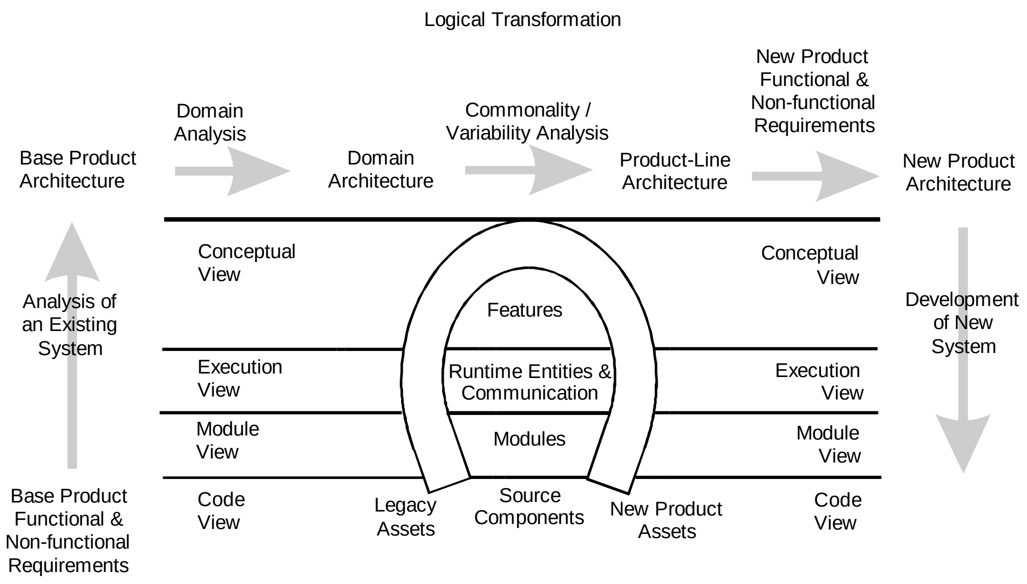

Logical Transformation
Base Product Architecture
Domain Analysis
Domain Architecture
Commonality / Variability Analysis
Product-Line Architecture
New Product Functional & Non-functional Requirements
New Product Architecture
Analysis of an Existing System
Base Product Functional & Non-functional Requirements
Conceptual View
Execution View
Module View
Code View
Features
Runtime Entities & Communication
Modules
Source Components
Legacy Assets
New Product Assets
Conceptual View
Execution View
Module View
Code View
Development of New System

Figure **5-9** Horseshoe model modified for software product line development.

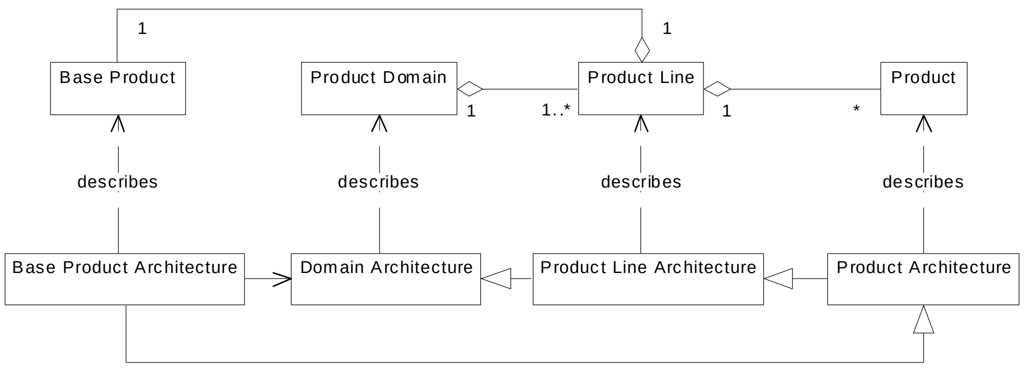


Figure 5-10 Relationships of the different stages of the architecture transformations

In the same manner that we use the USP to design software systems, a process model can be built. The use case diagram, as illustrated in Figures 5-11 shows each of the transformation processes as a use case; the participating objects, architectures, products and product features, are actors.

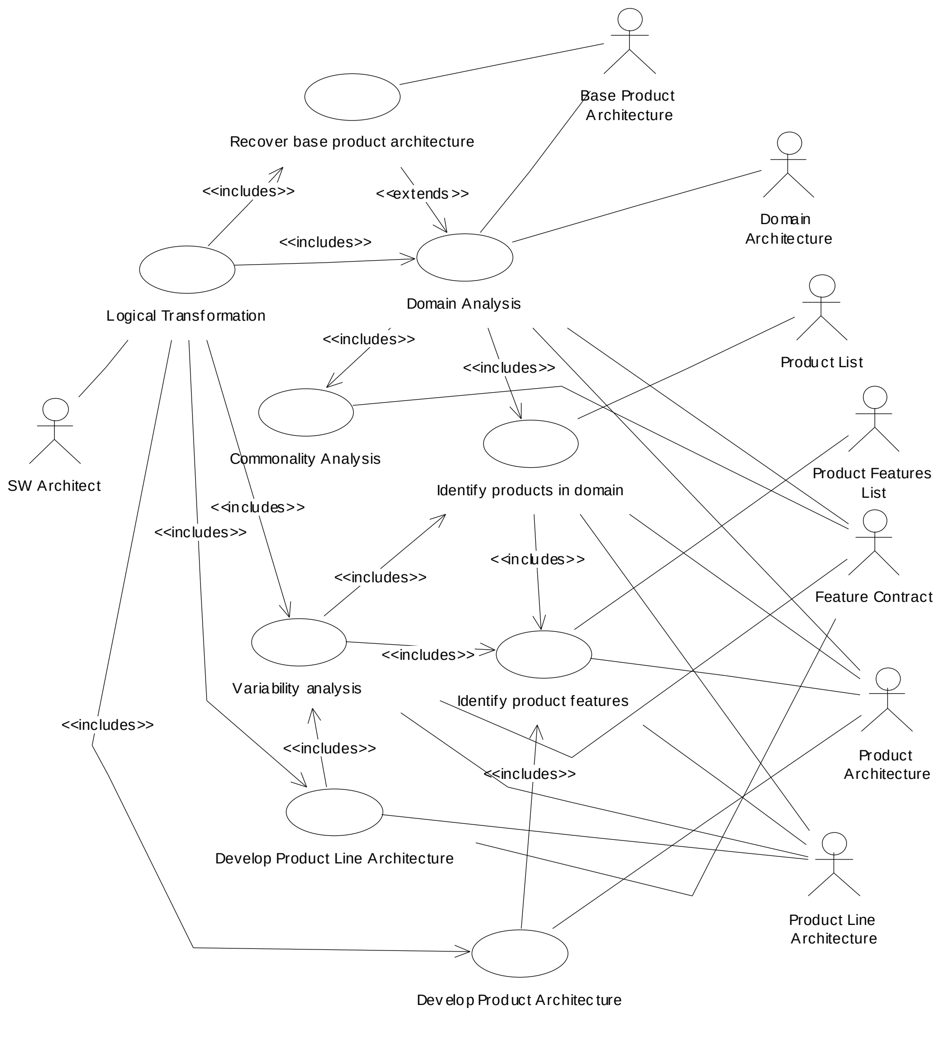


Figure 5-11 Use case model of the logical transformation from a base product to a product line architecture

5.2.1 Modelling Strategies

Due to the close relationship between features and use cases, it is not obvious which artifact should drive architecture development. Two modelling strategies are possible[14]:

- *Feature-driven development* is appropriate for mature organizations where domain experts, with experience in developing similar products are available. This permits exploration of architecture alternatives early in the product line development cycle. Feature-modelling focuses on the commonalties in a product line feature set and introduces variabilities as refinements. Use cases serve to identify and define the structure and behavior of the systems suitable for implementation by designers.
- *Use-case-driven development* is suitable in less mature organizations including projects of mature companies in a new domain, or situations where availability of domain expertise is limited. Use-case-modeling can serve to provide a product vision where one does not already exist or where it is not clearly defined.

There are several issues to consider in selecting a modelling strategy. One is the relationship between the base product architecture and its successors. Although it may be well understood (the USP is after all use-case-driven), the results of its transformation into the domain architecture is not clear at the beginning of the exercise. On the other hand, features are used to align the designer's focus with that of the user. This abstraction makes FOOM architecture-centric, providing a high level view of the product line and avoiding the loss of clarity where conceivably hundreds of use cases could eventually be required. FOOM also assumes domain expertise is available in the form of an existing base product and architects who have built similar systems.

A balance needs to be found between these opposing strategies. Work on the example provided a compromise solution. A use-case driven strategy is used to migrate from the base product architecture to the domain architecture. FOOM then uses a feature-driven strategy (Figure 5-12) for subsequent transformations..

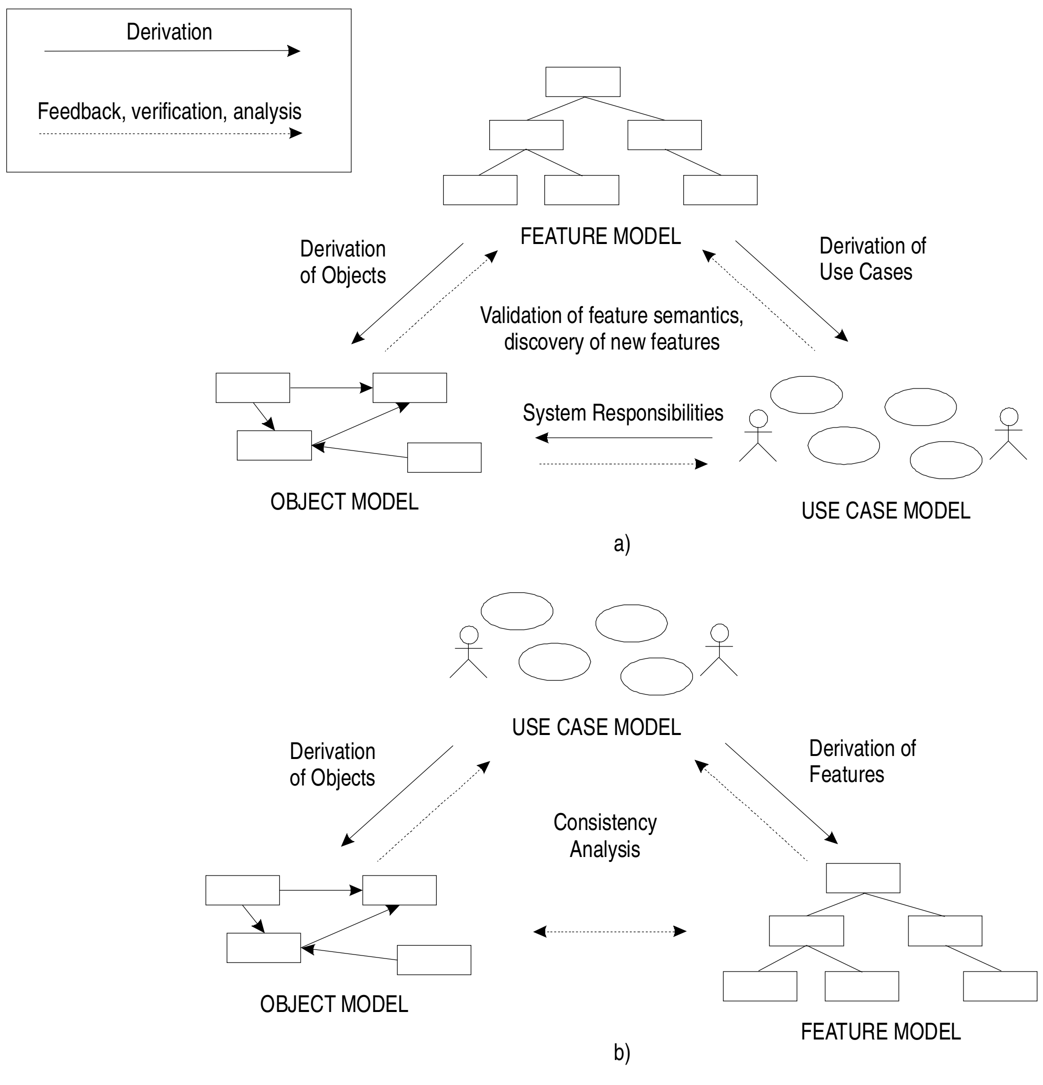


Figure 5-12 Modelling strategies used in FOOM [14]: a) feature-driven b)use-case-driven

5.2.2 Steps in the Architecture Transformation Process

Each of the steps in the architectural transformation can be documented as workflows in the same manner as use cases.

5.2.2.1 Base Product Architecture Recovery

Use case name	BaseProductArchitectureRecovery
Participating actors	SwArchitect, BaseproductArchitecture
Precondition(s)	• Domain experts have identified a product suitable to use as the base product • The architecture for the candidate base product is not adequately documented
Flow of events	1. Identify primary use cases from existing requirements documents 2. Elaborate each use case 3. Build analysis model (sequence and analysis class diagrams) 4. Build design model (subsystem decomposition and software deployment)
Postcondition(s)	Base product architecture defined

Table 5-1 Base product architecture recovery workflow

5.2.2.2 Domain Architecture Development

Use case name	DevelopDomainArchitecture
Participating actors	SwArchitect, BaseProductArchitecture, DomainArchitecture, ProductFeaturesList, ProductList
Precondition(s)	Product domain can be identified and bounded
Flow of events	1. Identify current and future products in the domain 2. Identify features of all products to be included in the domain. 3. Commonality analysis identifies functions 4. Top level features identify use cases 5. Elaborate use cases based on sufficiently abstracted features such that no difference between products exists. 6. Build domain feature model, identifying modelling

Use case name	DevelopDomainArchitecture
	elements associated with each feature. 7. Build the analysis model based on the model elements for each feature in the Feature Model 8. Refine the analysis model to accommodate relationships that arise as a result of multiple feature inclusion 9. Build and refine the design model based on the model elements from each feature in the Feature Model 10. Refine the design model to accommodate relationships that arise as a result of multiple feature inclusion
Postcondition(s)	Domain architecture developed.

