Examining, Documenting, and Modeling the Problem Space of a Variable Domain

Christina Varghese June 14, 2002

TR-CIS-0612-02

Department of Computer & Information Science
723 W. Michigan Street, SL 280
Indiana University – Purdue University
Indianapolis, IN 46202-5132

Acknowledgements

I would like to express my sincere gratitude to Dr. Andrew Olson for providing a great deal of input and guidance throughout my project. He has spent a great deal of time helping me to understand key concepts and to examine issues I may not have considered on my own. I would also like to thank Dr. Shiaofen Fang and Dr. Jiang Zheng for serving as members of my project committee.

Also, I would like to thank my husband Joby for supporting me throughout the time I have worked towards my M.S. degree. Finally, I want to thank my daughter Laurna for not being too upset about the many weekends and evenings I spent away from her while working on this project.

This project was supported by the U.S. Department of Defense and the U.S. Office of Naval Research under award number N00014-01-1-0746.

Table of Contents

Abstract

In order to reduce development time and costs associated with producing a series of related applications, it is necessary to stop focusing on each application individually. Instead a system family approach can be used to understand commonalities and differences between individual systems. Domain engineering methods provide guidance on how to achieve this change in focus. Generative programming can help reduce these costs even further by enabling us to move beyond the usual practice of manually searching for and then adapting relevant components. This report describes the topics of domain engineering and generative programming, examines various current methods of domain engineering, and proposes a new domain engineering method that will be used to capture and portray a problem domain.

The domain engineering process proposed by this report limits its concern to the problem space of a domain. The process consists of ten steps that can be followed to effectively understand, document and model the problem space. This process combines important aspects of existing domain engineering processes into one method. It also adds several elements that current processes lack including: the use of updated modeling techniques (the use of UML) and giving early consideration to concerns arising from the distributed heterogeneous programming environment that is becoming more and more common today. The process also specifies a grammar that can be used to describe the contents of an extended feature model. The proposed method is explained step by step and is illustrated through the use of an example problem.

Introduction

Project Summary

Most software development efforts focus on developing a single system or application. Domain engineering is a way to move that development focus from a single system to a family of systems. Generative programming is domain engineering that is taken a step further. The goal of generative programming is to be able to automatically generate an instance of a family of systems based on a specification. Generative programming requires the development of a generative domain model. This model consists of a problem space, a solution space, and the necessary configuration knowledge to map them together. This report examines the topics of domain engineering [1] and generative programming [2], examines current methods of domain engineering, and proposes a new domain engineering method that will be used to capture and portray a problem domain. The goal of this report is to use techniques from domain engineering together with those of generative programming and present a method that will allow the automatic creation of variants of instances of problems in a domain. The solution space is discussed in this report when applicable, however the process being proposed does not extend to this portion of domain engineering. The process will give guidance on how to go about capturing pertinent information when beginning development efforts for a new family of systems. It integrates ideas from current domain engineering methods and applies them to domains that may involve distributed, heterogeneous environments.

The existing domain engineering methods that were reviewed during the development of this proposed process include: Feature-Oriented Domain Analysis (FODA) [3,4], Organization Domain Modeling (ODM) [2,5,6], Family-Oriented Abstraction, Specification and Translation (FAST) [2, 7, 8, 9], Software Design Automation (SDA) [9], Domain-Specific Software Architecture (DSSA) [10], and Draco [11]. The first four of these methods are summarized later in this report. These four methods influenced the development of the proposed method in some significant way. Neither DSSA nor Draco provided any sort of a detailed plan on how to initially go about analyzing the problem domain. Consequently, they are not examined in greater detail.

Domain Engineering

Most software development efforts focus on developing a single system or application. Domain engineering is a way to move that development focus from a single system to a family of systems. A variety of methods (i.e. FODA, ODM, FAST, SDA, etc.) exist which provide varying degrees of guidance in how to achieve this change in focus. Most methods support vertical domains or families of systems. This is where entire systems are grouped into domains based on common functionality, for example financial system applications. Other methods also support horizontal domains where components or parts of systems are grouped together based on their functionality, for example database components. This project will explore several of the more detailed methods.

Domain engineering is typically divided into three different phases: Domain Analysis, Domain Design, and Domain Implementation. The Domain

Analysis phase typically involves domain scoping and modeling activities. Relevant domain information is gathered from a variety of sources including interviews with domain experts, work products of any existing systems, textbooks, known requirements for future systems, standards, etc. Domain boundaries are established and stakeholders are often identified. Domain design involves the design of a common architecture for the family of systems along with a production plan that describes how real components will be produced from the common architecture and components. Domain implementation consists of implementing the architecture, the components, and the production plan [1, 2].

Application engineering is the follow-up process where a new instance of a family of systems is developed by using the results from domain engineering.

Generative Programming

The goal of generative programming is to be able to automatically generate an instance of a family of systems based on a specification. The achievement of this goal requires the development of a model of the relevant product family, some way to specify new products, the availability of components from which the product can be assembled, and a means of mapping the problem specification to the required components using an implemented configuration generator.

Generative programming requires the development of a generative domain model. This model consists of a problem space, a solution space, and the necessary configuration knowledge to map them together. The problem space consists application concepts and features that the user would like to have

available. This problem space can be explored using techniques from domain engineering. The solution space is made up of the component implementations in all of their potential combinations. Configuration knowledge takes into account considerations such as illegal feature combinations, default settings, default dependencies, construction rules, and optimization rules. Configuration generators are created to implement this knowledge. A configuration generator (often referred to simply as a generator) is responsible for checking to see if the system can be built, completing the specification by computing defaults, and assembling the implementation components [1]. Separation of the problem and solution spaces allows each to develop somewhat independently.

An important concept to keep in mind when designing the problem space is that application programmers should only be required to specify as much information as is necessary to identify potentially appropriate