Table 5-2 Domain architecture development workflow

5.2.2.3 Product Line Architecture Development

Use case name	DevelopProductLineArchitecture
Participating actors	SwArchitect, DomainArchitecture, Product Line Architecture, ProductFeaturesList, ProductList, Product Line Feature Contract
Precondition(s)	Domain architecture has been developed
Flow of events	1. Variability analysis identifies product features 2. Build the product line Feature Model from the domain features, using variabilities of each product to refine existing use cases and identify new ones. 3. Build product line and individual product feature contracts 4. Populate the use case, analysis and design models with components from each fetaure. 5. Refine the use case, analysis and design models to accommodate relationships that arise as a result of multiple feature inclusion
Postcondition(s)	Product Line Architecture developed

Table 5-3 Product line architecture development workflow

5.2.2.4 Product Architecture Development

Use case name	DevelopProductArchitecture
Participating actors	SwArchitect, Product Line Architecture, Product Architecture, Product Line Feature Contract, Product Line Feature Contract
Precondition(s)	Product Line has been developed
Flow of events	1. Build the product line Feature Model based on the feature contracts of the product line and the product. 2. Populate the use case, analysis and design models with components from each feature. 3. Refine the use case, analysis and design models to accommodate relationships that arise as a result of multiple feature inclusion
Postcondition(s)	Product Architecture developed

Table 5-4 Product architecture development workflow

5.3 Sum of the Parts

When the extensions of each of the supporting methodologies are aggregated in the new model, a new set of processes arises that generate product line architectures that are entirely object-oriented. The same types of artifacts are produced for each constituent architecture. A feature model at each stage of architecture development allows the evolution of each feature to be tracked. It also provides, in a single model, the ability to develop road maps for future evolution (Figure 5-13).

5.3.1 Traceability and Stereotyping

In order to provide a very clear traceability mechanism, a series of object stereotypes are used, giving a clear indication of the development stage to which an artifact belongs. The general form of the stereotype is:

<<*architecture* + *artifact*>>

where architecture is the current phase in the logical transformation process and artifact is the object type. Table 5-5 lists labels that are used in FOOM.

Label	**Examples**
architectures	base product, domain, product-line and product
artifacts	feature, function, entity, boundary, controller, subsystem,

Table 5-5 Examples of stereotype label components to provide traceability in software product family architecture development.

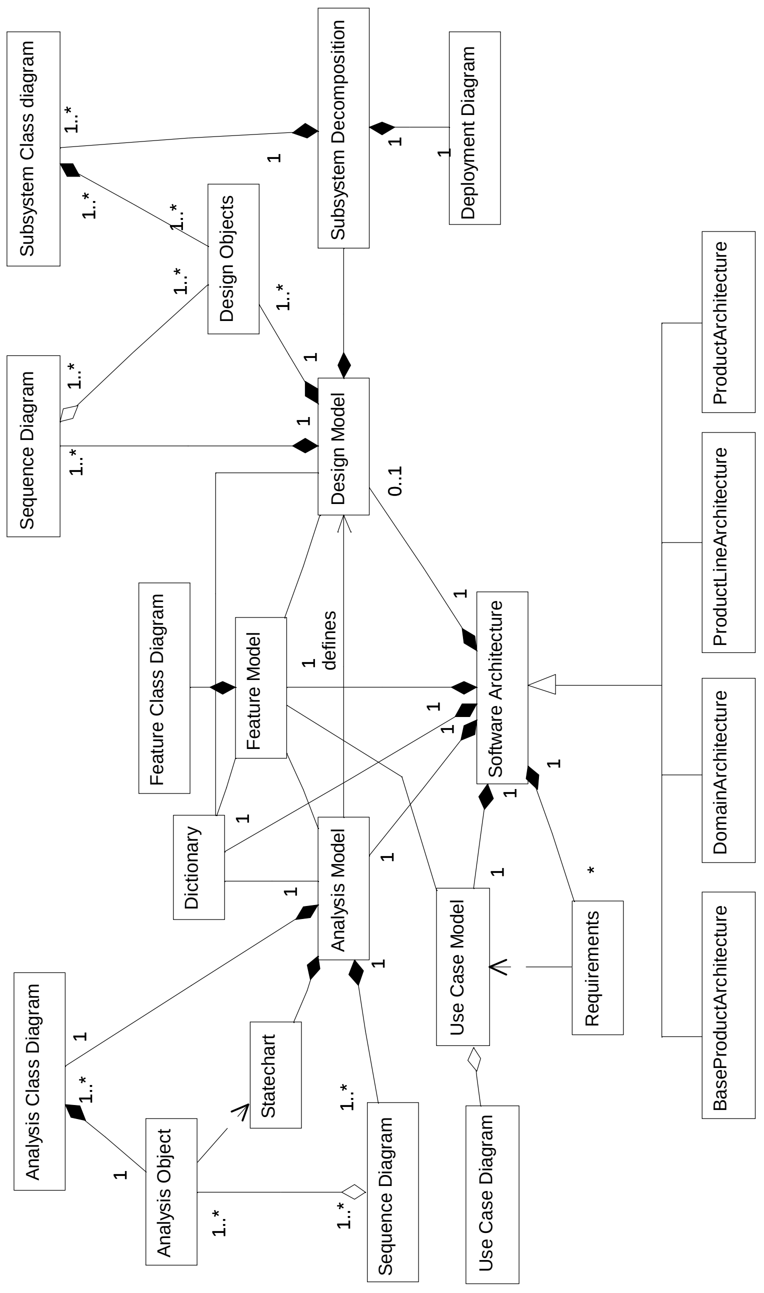


Figure 5-13 The same artifacts are used to describe the architecture at each step in the transformation process

Figure 5-14 shows the three dimensional nature of FOOM. There is a very close relationship between all the submodels with the object model. All submodels are internally consistent with their siblings in each architectural level and with their descendants.

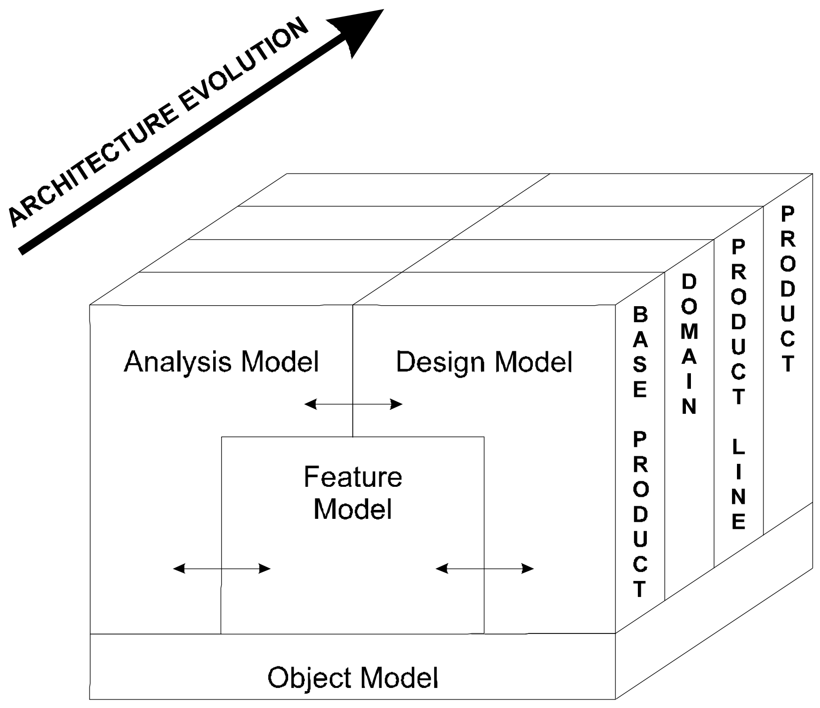


Figure 5-14 FOOM uses the same methodologies and artifacts, in a tightly coupled relationship, to model architectures at each step in the transformation process.

5.4 Summary

The development and evolution of software product family models has lagged far behind the adoption of object-oriented modelling techniques used for single systems. Fortunately, tried and true methods do not have to be scrapped; rather, they can be modernized by abstracting their more salient aspects and integrating them with modern methods. The model presented here does exactly that: it introduces the feature model as an integral component of a software architecture, provides mechanisms for migrating architectures and instantiating products. It also provides a view of the product line sufficiently abstracted to allow for the development of medium- and long-term evolution road maps.

Chapter 6

Model Application -

A Family of Sonar Systems

This chapter covers the step by step application of FOOM on an example application - a family of sonar systems. Its hypotheses in modelling the architectures of a complete product family are tested, validated and refined. Although none of these systems is currently in production, they are based on existing systems and future prototypes of military antisubmarine and speciality sonars.

6.1 What is Sonar?

Sonar is a system that captures transmitted and reflected sound in order to detect and locate underwater objects. An 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 sonars use 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 6-1 provides an overview of this process.

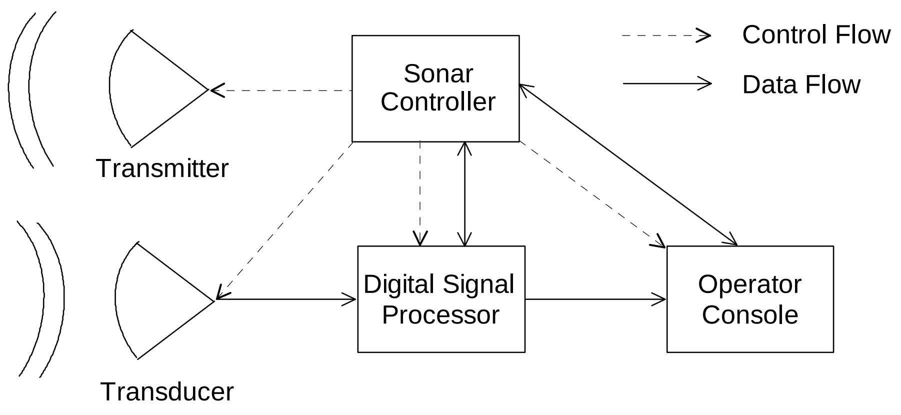


Figure 6-1 Sonar system overview

In this analysis, four types of systems which are deemed to be sufficiently diverse as to represent a good cross-section of the sonar domain are examined.

1. *Hull-mounted sonar (HMS):* An all-purpose type that does many tasks reasonably well, but excels at one: detect submarines in active mode (transmitting acoustic energy into the water and listening for echoes on a specified frequency). It is also capable of performing passive detection (listen only), using different digital signal processing (DSP) and display software than that used for active detection. The primary computing assets are a DSP, system control and an operator console. What characterizes this type of system is that its transmitter and transducer (sometimes referred to as the "wet end") are mounted in a housing external to the ship's hull, along the keel, about one third of the ship's length from the bow.
2. *Variable Depth Sonar (VDS):* This system incorporates essentially the same computing assets and functionality as an HMS. Its distinguishing feature is that its wet end is in a separate winch-controlled tow body which is pulled behind the ship. The reason for such a configuration is beyond the scope of this document. Suffice it to say that it is the only way to detect objects in the ocean below the prevalent thermal boundary layer at a depth of about 300 to 400 meters.
3. *Towed-Array Sonar (TAS):* This system is a passive type with computing hardware similar to active systems, but requiring only the software for passive DSP and displays. Its physical configuration is characterized by a winch-controlled string of hydrophones (underwater microphones) towed behind the ship (hence, towed array) in place of the transducer used in HMS and VDS.

4. *Mine Hunting Sonar (MHS):* Very similar to an HMS, this type of system operates at much higher frequencies than other active sonars. Additionally, its transmitter and transducer are located at the ship's bow, rather than the more amidship positioning of an HMS.

Figure 6-2 illustrates the various configurations of these products. In this example, the HMS, VDS and TAS will be considered as existing products. Mine hunting will be examined as a retrofit to the HMS and as a new product.

To support the development of the architectures for this product line, a typical systems level description of the Hull Mounted Sonar is provided in Appendix B.

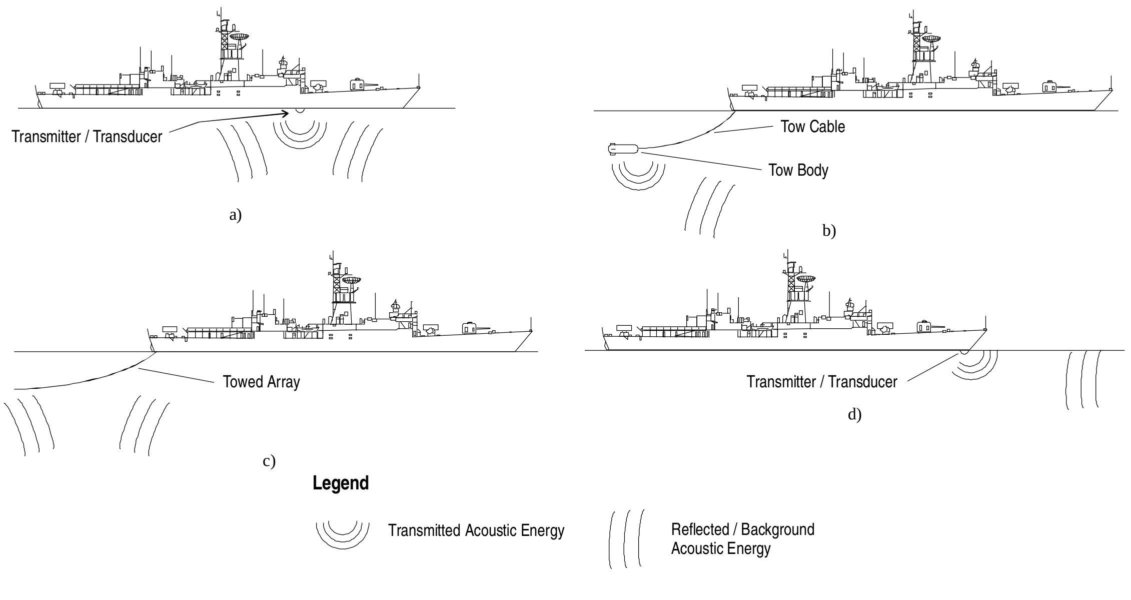


Figure 6-2 Types of sonar: a) Hull Mounted b) Variable depth c) Towed array d) Mine hunting

6.2 Applying FOOM to Sonar Systems

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 here, then some architecture extraction work is required. Such was the case with the system identified as the base product for our example - the hull mounted sonar. Once the base product architecture is defined, it is revisited to adapt it to our model. Extraneous use cases are removed, notations are modified to conform the traceability rules through stereotypes, and, a feature model is developed.

Domain analysis identifies current and future members of the product family and the features - user visible behaviors and attributes - for each product. Through abstraction, differences between the products are removed so that a domain architecture, including a domain feature model, can be developed. The importance of a product-agnostic architecture should not be overlooked. Such a perspective permits the architect to envision new applications in the domain, and, quite possibly, to expand the domain itself.

To develop the product line architecture, which will eventually form a framework for all applications in the product family, variability analysis re-associates features with their respective products. However, in the product line approach, this association is done with the intent of reducing the code base as much as possible, applying inheritance and parameterization as much as possible.

With a product line framework in place, contracts for each family member are developed to formalize their feature content. This formalization provides a mechanism more concise than UML's multiplicity and, given that there is a single instance of it in the model, makes the model more maintainable.

As the architecture transformation progresses, the focus will be on the development of the feature model. The importance of the analysis and design models is not being trivialized. It is recognized that a solid understanding of these architectural components is key to developing a successful product, but they are processes that are already well defined and understood.

The relationship between the feature model and the analysis and design models changes as a function of the stage of logical transformation. The analysis model relationships dominates the early stages - domain engineering - due to its abstract nature. The design model will be of more interest as the product line architecture is developed, since it is at this point that assets that will be applied to real products are being created.

6.3 Adapting the Base Product Architecture

For this application, the Hull Mounted Sonar (HMS) has been selected as the base product for two reasons. It is known to include functionality found in several competing speciality sonars, but, apart from submarine detection, its performance isn't comparable. Experience also shows it to be very adaptable when new functionality is required.

Adapting the architecture model of the HMS - whose data dictionary is included as Appendix C for the purposes of documenting the detailed aspects of the architecture - to be suitable for the logical transformation to domain and product line architectures, starts with selecting the top 5-20% of the use cases. These use cases will identify all the major software components[18]. A new analysis model is developed from these use cases, modifying the model's stereotyping notation to reflect that it is the base product. From there, the Feature Model is constructed by transforming the analysis classes and control objects to functions and features.

For the HMS, the following use cases were selected:

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

Table 6-1 Primary hull-mounted sonar use cases

The revised use case diagram, Figure 6-3, using only the primary use cases, highlights what will eventually become the major components of the feature model. The use cases become features because these are what are visible to the user. In addition to these, there are also many control, processing and I/O objects that represent underlying functionality, but are not directly visible to the user. In FOOM, the definition of external actors as described in [31] is altered to reflect user visibility. They are represented by the actors in the diagram and will be identified as "functions" in the Feature Model class diagram, Figure 6-4. Objects are created based on the actors that are not directly user visible and designate them with the stereotype *<<base product function>>*. Similarly, objects are created based on user-visible boundary objects and assign them stereotypes *<<base-product feature>>.*

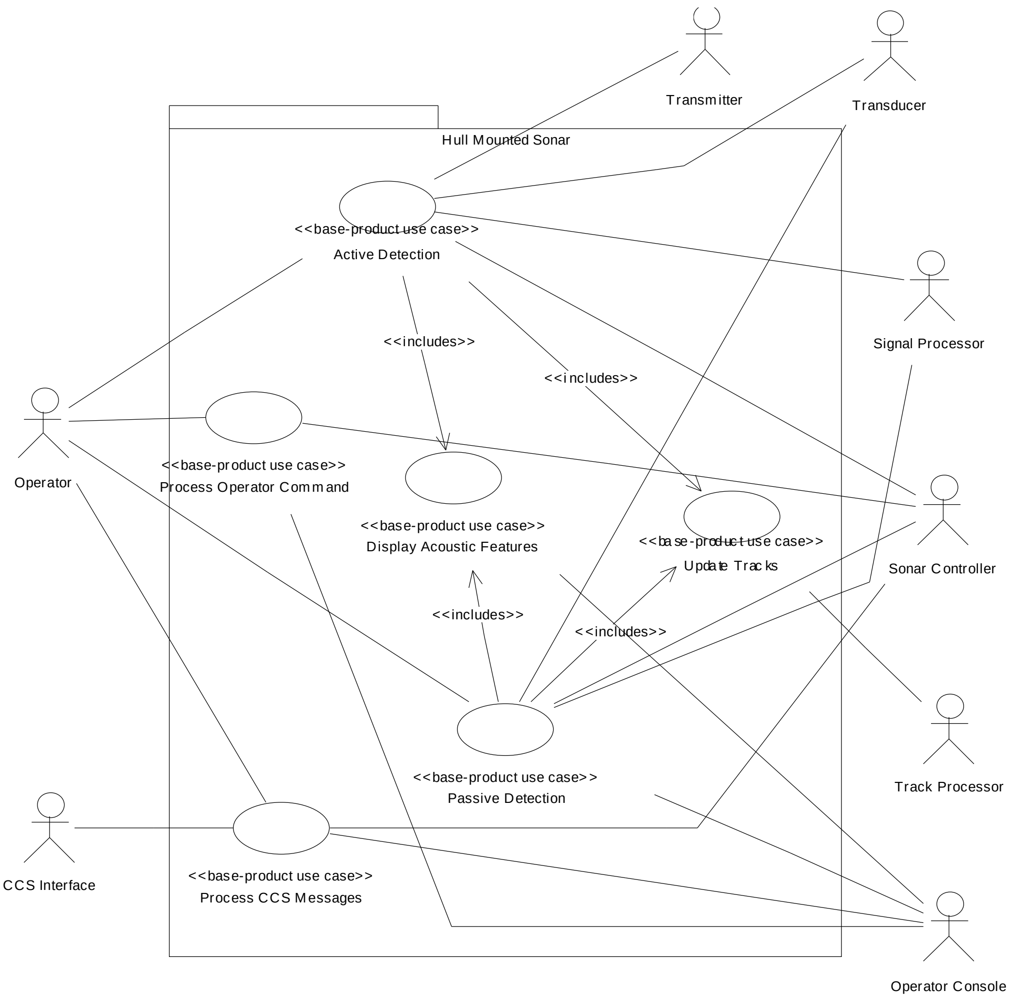
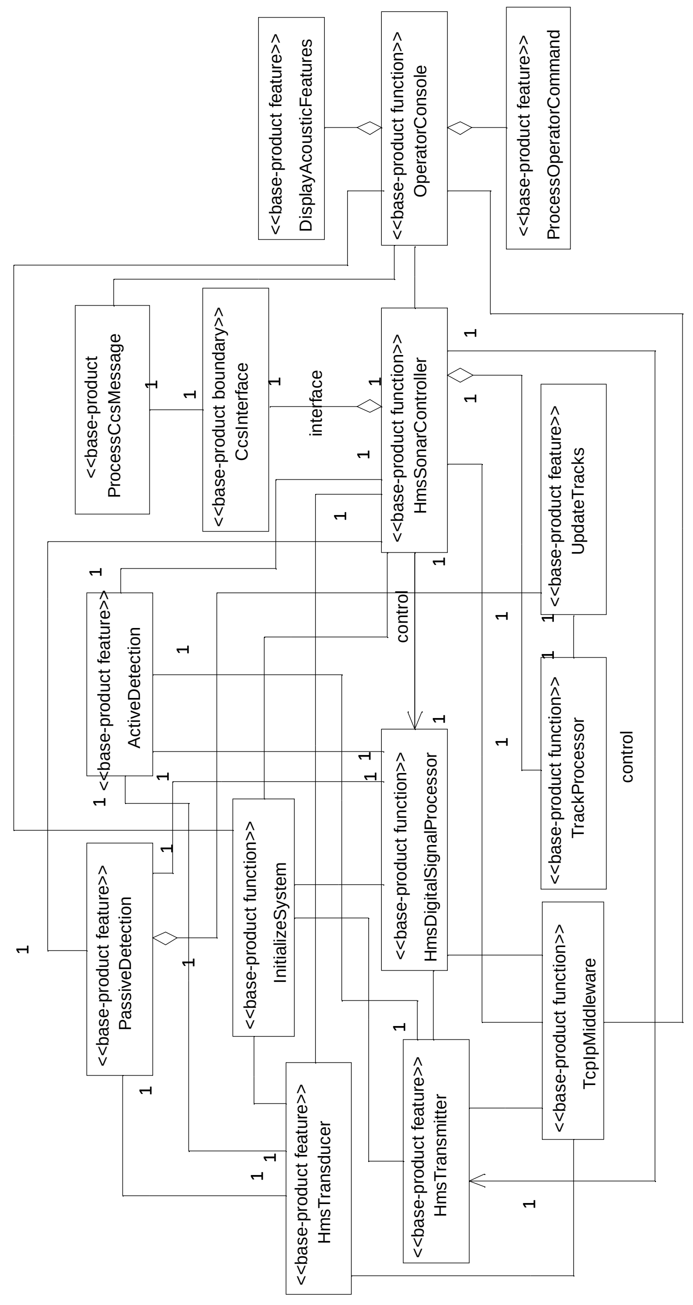


Figure 6-3 Base product use case diagram



<<base-product feature>> DisplayAcousticFeatures
<<base-product function>> OperatorConsole
<<base-product feature>> ProcessOperatorCommand
<<base-product ProcessCcsMessage
<<base-product boundary>> CcsInterface
interface
<<base-product function>> HmsSonarController
<<base-product feature>> UpdateTracks
<<base-product feature>> ActiveDetection
control
<<base-product function>> TrackProcessor
control
<<base-product function>> HmsDigitalSignalProcessor
<<base-product function>> InitializeSystem
<<base-product feature>> PassiveDetection
<<base-product function>> TcpIpMiddleware
<<base-product feature>> HmsTransmitter
<<base-product feature>> HmsTransducer
1

Figure 6-4 Base product feature model

6.4 Developing the Domain Feature Model

Once there is a good understanding of the base product as represented by architectural artifacts (analysis, feature and design models), development of the domain architecture can proceed. The object of this set of activities is to distil the base product architecture so that it can be applied to all products in the domain. The domain architecture process begins with identifying all the products, current and future, in the domain. From that list, commonality and variability analysis is applied to determine the features and functions that are found in each product. This gives rise to a new domain feature model which then helps to populate domain use case diagram.

6.4.1 Understanding the Products in the Domain

As stipulated at the beginning of this chapter, the products in the sonar domain that will be used to apply FOOM are: hull-mounted, variable depth, towed-array and mine-hunting sonars. The product-feature matrix [26] in Table 6-2 identifies the base product feature, or some version of them, that would be included in each of the domain products.

Examination of Table 6-2 reveals that Active Detection and Passive Detection features are not common across all products. Abstraction of this feature to Detect Acoustic Features is sufficiently general to be applicable to all products.

	Product Features						
Sonar Type	SystemInit.	Active Detection	Passive Detection	Track Processing	Acoustic Features Displav	Process Operator Command	Ccs Comm'n.
Hull Mounted	•	•	•	•	•	•	•
Variable Depth	•	•	•	•	•	•	•
Towed Array	•		•	•	•		•
Mine Hunting	•	•		•	•	•	•

Table 6-2 Summary of features for the various sonar types

After the domain use cases have been determined, defining the underlying functions that will support the features follows. Domain expertise is required to carry out this phase since the architect cannot understand the inner workings of the systems without previous firsthand experience in at least some aspect of their design. Table 6-3 lists the products and the components required to implement them.

Note that there is a function present which has not as yet been seen - the Winch. It is known that, although it is not part of the base product, it does exist in other systems. It is listed here to capture its existence to provide traceability, but will be removed as the product features are abstracted. Also note that passive systems do not have a transmitter, causing it to be removed from the component list for the domain. 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 the incoming acoustic

	Product Functions / Subsystem Controllers								
	Transmitter	Transducer	Digital Signal Processor	System Controller	Track Processor	CCS interface	Distributed Communication	Operator Console	Winch
Hull Mounted	•	•	•	•	•	•	•	•	
Variable Depth	•	•	•	•	•	•	•	•	•
Towed Array		•	•	•	•	•	•	•	•
Mine Hunting	•	•	•	•	•	•	•	•	

Table 6-3 Summary of Product Functions for various types of sonar

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 CcsInterface. With these, the domain architecture feature model and use case diagrams can be built See Figures 6-5 and 6-6 respectively.

It should be noted that, similar to the base product model, the communication middleware does not appear in the use case diagram. This is done solely for clarity. Since it is known that the sonar is a distributed system, the implication is that there is some form of distributed communication system. The communication mechanism is, however, introduced in the domain feature model for traceability purposes. The operating systems could be treated in a similar manner.

So far, data objects - <<entity>> - have been ignored in the development of the feature models for the base product and the domain. They can vary widely from product to product in a given domain, such as sonar, their variation is easily modelled with inheritance and parameterization and will continue to be left out of the models presented in this document. Other domains where entities such as databases play a prominent role require a more thorough treatment of these object types.

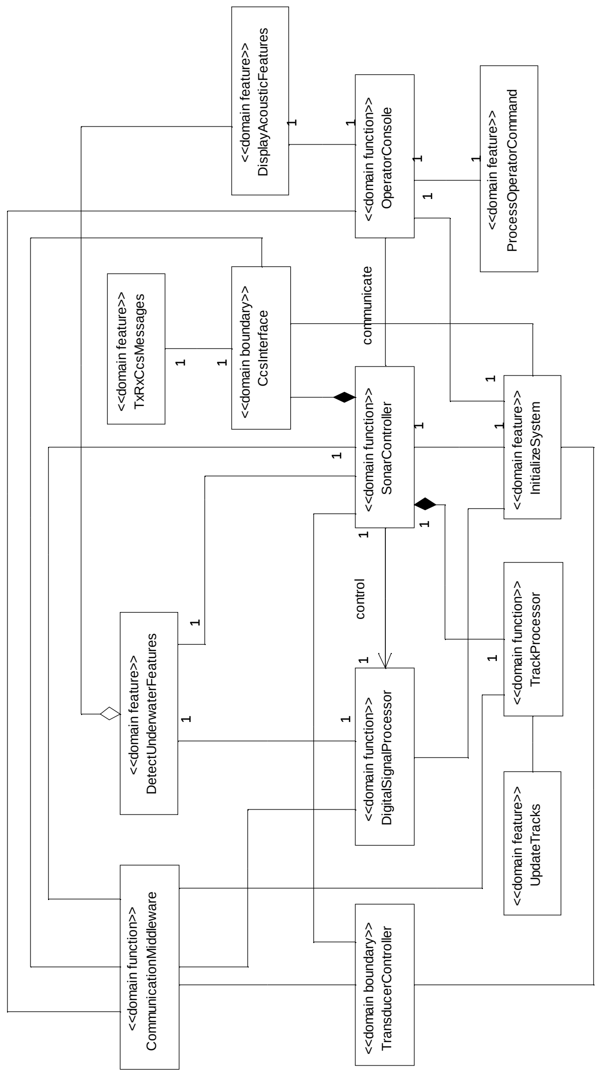

<<domain feature>> DisplayAcousticFeatures
<<domain function>> OperatorConsole
<<domain feature>> ProcessOperatorCommand
<<domain feature>> TxRxCcsMessages
<<domain boundary>> CcsInterface
communicate
<<domain function>> SonarController
<<domain feature>> InitializeSystem
control
<<domain feature>> DetectUnderwaterFeatures
<<domain function>> DigitalSignalProcessor
<<domain function>> TrackProcessor
<<domain feature>> UpdateTracks
<<domain function>> CommunicationMiddleware
<<domain boundary>> TransducerController

Figure 6-5 Sonar domain feature model

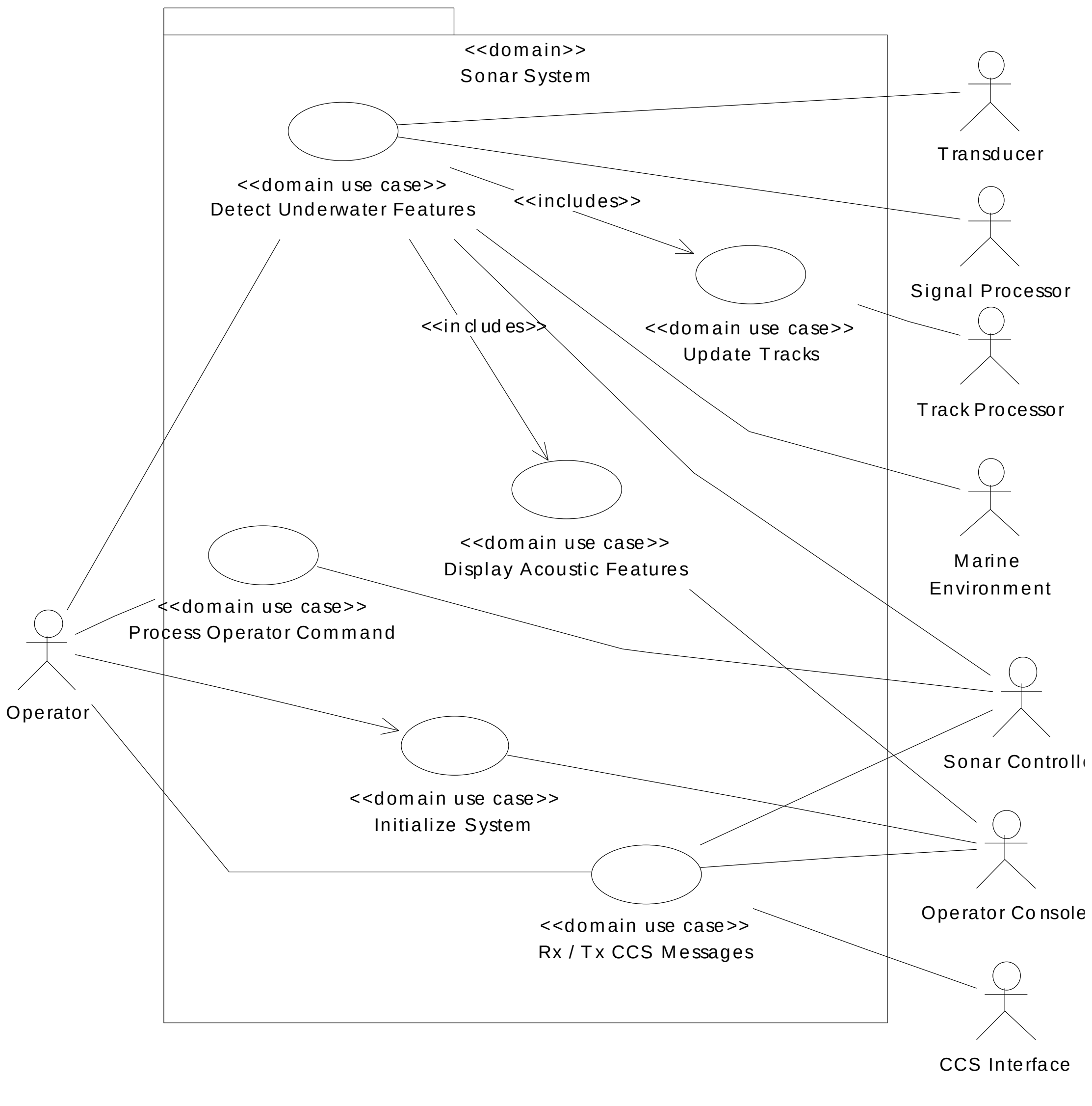


Figure 6-6 Sonar domain use case diagram

6.5 Developing the Product Family Feature Model

Now that the feature model for the domain has been developed, the next phase of the logical transformation can proceed: developing the product line feature model. The process starts with revisiting the product family members, performing commonality and variability analysis to enumerate each product's features and functions. From there use cases are identified. The goal is to build a hierarchical use case model derived from the product family's features[33][35]. The domain features become the abstract superclass for subfamilies of features found at the product line and product levels. This in turn generates a hierarchy of use cases to drive the development of the product line analysis and design models.

The process starts with building a table of all the features for all products in the family. Each feature is examined to determine if it, or a more generalized form, can be found on another product. If so, it is assigned the <<*product line feature*>> stereotype. In turn, features found only on specific products are designated with the <<*product feature*>> stereotype. This process must be done methodically to ensure that the true relationships between the features throughout the product family are captured. To complete the product line feature model, <<*domain function*>> objects are migrated to <<*product-line function*>> objects.

Table 6-4 lists the major features in the product family and indicates the products that would use them. They are used to build the feature model for the sonar product family.

See Figure 6-7.

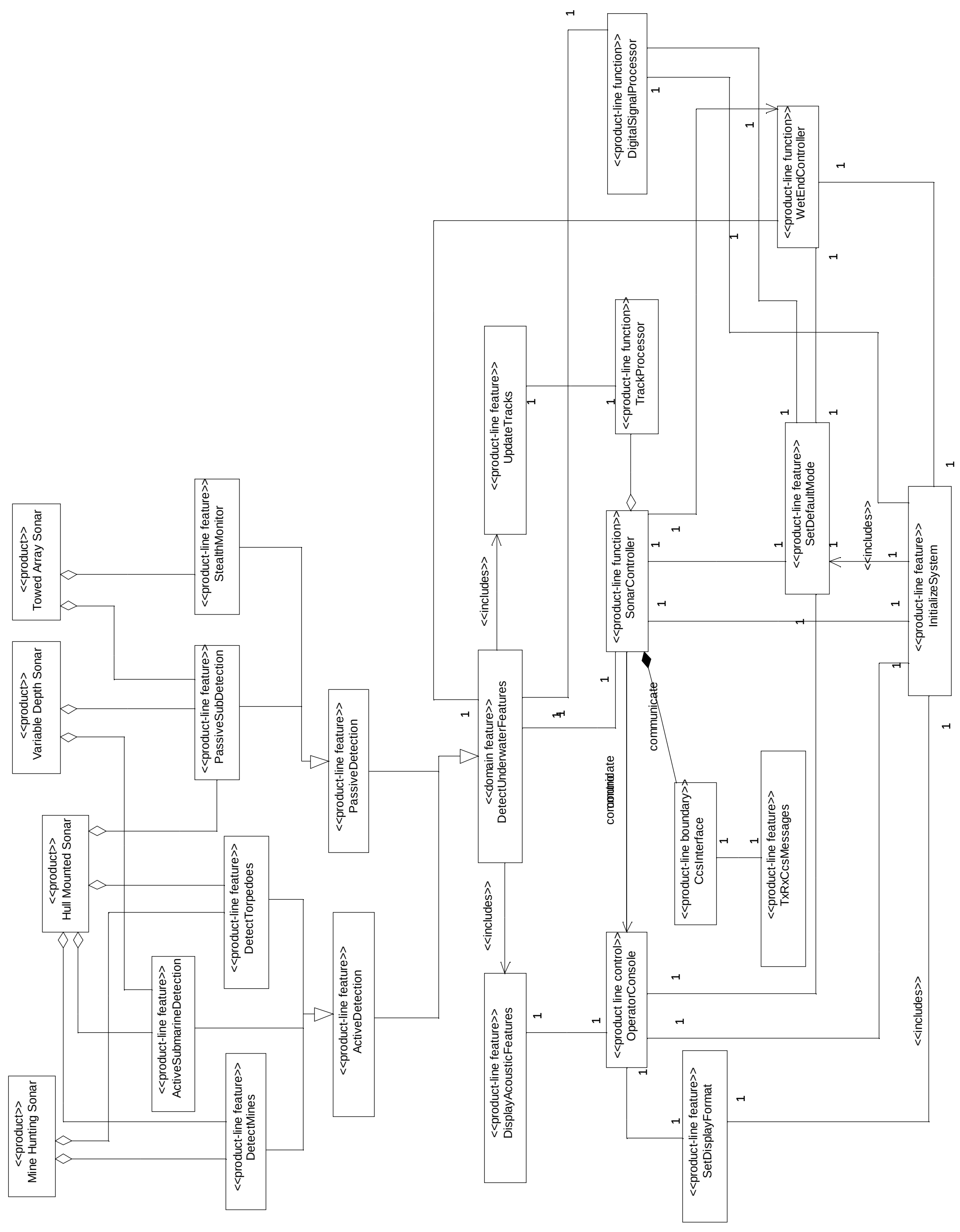
Feature	**Description**	**HMS**	**VDS**	**TAS**	**MHS***
InitializeSystem	Manage power-up sequences and initial configuration of subsystems	•	•	•	*
SetPassiveMode	Set subsystems in passive detection mode	•	•	•	
PassiveDetection	Process and display income passive acoustic features	•	•	•	
Display Acoustic Features	Displays acoustic features, tracks and other visuals on the operator console	•	•	•	*
SetDisplayFormat	Sets the format for displaying acoustic features	•	•	•	*
SetActiveMode	Set subsystems in active detection mode	•	•		
SetDetectionZone	Set the angular window in which detection will be carried out	•	•		*
SetTransmissionType	Set the acoustic energy transmission type (CW or FM)	•	•		
ActiveDetection	Detect objects in the surrounding water	•	•	•	*
ProcessCCSMessages	Process (Rx/Tx) messages between sonar and ship's command and control system	•	•	•	*
DetectSubmarines	Default active configuration for HMS / VDS	•	•	•	
DetectMines	Special configuration of active mode for detecting mines	*			*
DetectTorpedoes	Special configuration of active mode for detecting torpedoes	*			

Table 6-4 Sonar product line features (• = existing, * = future)

The product line features can be specialized versions of domain features where the specializations vary amongst several subgroups in the product family, similar to [33][35]. An example in the sonar would be groups of active and passive sonars, and some that have both functionalities. This allows the Subclassing of features.

The product line feature model (Figure 6-7) includes features of individual products to provide a mechanism for correlating the products back to the product family. Two specific areas in the feature model show multiple levels of specialization going from the domain through to individual products: *Detect Underwater Features* and *Display Underwater Features.* In situations where functions are applicable across the product line, such as the sonar controller, the signal processor and the transducer, the association has been left between the equivalent actor and the product line feature to avoid cluttering the diagram.

Development of the product line use case diagram (Figure 6-8) begins with the mapping of features to use cases and functions to actors. Unlike the normal practice for single systems, the FOOM's product line use case diagram includes inheritance, aggregation and other associations. This can cloud the underlying architecture.



<<product>> Towed Array Sonar
<<product>> Variable Depth Sonar
<<product>> Hull Mounted Sonar
<<product>> Mine Hunting Sonar
<<product-line feature>> StealthMonitor
<<product-line feature>> PassiveSubDetection
<<product-line feature>> DetectTorpedoes
<<product-line feature>> ActiveSubmarineDetection
<<product-line feature>> DetectMines
<<product-line feature>> PassiveDetection
<<product-line feature>> ActiveDetection
<<domain feature>> DetectUnderwaterFeatures
<<product-line feature>> UpdateTracks
<<product-line feature>> DisplayAcousticFeatures
<<includes>>
<<includes>>
<<product-line function>> TrackProcessor
<<product-line function>> DigitalSignalProcessor
<<product-line function>> WetEndController
<<product-line function>> SonarController
<<product-line feature>> SetDefaultMode
<<product-line feature>> InitializeSystem
<<includes>>
<<includes>>
communicate
communicate
<<product-line boundary>> CcsInterface
<<product-line feature>> TxRxCcsMessages
<<product line control>> OperatorConsole
<<product-line feature>> SetDisplayFormat
1

Figure 6-7 Sonar product line feature model.

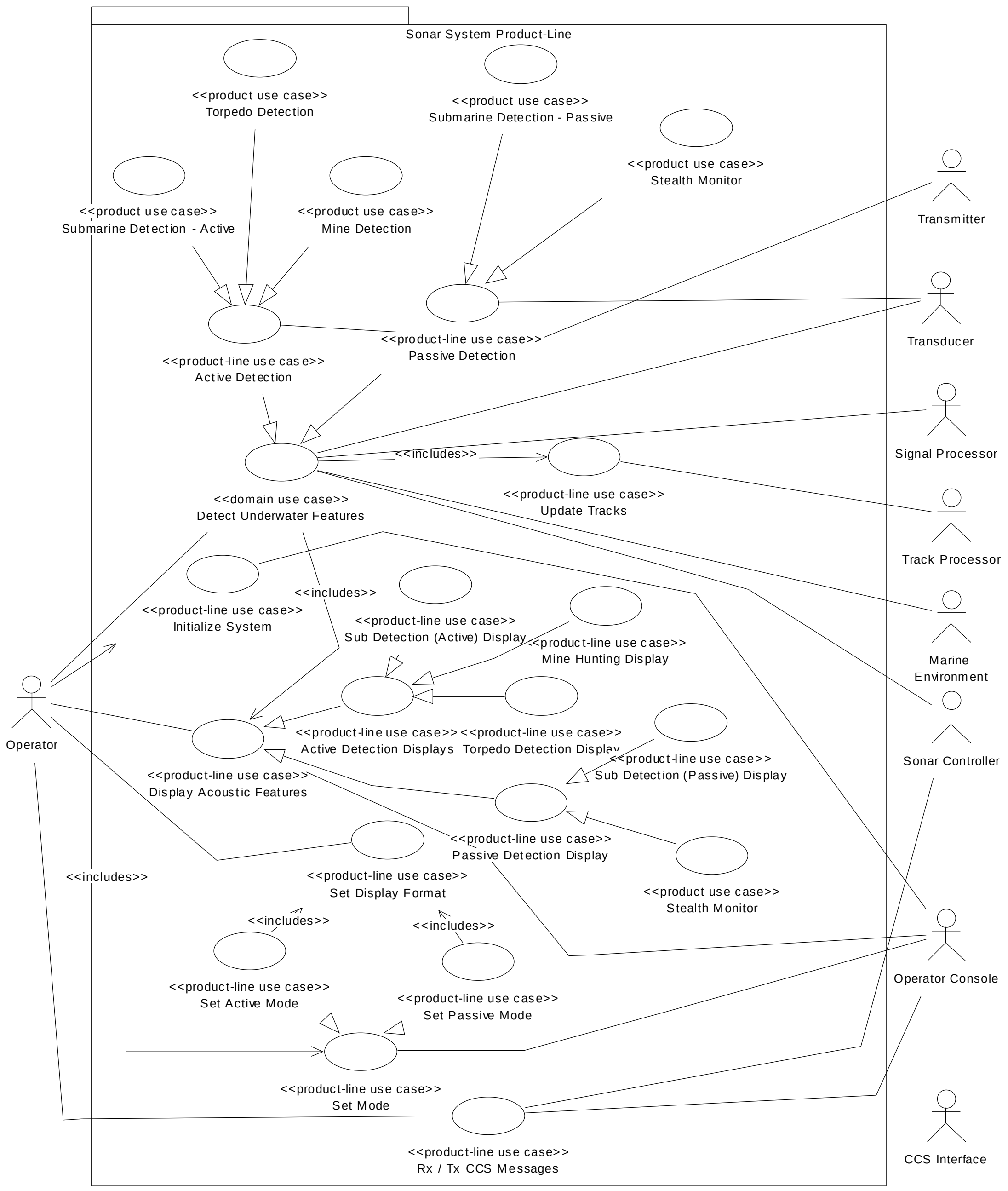

Sonar System Product-Line
<<product use case>> Torpedo Detection
<<product use case>> Submarine Detection - Passive
<<product use case>> Stealth Monitor
<<product use case>> Submarine Detection - Active
<<product use case>> Mine Detection
<<product-line use case>> Passive Detection
<<product-line use case>> Active Detection
<<domain use case>> Detect Underwater Features
<<includes>>
<<product-line use case>> Update Tracks
<<product-line use case>> Initialize System
<<includes>>
<<product-line use case>> Sub Detection (Active) Display
<<product-line use case>> Mine Hunting Display
<<product-line use case>> Active Detection Displays
<<product-line use case>> Torpedo Detection Display
<<product-line use case>> Sub Detection (Passive) Display
<<product-line use case>> Display Acoustic Features
<<product-line use case>> Passive Detection Display
<<product use case>> Stealth Monitor
<<product-line use case>> Set Display Format
<<includes>>
<<includes>>
<<includes>>
<<product-line use case>> Set Active Mode
<<product-line use case>> Set Passive Mode
<<product-line use case>> Set Mode
<<product-line use case>> Rx / Tx CCS Messages
Transmitter
Transducer
Signal Processor
Track Processor
Marine Environment
Sonar Controller
Operator
Operator Console
CCS Interface

Figure 6-8 Product line use diagram

Examination of the feature model indicate that there are two features which touch on all or most of the system: Active Detection and Passive Detection. Relying on FOOM's definition of a feature - a user visible behavior or attribute that aggregates many requirements, representation of a feature evolves to include aspects from both the product line feature model and use case diagram.

Figures 6-9 and 6-10 include not only the artifacts usually found in use case diagrams, but also include relationships between use cases and between actors. At this level of granularity, the view of the entire system - the architecture - begins to take shape. Since each major feature includes the same infrastructure, namely the <<*function*>> objects, the architectures of all features in a single product and all the products in the domain will be internally consistent.

While developing the product line feature model, the notion of use cases as architectural patterns [26][32] began to evolve. Like design patterns, their detailed implementation can vary significantly, but when viewed from the domain and product line perspective, very few differences across products emerged. What this led to was a relaxation of the rigor in performing analysis, because, in fact, the behaviors and attributes were quite similar. The potential for productivity improvement is substantial, allowing architects to focus on building the product framework and individual products, revisiting the analysis portion of the models only for regression purposes or for introducing a feature not yet implemented on any of the existing products.

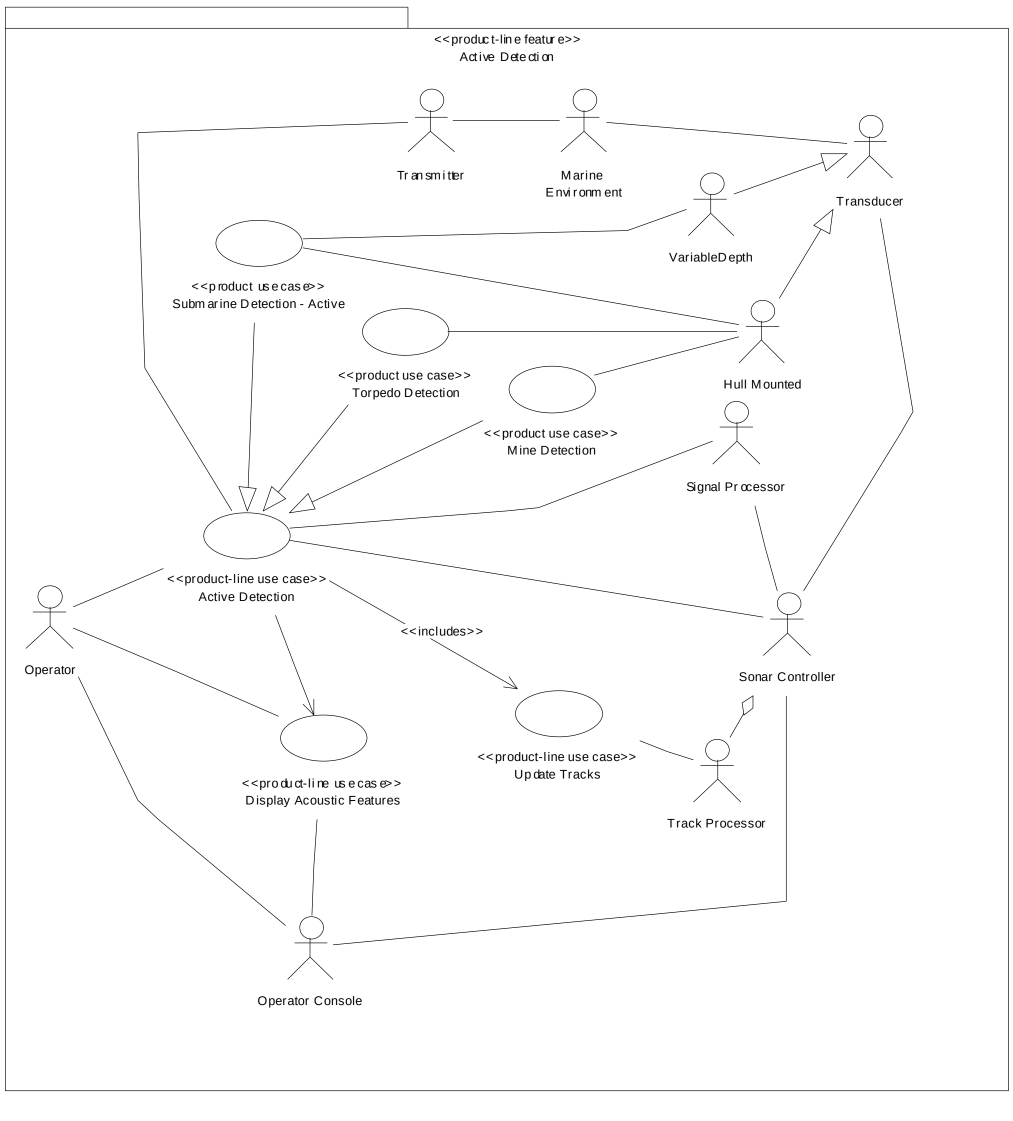


Figure 6-9 Sonar family Active Detection feature

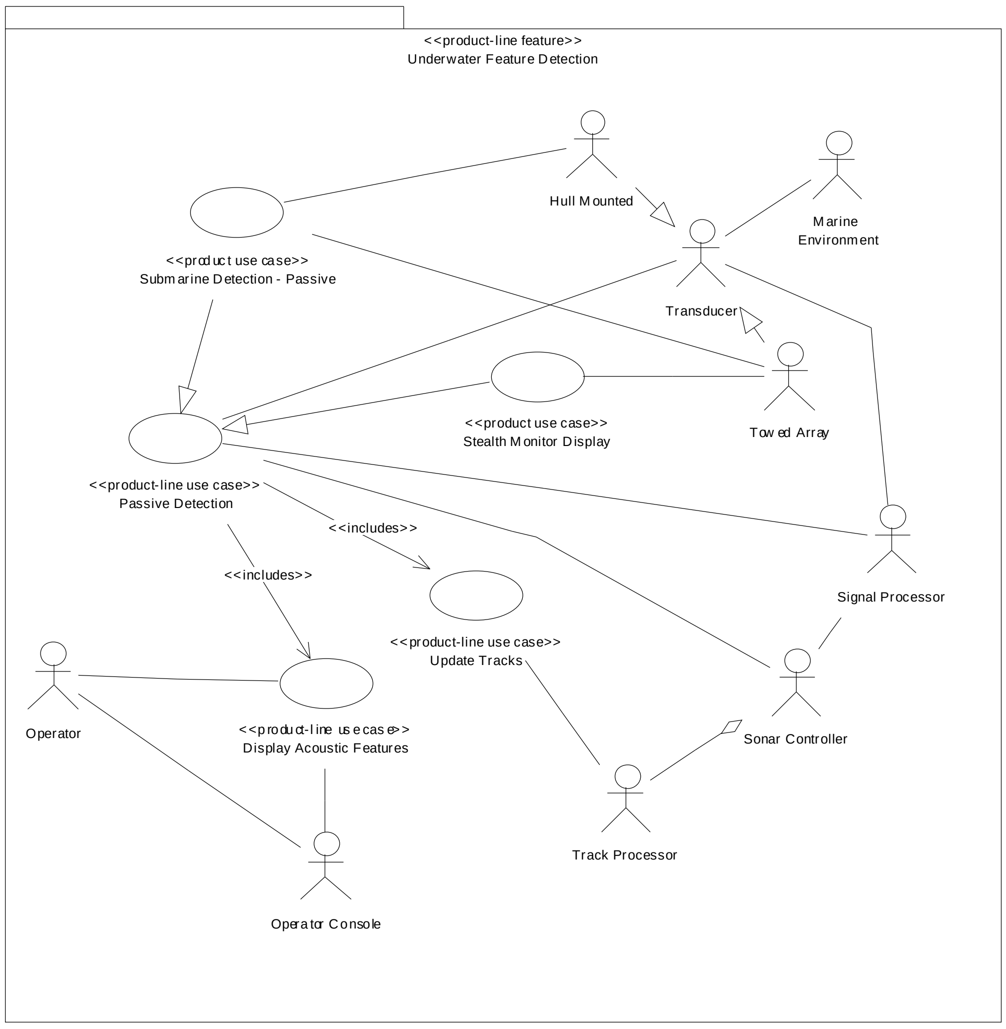


Figure 6-10 Sonar family Passive Detection feature

6.6 Product Contracts

Now that the products in the family and the features available for each have been identified, the process of building the product contracts begins. To review, these 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 the Object Constraint Language (OCL), a component of UML.

First look at the contract for the product line. It includes the enumeration of all the features and functions available across all products, as well as the attributes for all common functions and features. Using an inheritance relationship between the product line feature contracts and the product contracts, the enumerations are passed to the sub classed objects, via the class invariants for each.

For the sonar family, the 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
        -- members of the product line
        ProductFunctions enum{  SonarController,
                            SignalProcessor,
                            OperatorConsole,
                            TrackProcessor,
                            Transducer,
                            Transmitter,
                            Winch,
                            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;
```

A typical contract for one of the family members identifies the features and functions required to instantiate the architecture for that product. The feature contract for the Hull-mounted sonar (HMS) is presented here

```
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( TxRxCcsMessages ) 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;
```

6.7 Building the Product Line Design Model

With the feature model for the known members of the product family, and a set of contracts for each product in place, the task of examining each subsystem with respect to the role it plays for each product begins. When completed, the design model will represent the detailed design for an entire family of products. The largest constituent of the design model is the subsystem decomposition class diagram and detailed views of each subsystem. This section will be limited to the more interesting subsystems and provide some perspective on how the structure was developed.

6.7.1 Operator Console Subsystem

The operator console is, by far, the most complex in terms of features. But this is to be expected as it is, by nature, the subsystem with most interaction with the user. Start by examining the two most obvious components of the operator console: the operator input devices and the video display. The input devices are given a base class so that, although not immediately evident, some reuse is possible.

The video, however, 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 sonars have active and passive submarine detection displays. Others have displays, such as mine hunting and stealth monitoring, that are used on only one product. However, by grouping these features at the product line level, the possibility for reuse is exposed that might not otherwise be so evident. Figure 6-11 shows the Operator Console subsystem class diagram from the product line architecture.

6.7.2 Digital Signal Processor Subsystem

The digital signal processor subsystem is a grouping of very complex algorithms with very strict real-time performance constraints. It also presents opportunities for reuse, but this time through parameterization, not inheritance.

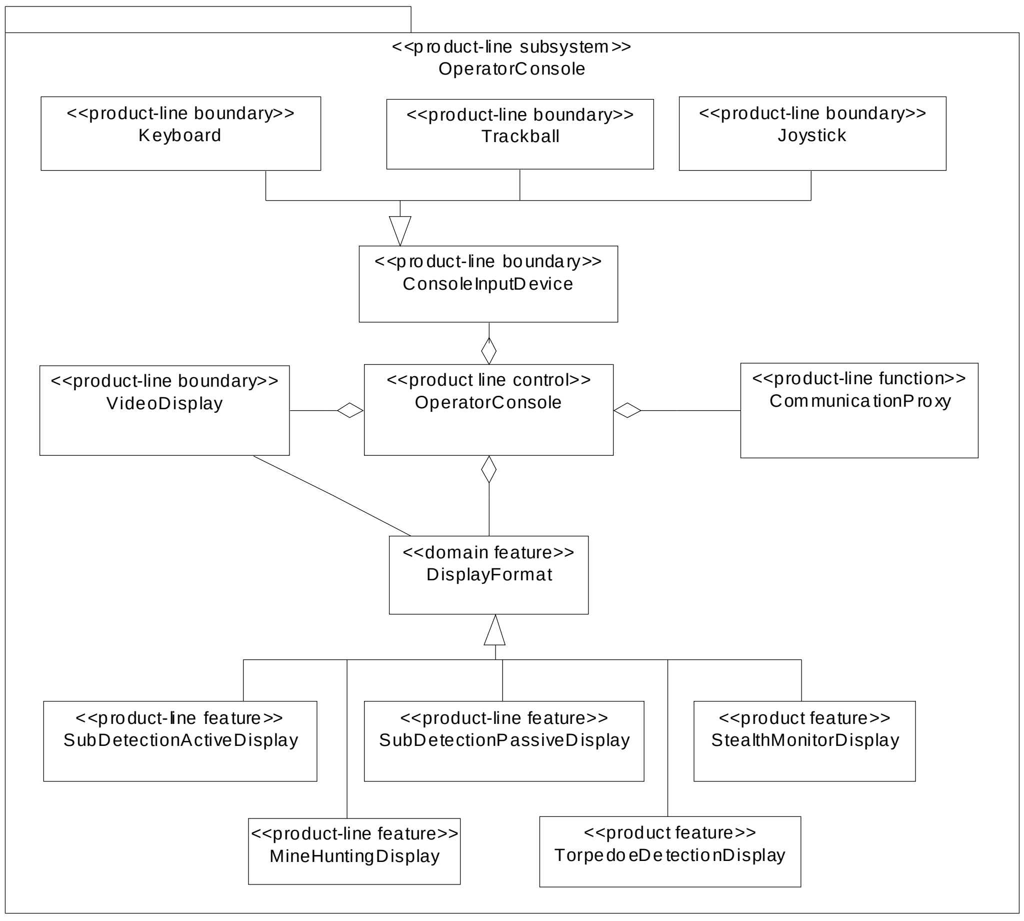


Figure 6-11 The Operator Console subsystem as defined in the product line design model.

The processing modules are divided along two lines, active and passive. Each uses a set of algorithms different enough from each other that there would be no reason to harmonize them. However, within active or passive modules, the algorithms are nearly identical, varying according to parameter sets. In active sonar, these would include transmission type, transmission frequency, and the number of channels collecting data from the marine environment. Passive sonars would include a time index and the active channels in the transducer. As complex as it is inside, from the product line perspective the main engines of the signal processor modules can be represented with two parameterized classes: ActiveProcessor and PassiveProcessor. Adding a CommunicationsProxy provides a connection to the rest of the system. Figure 6-12 illustrates the product line view of the Digital Signal Processor subsystem.

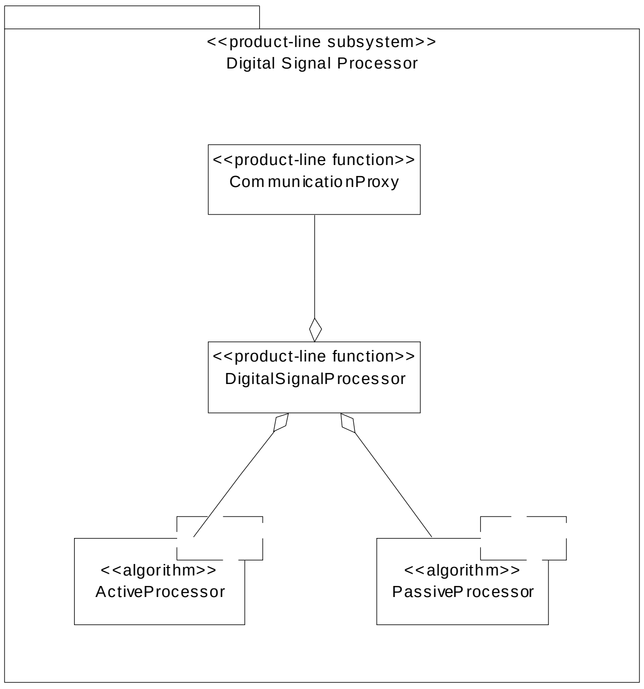


Figure 6-12 Digital Signal Processor subsystem class diagram from the product line architecture

6.7.3 The Command and Control (CCS) Interface

The command and control interface, unlike the other systems, does not vary as a function of the product to which it belongs. Instead, it varies as a function of the external environment to which it will be connected. The flavors of the CCS Interface are, say, CanadianNavy, USNavy and SwissNavy. The choice of which one will be used on a given product is directly correlated to "affiliation" of the vessel on which any given sonar is installed. This makes the CCS Interface, in essence, a product within a product. Figure 6-13 shows the CCS interface hierarchy.

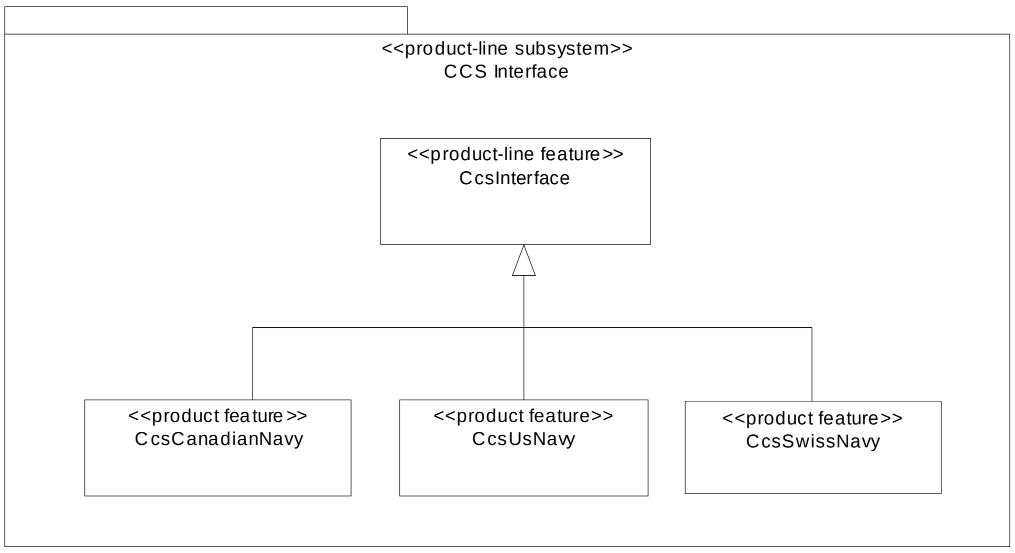


Figure 6-13 Command and Control Interface subsystem from the product line architecture.

6.7.4 Software Deployment

With the software architecture for the product family in place, a deployment diagram is developed to understand the relationship between the software and the hardware. Until this point, it has been assumed that physical computing assets, of some form, would be available on which the software would run. Performance requirements / constraints were accounted for in the design.

The sonar product line deployment diagram shows the target nodes on which the software components will be deployed. From this, software architects and systems engineers have a common understanding of the types of hardware family members will require.

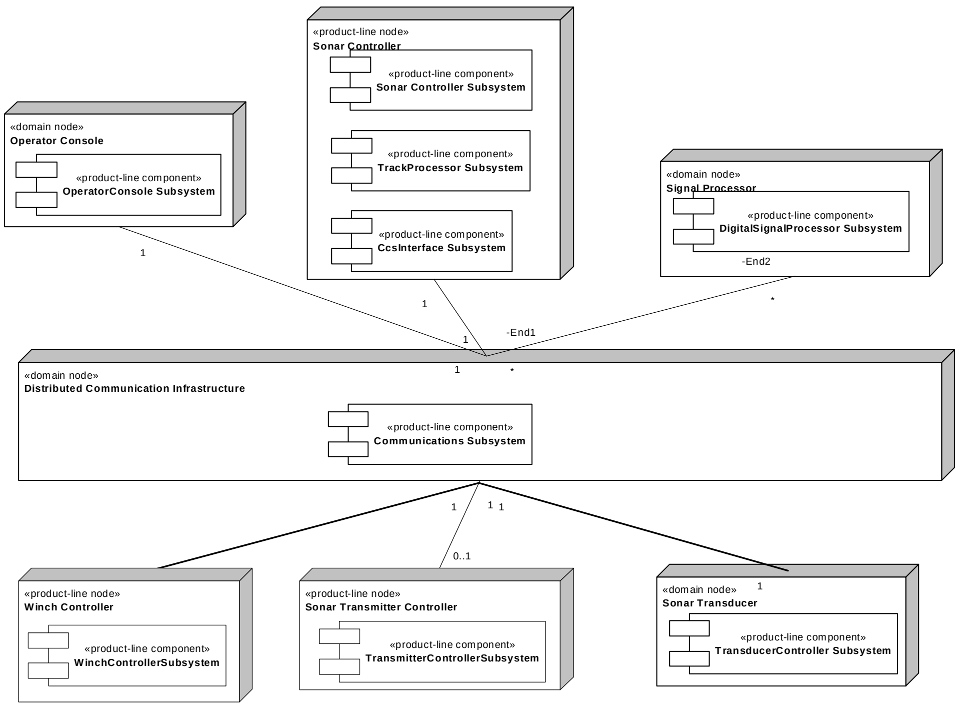


Figure 6-14 Sonar product-line deployment diagram

6.8 Summary

In this chapter FOOM has been applied to a family of sonars as a test case. The model was then systematically applied starting with a candidate for the base product, then adapting its architecture so it is suitable for transformation to a domain and product line architecture. Domain analysis identified other products as members of the product family. From there commonality analysis permitted the development a domain architecture that was product-agnostic. The first part of the transformation relied mostly on the analysis portion of the USP to support the detailed architectural evolution. The design model, particularly the subsystem decomposition, was developed primarily as a regression mechanism to ensure the models were internally consistent as the transformation progressed from one stage to the next.

Evolution of the product line architecture is where features in analysis served primarily as design patterns for similar functionalities across products. The product line feature model to assisted in identifying the components in each subsystem. Reviewing each feature model exposed areas for productivity gains from reuse, inheritance and parameterization.

Throughout the process, the feature model served as the focus of the modelling effort. Transformation of features and functions to model components during analysis provided a connection between the user visible aspects of the system, and the infrastructure underneath.

Finally, a series of processes have been provided that couple architecture development and evolution with a modelling standard that is equally applicable at every step of the transformation. The processes themselves and all their artifacts created along the way are object-oriented.

Chapter 7

Conclusion

In this work, a set of processes has been developed that will assist an organization to adopt a product line practice. This practice will be grounded in the most current of modelling technologies, lend itself to constant evolution, and provide a high degree of customer focus.

7.1 Evaluation of FOOM

FOOM represents the extension and amalgamation of several proven methodologies to provide a set of feature-based architecture-centric development processes. It adopts the UML notation using its extension mechanisms to provide an extra layer of traceability. And, its macroscopic view of the product line exposes opportunities for reuse that might not otherwise be apparent.

7.1.1 Feature-based Development

FOOM has extended the notion of a feature beyond that of FODA. A feature is not only the view of a system and its components from the user's perspective. It also incorporates many of the architectural assets and the relationships between them. This makes features a form of architectural pattern.

FOOM takes the idea of a feature and extends the USP to include a feature model as a peer to the use case, analysis and design models. A strong correlation has been established between features and use cases providing a direct link for features to drive the development of an entire product family.

These aspects of FOOM can be contrasted to FODA which does not provide a clear view on the relationship between the feature model and any development process - object-oriented or otherwise. FODA also does not put forth a set of processes for evolving features, let alone architectures beyond the domain to product lines and individual applications.

7.1.2 Architecture-centric System and Product Line Development

Experience has shown that a well-defined stable architecture is key to the successful development of individual systems. Product line development implies that the resulting architectures must incorporate current, and known and unknown future projects. However, as new products and features are introduced, the architectures must be sufficiently malleable as to not "break" existing products. FOOM's high level view of software systems provides architects and designers alike with a view that forces them to examine the ripple effect of any new feature or change.

KobrA on the other hand takes a diametrically opposed view. It generates a "hardwired" architecture early on in the development process. The intention is to free designers from worrying about big-picture issues, allowing them to concentrate instead on developing individual components to be plugged into a system. This ignorance of system level issues can preclude an individual designer from adequately engaging architects as new features and products are introduced.

FOOM's focus on architecture provides a seamless mechanism for adding a new feature to one or more of the products or in the creation of new products, such as the addition of minehunting to the HMS. The sonar example demonstrates this aspect of FOOM with the mine hunting feature. In developing the product line architecture, no distinction was made between existing and future features and / or systems. This implies that a product line evolutionary road map can be built and maintained from the earliest stages of product development. FODA does provide mechanisms for evolving features, but does not put them in the context individual products.

FOOM's common architecture approach can facilitate the creation of a suite of systems built from several members of the same family. In the case of sonar, this could be a system that integrates separate mine hunting, towed array and hull mounted products on the same ship. Their commonalties make possible the sharing of acquisition, signal processing, display and control functionalities. Moreover, entirely new "integration features" can be built from the artifacts of existing features and functions.

7.1.3 Application Generation

Mechanisms and languages for generating new product architectures based on the product line architecture are built into FOOM. The feature lists at the product line and product provide contracts for specify the features of new products; OCL defined guard conditions and constraints provide the required precision at the product level. The UML / OCL and the USP form the language for generating product architectures.

FAST provides placeholders for these mechanisms / languages but does not define them: this is left to the development team to do. FODA focuses primarily on the domain aspects of a product family, not providing specific mechanisms for application generation. KobrA integrates decision models into the product line framework as text documents apart from the models themselves.

7.1.4 Adoption of the USP and UML

FOOM's underlying processes rely heavily on the USP for building each of the required architectures, making it easier to incorporate into an organization's software development processes. Use of the UML - the de facto industry standard notation - eliminates the need for learning new forms of "boxology" as designers move between organizations. Such standardization permits the use of third party tools for building models, tracking requirements and imposing configuration management and version control. Methodologies like FAST require organizations to build tools to support the process, consuming efforts of designers in tool rather than product development.

7.2 Limitations of FOOM

In its current form, FOOM does not explicitly provide guidance for instantiating a product line architectures where a base product does not exist. With that said, the domain analysis activites of FODA do provide insights into how this is done. These activities need to be further formalized within FOOM to make the case of the inroduction of a new product line, or even a new domain, manageable

One aspect of product line engineering that has not been covered in this first treatment of FOOM is the addition of a feature to an existing product family. The expectation is that analysis at the domain level would have to be conducted to determine that any new feature would in fact be within the family's domain. Following that, rigorous analysis would be required to determine how the new feature would propogate through the family. The details of this process would need to be examined in ongoing development of FOOM.

7.3 Conference Feedback

Two papers[37][38] based on the work in this thesis were presented / published in connection two conferences. Several interesting issues were raised; details of the ensuing discussions follow.

7.3.1 Tools for FOOM

One line of inquiry revolved around whether a TOOL had been developed specifically for FOOM. As previously stated, one of the primary objectives of this work was to leverage existing tools such as Rational Rose, thus short-circuiting the notion of building a proprietary tool. Nonetheless, tools such as Rose were found to be wanting in their malleabilty to adapt outside their hardwired views.

Another issue invloved the automated generation of models, designs and code based on the OCL feature contracts. To date, the author has no knowledge of third-party tools with this functionality. Maybe commercial vendors will offer such a feature in the future.

7.3.2 Architecture Validation

Another question was raised as to whether FOOM validates the generated product architectures. The initial response was that domain expertise was always required, meaning that some human interaction is always necessary. Subsequent reflection provided the same response, however, architects might be able to use objective measures such as requirements and quality metrics to provide an initial objective indicator.

7.3.3 Non-functional Requirements

Since FOOM has its base in the USP and the UML, non-functional requirements can be accommodated for a product line in the same manner as is done for single products - organizational policy, standard practices, etc. A caveat here would be that only non-functional requirements that apply to all or most of the products in the family be included in the family's architecture.

7.3.4 Effectiveness of FOOM

There was some interest in whether a control study had been done to compare FOOM to current methods (or lack of). The first step was to develop FOOM itself, permitting a control study to determine if it does infact provide the economies it espouses.The SEI has done some research[36] into the payback timeline for introducing a product line practice: their results indicate that economies as high as 30% are achievable.

7.4 Contributions

Following are the research contibutions of FOOM:

- A feature model has been added to the Unified Software Development Process (USP). The purpose of the feature model is manifold. Its primary function is to formalize the relationship between the user's perception of the system and how developers go about its design and implementation. It also provides a tighter coupling between the analysis and design segments of the USP. Finally, at the domain and product line levels, it highlights possibilities for reuse that might not otherwise be evident.

- The complement to the previous item is that the USP has been "bolted" onto FODA. FODA does not provide detailed processes for the generation of product line and product architectures.

- A set of processes has been established by extending FODA and the Horseshoe Model and building on techniques from FAST. These processes assist architects in extracting an architecture from a single product, examining it in the context of that product's domain. That architecture is designed to be evolved over time to generate multiple products and multiple variants of each member of the family.

- FOOM has provided a set of mechanisms for generting products that are based, in part, on the OCL, an existing precision component of the UML.

- FOOM has presented constructs that are suitable for incorporation into a UML profile for product lines.

7.5 Future Work

As with all research efforts, every last aspect of a problem can't be examined in one try. There were several areas that would have merited further investigation, but were beyond this initial treatment. Additional work needs to be carried out that will further integrate the UML enhancements and the integration of the feature model into the USP. These include:

- *Application of FOOM in another domain* - This thesis has applied FOOM in only one domain - sonar. Modeling a different family of products can further test the original hypotheses.

- *Refine and expand the role of the OCL in FOOM* - The OCL has been introduced as a mechanism for formally specifying individual products. Application generation rules need to be further studied and expanded. Incorporating the set theory concepts [34] and decision models [35] could prove useful in this area.

- *Developing a profile for software product lines.* Profiles provide a way of grouping UML extensions, such as stereotypes and tagged values, and applying them to a problem domain [31]. FOOM has introduced a raft of extensions that need to be further refined and organized so that they may be standardized and put in a form that is incorporated into third-party tools.

- *Assess the effect of the coarse granularity in this model* - On a first pass, the finer details of implementation were hidden. Further analysis needs to be done to determine if the resulting architectures can be markedly improved with an incremental refinement in detail.

- *Investigate the possibility of interfacing FOOM with KobrA* - there are many similarities between the artifacts generated by FOOM and KobrA. An opportunity may exist for FOOM to provide a greater architecture perspective to KobrA.

- *Further use of third party tools* - In developing FOOM, a simple modelling version of a UML CASE tool was used. But several UML tools are also able to generate code. Using a single tool to both model all architectures in the product and then generate code for each product could provide additional economies of scale than simply building models.

Bibliography

[1] David M. Weiss, Chi Tau Robert Lai, *Software Product Line Engineering: A Family-Based Software Development Process,* Addison-Wesley, 1999

[2] P. Clements, L. Northrup, *Software Product Lines: Practices and Patterns*, Addison-Wesley, 2002

[3] K. Kang, et al, FORM: A feature-oriented reuse method with domain-specific reference architectures, *Annals of Software Engineering*, 5:143--168, 1998

[4] G. Booch, J. Rumbaugh, I. Jacobson, *The Unified Modelling Language User Guide*, Addison Wesley, 1999

[5] R. Kazman, The Architecture Tradeoff Analysis Method, *Proceeding of ICSE'98*

[6] J. Bergey, L. O’Brien, D. Smith, *Mining Existing Assets for Software Product Lines,* Carnegie Mellon Software Engineering Institute, Technical Note CMU/SEI-2000-TN-008

[7] I. Jacobson, M. Griss, Patrik Jonsson, *Software Reuse: Architecture, Process and Organization for Business Success,* Addison Wesley, 1997

[8] I. Jacobson, G. Booch, J. Rumbaugh, *The Unified Software Development Process,* Addison Wesley, 1999

[9] C. Hofmeister, R. Nord, D. Soni, *Applied Software Architecture,* Addison Wesley, 2000

[10] J. Bayer, C. Gacek, T. Widen, PuLSE-I: Deriving Instances from a Product Line Infrastructure, *Proceedings of 7th IEEE International Conference and Workshop on the Engineering of Computer Based Systems,* April, 2000

[11] B. Bruegge, A. Dutoit, *Object-Oriented Software Engineering: Conquering Complex and Changing Systems,* Prentice Hall, 2000

[12] J. Warmer, A. Kleppe, *The Object Constraint Language: Precise Modelling With UML,* Addison Wesley, 1999

[13] P. America, W. van der Sterren, Dealing with Evolution in Family Architectures, *Proceedings of 13th European Conference on Object-Oriented Programming*, June 1999

[14] G. Chastek et al, *Product Line Analysis: A Practical Introduction*, Carnegie Mellon Software Engineering Institute, Technical Report CMU/SEI-2001-TR-001

[16] D. Weiss, Commonality Analysis: A Systematic Process for Defining Families, *Proceedings of ESPRIT ARES Workshop 1998*, Springer 1998, ISBN 3-540-64916-6

[17] J. Coplien, D. Hoffman, and D. Weiss, Commonality and Variability in Software Engineering, *IEEE Software* 15(6), November, 1998

[18] F. Brooks, *The Mythical Man-Month*, Anniversary Edition, Addison Wesley, 1995

[19] R. Kazman, S.G. Woods, S.J. Carriere, *Requirements for Integrating Software Architecture and Recovery Models: CORUM II*, 1998 Working Conference on Reverse Engineering.

[20] Carnegie Mellon Software Engineering Institute, *Product Line Analysis*, http://www.sei.cmu.edu/plp/plp_analysis.html

[21] M. Clauss, Modelling Variability with UML, *Proceedings of the Young Researchers Workshop 2001*, ISBN = 3-00-008419-3

[22] K. Kang, et al, *Feature-Oriented Domain Analysis (FODA) Feasibility Study*, Technical Report No. CMU/SEI-90-TR-2, November 1990

[23] L. Bass, et al, *Third Product Line Practice Workshop Report*, Software Engineering Institute Technical Report No. CMU/SEI-99-TR-003, March 1999

[24] Bass, Clements, and Kazman. *Software Architecture in Practice*, Addison-Wesley, 1997

[25] J. Bayer, et al, PuLSE: A Methodology to Develop Software Product Lines, *Proceedings of Symposium on Software Reusability*, May 1999

[26] J. Bosch, *Design & Use of Software Architectures: Adopting and evolving a product-line approach*, Addison-Wesley, 2000

[27] J. Bayer, C. Gacek, T. Widen, PuLSE-I: Deriving Instances from a Product Line Infrastructure, *Proceedings of 7th IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, April, 2000

[28] J. Bayer, D. Muthig, T. Widen, *Customizable Domain Analysis*, GCSE '99, Erfurt, Germany, September 1999

[29] C. Atkinson, J. Bayer, D. Muthig, *Component-Based Product Line Development: The KobrA Approach*, Fraunhofer Institute for Experimental Software Engineering

[30] C. Atkinson, et al, *Component-Based Product Line Engineering with UML*, Addison-Wesley, 2002

[31] H. Gomaa, *Designing Concurrent, Distributed and Real-Time Applications with UML*, Addison-Wesley, 2000

[32] R.J.A. Buhr, "Use Case Maps as Architectural Entities for Complex Systems", *Transactions on Software Engineering*, IEEE, Vol. 24, No. 12, December 1998, pp. 1131-1155.

[33] H. Gomaa, *Modeling Software Product Lines with UML, Proceedings of the Second International Workshop on Software Product Lines: Economics, Architectures, and Implications*, Toronto, Canada, May 2001, pp. 27--31 IESE-Report. No. 051.01/E

[34] J.M. Thompson, M.P.E. Heimdahl, Ideas on How Product-Line Engineering Can be Extended, *Proceedings of the Second International Workshop on Software Product Lines: Economics, Architectures, and Implications*, Toronto, Canada, May 2001, pp. IESE-Report. No. 051.01/E

[35] Z. Stephenson, J. McDermid, Tracing Features With Decision Models, *Proceedings of the Second International Workshop on Software Product Lines: Economics, Architectures, and Implications*, Toronto, Canada, May 2001, pp. IESE-Report. No. 051.01/E

[36] S. Cohen, Predicting When Product Line Investment Pays, *Proceedings of the Second International Workshop on Software Product Lines: Economics, Architectures, and Implications*, Toronto, Canada, May 2001, pp. IESE-Report. No. 051.01/E

[37] S. Ajila, P.J. Tierney, The FOOM Method – Modelling Software Product Lines in an Industrial Setting, *Proceedings of the 2002 International Conference on Software Engineering and Practice*, Las Vegas, Nevada, June 2002

[38] P.J. Tierney, S. Ajila, FOOM - Feature-based object-oriented modeling: Implementation of a process to extract and extend software product line architectures, *Proceedings of the 8th International Conference on Information Systems Analysis and Synthesis*, International Institute of Informatics and Systemics, pp. 510-515, Orlando, Florida, July 2002.

Appendix A

Acronyms

CASE	Computer Aided Software Engineering
CCS	Command and Control (System) Interface
DSP	Dogital Signal Processor
FODA	Feature Oriented Domain Analysis
FOOM	Feature-based Object Oriented Modelling
HMS	Hull Mounted Sonar
MHS	Mine Hunting Sonar
OCL	Object Constraint Language
OO	Object Oriented
OOAD	Object Oriented Analysis & Design
SEI	Software Engineering Institute
TAS	Towed Array Sonar
UML	Unified Modelling Language
USP	Unified Software Development Process
VDS	Variable Depth Sonar

Appendix B

Hull Mounted Sonar

System Specification

Background

The Hull Mounted Sonar (HMS) is an active sonar that is responsible for managing a transducer (transmitter & receiver) such that data may be collected, processed, and presented to an operator. This allows the operator to maintain an awareness of the activity in the marine environment around the transducer. The operator interacts with the sonar on the ship to which the transducer is attached. The sonar is also connected to other systems on the same ship, such as the Command & Control System (CCS) and the underwater telephone (UWT). For configurations where two instances of the sonar may be operating on the same ship, i.e. one Hull Mounted Sonar, and HMS and one Variable Depth Sonar (VDS), each instance will co-ordinate its activities -- specifically, transmission -- with the other.

The sonar is installed onboard ship in physically separated compartments: an operator space; an equipment space; and a sonar trunk. Cabling is used to connect the elements together. The operator space houses the operator interface equipment of the sonar and is shared by the interfaces to other systems; it provides an environment that allows the sonar operators and ship's staff to co-ordinate the activities of all the systems. The equipment space is used to house the remaining (bulkier) components of the sonar. The sonar trunk houses the hull outfit, the part of the system that actually comes into contact with the water.

HMS Behavior Description

Intermingled amongst the documents are descriptions of behaviors that correlate well to Use Cases. They include operational, maintenance, and training scenarios. This report will only document the operational use cases. For the purposes of this project, it is felt that the effect of including the training and maintenance scenarios would be trivial, and only serve to obscure the primary focus, the operational scenarios.

The operational scenarios that will be considered in this study are:

- Starting up
- Processing passive data (including tracking)
- Processing active data (including tracking)
- Manipulating operational displays
- Adjusting operational parameters
- Controlling operation
- Providing notation
- Pinging
- Monitoring health
- Handling CCS

HMS Required Functionality

The system specifications identify activities to be carried out by the Sonar. These activities are:

- Collect acoustic data
- Generate passive data
- Generate active data
- Save DCS data
- Process passive tracks
- Process active tracks
- Present audio
- Present visuals
- Select presentation
- Set parameters
- Apply control
- Accept notation
- Generate ping
- Monitor health
- Handle CCS
- Handle environment
- Diagnose errors

Appendix C

Hull Mounted Sonar

Data Dictionary

Model Element

Entity Classes

DigitizedData
Container base-class for the result of the analog-to-digital conversion output from the Transducer

Invariant:

DigitizedActiveData
Container for the result of the analog-to-digital conversion active output from the Transducer

Invariant:
self.DataBlock->size >= 1

DigitizedPassiveData
Container for the result of the analog-to-digital conversion passive output from the Transducer

Invariant:
self.DataBlock->size >= 1

DisplayReadyAcousticData
ProcessedData that has been formatted for display.

Invariant:
self.DataBlock->size >= 1

ProcessedData
Base-class for data that has been processed by the digital signal processor (DSP).

Invariant: N/A

ProcessedActiveData
Active data that has been processed by the digital signal processor (DSP).

Invariant:
self. DataBlock->size >= 1

ProcessedPassiveData
Passive data that has been processed by the digital signal processor (DSP).

Invariant:
```
self. DataBlock->size >= 1
```

ProcessingParameters
A data store of transmission parameters for the current ping and ambient environmental conditions (water temperature, salinity, etc.)

Invariant:
```
{
 {ShipSpeed >= 0 and ShipSpeed <= SHIP_MAX_SPEED} and
 {ShipHeading >= 0 and ShipHeading < 360} and
 {StartChannel >= 0 and StartChannel < HMS_NUM_CHANNELS} and
 {TxWindow >=1 and TxWindow < HMS_NUM_CHANNELS} and
 {VOSIW >= 1400 and VOSIW <= 1600}
}
```

SonarTrack
Super class for a persisant acoustic feature. In active mode, it has been present in at least three successive "pings". In passive mode, it is an acoustic feature that, at any given moment in time, stands out from the background ambient noise

Invariant:
```
self.Bearing->size = 1 and
self.Intensity->size = 1 and
self.ContactType->size = 1
```

ActiveTrack
An acoustic feature that has been present in at least three successive "pings".

Invariant:
```
self.PingFirstDetected->size = 1 and
self.PingLastDetected->size = 1
```

PassiveTrack
An acoustic feature that, at any given moment in time, stands out from the background ambient noise

Invariant:
```
self.TimeFirstDetected->size = 1 and
self.TimeLastDetected->size = 1
```

TrackDatabase
The list of all Tracks currently known to the system

Invariant:
```
{
 {self.NumberActiveTracks >= 0 and
  self. NumberActiveTracks <= HMS_MAX_ACTIVE_TRACKS}
 and
 {self.NumberPassiveTracks >= 0 and
  self. NumberPassiveTracks <= HMS_MAX_PASSIVE_TRACKS}
 and
 self.TrackList->size = self.NumberActiveTracks + self.NumberPassiveTracks
}
```

CcsMessage
Container for messages transferred over the CcsInterface

Invariant:
```
self.Contents->size = 1
```

ControlMessage
Container for control messages transferred between the various subsystems in the Sonar

Invariant:
```
self.Contents->size = 1
```

PreparedData
Container for data at an intermediate stage of the signal processing

Invariant:
```
self.PreparedDataBuffer->size = 1
```

Boundary Classes

CcsInterface
The interface between the sonar system and the ship's command and control system

Invariant:
```
self.IncomingMessageBuffer->size >= 1 and
self.OutgoingMessageBuffer->size >= 1 and
self.IsActive->size = 1
```

ConsoleInputDevice
A base class for the operator console input devices (keyboard, joystick, trackball)

Invariant:
```
self.DeviceType->size = 1 and
self.DeviceType = INPUTDEVICE_KEYBOARD or
self.DeviceType = INPUTDEVICE_TRACKBALL or
self.DeviceType = INPUTDEVICE_JOYSTICK
```

TransmitterController
Interface to the acoustic energy transmitter. It is only used during active detection.

Invariant:
```
{
 {TxStartChannel >= 0 and
  TxStartChannel < HMS_MAX_CHANNELS}
 and
 {TxNumberChannels >= 1 and
  TxNumberChannels < HMS_MAX_CHANNELS}
 and
 {TxType = TXTYPE_CW or
   TxType = TXTYPE_FM}
}
```

TransducerController
Interface to the transducer (the device that coverts analog acoustic energy into a digital signal). Used in both active and passive detection.

Invariant:
```
self.RxFrequency->size = 1
self.RxMode->size = 1
```

TcpIpSocket
Encapsulation of the TcpIp sockets.

Invariant:
```
self.FileDescriptor > -1 and
self.FileDescriptor <= MAX_FILE_DESCRIPTOR_VALUE
```

Control Classes

InitializeSystem
Take the entire system from a powered-down state to the default passive detection mode. Includes powering up and initialization of all subsystems: transmitter / transducer, DSP, sonar controller and operator console

Invariant:

PassiveDetection
Detect sources of underwater acoustic energy by listening only

Invariant:

ActiveDetection
One of the primary detection modes of sonar, using transmitted acoustic energy (pinging) to assist in the detection of underwater objects

Invariant:

ProcessCcsMessage
Receive incoming message from ship and process or Prepare a message and send to the ship's command and control system

Invariant:

SetActiveMode
Put the system and its subsystems in active detection mode

Invariant:

SetPassiveMode
Put the system and its subsystems in passive detection mode

Invariant:

UpdateTracks
Add / remove / update the tracks in the track database

Invariant:

DigitalSignalProcessor
The subsystem that converts the raw digitized data into features using digital filters and frequency-domain transformations

Invariant:
```
self.UnpackedDataBuffer->size = 1 and
self.PreparedActiveDataBuffer->size >= 1 and
self.PreparedPassiveDataBuffer->size >= 1 and
self.DataPreparationModule->size = 1 and
self.PassiveDataDspModule->size = 1 and
self.ProcessedPassiveDataBuffer->size >= 1 and
self.ActiveDataDspModule->size = 1 and
self.ProcessedActiveDataBuffer->size >= 1
```

SonarController
The subsystem that controls the behavior of the sonar. It also monitors the status of the subsystems

Invariant:
```
{
 {self.SystemMode = DEFAULT_MODE or
   self. SystemMode = SYSTEMMODE_ACTIVE or
   self. SystemMode = SYSTEMMODE_PASSIVE
 } and
 self.HmsProcessedData->size >= 1 and
 self.HmsPostProcessor->size = 1 and
 self.HmsDisplayReadyData->size >= 1 and
 self.HmsTrackProcessor->size >= 1 and
 self.HmsTrackDatabase->size = 1 and
 self. HmsCcsInterface->size = 1 and
 self. IncomingCcsMsgBuffer->size >= 1 and
 self. OutgoingCcsMsgBuffer->size >= 1
}
```

TrackProcessor
Module that analyzes incoming acoustic data looking for potential features. Maintain and update the history of features detected

Invariant:
```
self.ProcessedDataBuffer->size >= 1
```

OperatorConsole
The man-machine-interface for acquiring operator input and displaying information to operator

Invariant:
self. SystemMode->size = 1 and
self.ControlMessageBuffer->size >= 1 and
self.CurrentDisplayFormat->size = 1 and
self.IncomingCcsMessageBuffer->size >= 1 and
self.OutgoingCcsMessageBuffer->size >= 1 and
self.IncomingControlMessageBuffer->size >= 1 and
self.OutgoingControlMessageBuffer->size >= 1 and
self.Display->size = 1 and
self.Keyboard->size = 1 and
self.TrackBall->size = 1 and
self.Joystick->size = 1

DataPreparationProcessor
Unpacks raw digitized data and prepares it for further processing

Invariant:
self.UnpackedDataBlock->size = 1 and
self.ProcessedDataBlock->size = 1

PassiveDataProcessor
Module to perform passive processing in the DSP

Invariant:
self.InputBuffer->size = 1
self.OutputBuffer->size = 1

ActiveDataProcessor
Module to perform passive processing in the DSP

Invariant:
self.InputBuffer->size = 1
self.OutputBuffer->size = 1

Attributes

DataBlock	A structure containing the various components of an <<entity>> object
ShipSpeed	The speed of the ship in knots
ShipHeading	The True heading of the ship
StartChannel	The first stave-channel in the transmission window
TxWindow	The number of staves in the transmission window
VOSIW	Velocity Of Sound In Water - calculated as a function of environmental factors, including temperature, salinity, depth, etc.
Bearing	Relative bearing of the track to the ship
Intensity	The strength of the contact
ContactType	Identification of the type of contact
Range	Distance to target at the time of the last ping
FirstPingDetected	Ping number when the target was first detected
LastPingDetected	Most recent ping number when the target was detected
TimeFirstDetected	The time a passive track was first detected
TimeLastDetected	The time a passive track was last detected
NumberActiveTracks	The number of active tracks currently stored
NumberPassiveTracks	The number of passive tracks currently stored
TrackList	The collection of tracks
Contents	A structure containing the contents of a CCS message
PreparedDataBuffer	Structure containing the results of the data preparation module in the DSP
DeviceType	Enum defining the type of operator console input device
TxFrequency	The operating frequency of the transmitter
TxType	The type of transmission (CW or FM)
TxStartChannel	The first stave channel in the transmission window
TxNumberOfChannels	The number of stave channels in the transmission window
ControlMessageBuffer	A buffer to contain control messages. The size will be influenced by the real-time behavior of the system.

RxFrequency	The frequency at which the transducer should listen. In active mode, it will be the same as TxFrequency
RxMode	Transmission mode - CW or FM
ProcessingMode	Processing mode of the DSP - active or passive
UnpackedDataBuffer	Intermediate storage buffer for unpacked data in DSP
PreparedActiveDataBuffer	Intermediate storage buffer for ??? data in DSP
PreparedPassiveDataBuffer	Intermediate storage buffer for ??? data in DSP
DataPreparationModule	Software module to prepare raw acoustic data for the DSP
PassiveDataDspModule	DSP software module to perform passive processing
ProcessedPassiveDataBuffer	Intermediate storage buffer for ??? data in DSP
ActiveDataDspModule	DSP software module to perform passive processing
ProcessedActiveDataBuffer	Intermediate storage buffer for ??? data in DSP
SystemMode	The mode of the system - active or passive
HmsProcessedData	Intermediate storage for processed data
HmsPostProcessor	Module to perform additional processing on the DSP output
HmsDisplayReadyData	Acoustic data formatted for display on the Operator Console
HmsTrackProcessor	Module to analyze acoustic data - tracks are added, updated or deleted.
HmsTrackDatabase	Collection of SonarTracks
ProcessedDataBuffer	Base class for processed data
CurrentDisplayFormat	The display format currently active on the Operator Console
IncomingCcsMessageBuffer	Collection of incoming CCS messages. Size will be a function of the real-time behavior of the system
OutgoingCcsMessageBuffer	Collection of outgoing CCS messages. Size will be a function of the real-time behavior of the system
Display	Video display interface
ConsoleInputDevice	Base class for console input devices
Keyboard	Keyboard interface
TrackBall	Trackball interface
Joystick	Joystick interface
UnpackedDataBlock	*DataPreparationProcessor*
DestinationDataBlock : PreparedData*	*DataPreparationProcessor*

InputBuffer : PreparedData	*PassiveDataProcessor*
OutputBuffer : ProcessedPassiveData	*PassiveDataProcessor*
InputBuffer : PreparedData	*ActiveDataProcessor*
OutputBuffer : ProcessedActiveData	*ActiveDataProcessor*

ConsoleInputDevice::AcquireInput()
Capture input from a console device and forward for processing

VideoDisplay::PresentVisuals() : void
Display acoustic features, tracks, and other information on the console display.

PostProcessor::PerformPostProcessing() : void
Perform post-processing on DSP output if required for current display format.

TrackProcessor::ProcessAcousticFeatureData() : void
Analyze DSP processed data for features.

TrackProcessor::UpdateTracks() : void
Update TreackDatabase as required.

TrackDatabase::AddActiveTrack(SonarTrack* track) : bool
Add an active track to the track database
post: TrackDatabase->includes(track)

TrackDatabase::AddPassiveTrack (SonarTrack* track) : bool
Add a passive track to the track database
post: TrackDatabase->includes(track)

TrackDatabase::UpdateTrack (SonarTrack* track, struct trackinfo) : bool
Update a track's contents.
post: TrackDatabase.track->Contents = trackinfo

TrackDatabase::DeleteTrack (SonarTrack* track) : bool
Remove a track from the track database
not.TrackDatabase->includes(track)

CcsInterface::Initialize() : bool
Initialize the Command and Control interface.
post: CcsInterface.IsActive()

CcsInterface::ReceiveCcsMessage() : int
Receive messages from the ship.

CcsInterface::ProcessCcsMessage() : void
Process messages from the ship.

CcsInterface::SendCcsMessage() : int
Send messages to the ship.

TcpIpSocket::recv(int FileDescriptor, char* Buffer, int BufferSize) : int
Receive data on the socket.
post: TcpIpSocket.BytesReceived = BufferSize

TcpIpSocket::send(int FileDescriptor, char* Buffer, int BufferSize) : int
Send data on the socket.
post: TcpIpSocket.BytesSent = BufferSize

HmsOperatorConsole::ProcessOperatorInput() : void
Determine action required by operator request, then execute.

HmsOperatorConsole::SetDisplayFormat(enum displayformat) : bool
Configure the operator console display for the requested format.
post: OperatorConsole.CurrentDisplayFormat = displayformat

HmsOperatorConsole::ProcessControlMessages() : bool
Parse incoming control messages and execute requested action

ProcessingParameters::UpdateParameters(int Parameter, void ParameterValue) : void
Update the value of the specified parameter.
post: ProcessingParameters.Parameter = ParameterValue

ProcessingParameters::GetParameters(int Parameter, void ParameterValue) : void
Retrieve the value of the specified parameter.

DataPreparationProcessor::UnpackRawData() : void
Unpack incoming data from transducer.

DataPreparationProcessor::PrepareData() : void
Prepare unpacked data for DSP.

ActiveDataProcessor::ProcessActiveData() : void
Apply DSP algorithms to incoming active data.

PassiveDataProcessor::ProcessPassiveData() : void
Apply DSP algorithms to incoming passive data.

HmsTransmitterController::Initialize() : bool
Initialize transmitter.

```
[self.TxFrequency = TXFREQ_DEFAULT and
 self.TxType = TXTYPE_DEFAULT and
 self.TxStartChannel >= 0 and
 self.TxStartChannel < MAX_CHANNELS and
 self.TxNumberChannels > 0 and
 self.TxNumberOfChannels < MAX_CHANNELS]
```

HmsTransmitterController::InitiatePing() : bool
Transmit acousting energy into marine environment ("ping").

HmsTransmitterController::SetTransmissionParameters(enum Parameter, void ParameterValue) : bool
Update the requested parameter's value.
post: HmsTransmitterController.Parameter = ParameterValue

HmsDigitalSignalProcessor::Initialize() : bool
Initialize the DSP.

HmsDigitalSignalProcessor::SetConfiguration(enum Configuration) : bool
Set the DSP's configuration - active or passive processing.
post: HmsDigitalSignalProcessor.ProcessingMode = Configuration

HmsDigitalSignalProcessor::PrepareRawData() : void
Call the DataPreparationProcessor to unpack prepare the raw data

HmsDigitalSignalProcessor::ProcessActiveData() : void
Call the active data processor.

HmsDigitalSignalProcessor::ProcessPassiveData() : void
Call the passive data processor.

HmsDigitalSignalProcessor::SendProcessedData() : void
Send processed data to sonar controller.
post: ProcessedDataSocket.BytesSent = ProcessedDataSocket.BufferSize

HmsSonarController::ReceiveOperatorControlMsg() : bool

Receive control messages from the OperatorConsole.

HmsSonarController::ProcessOperatorCommand() : bool
Process operator commands.

HmsSonarController::SetSystemMode(int Mode) : bool
Configure the system for the selected mode.
post: SystemMode = Mode

HmsSonarController::ReceiveProcessedData() : bool
Receive data from the DSP

HmsSonarController::SendDisplayReadyData() : bool
Send data to operator console for display.

HmsSonarController::SendTransmitterControlMsg() : bool
Send control messages to the transmitter

HmsSonarController::ReceiveTransmitterControlMsg() : bool
Receive control messages from the transmitter.

HmsSonarController::SendTransducerControlMsg() : bool
Send control messages to transducer.

HmsSonarController::ReceiveTransducerControlMsg() : bool
Receive control messages from the transducer.

HmsSonarController::UpdateTrackDatabase() : bool
Update the track database to account for newly processed data.

HmsTransducerController::Initialize() : bool
Initialize the transducer.
post: HmsTransducerController.RxMode = RXMODE_PASSIVE

HmsTransducerController::SetConfiguration(enum Configuration) : bool
(Re-)Configure the transducer.
post: HmsTransducerController.RxMode = Configuration

HmsTransducerController::Listen() : void
Acquire acoustic signals from the marine environment.

HmsTransducerController::SendRawAcousticData() : bool
Send data from transducer to DSP.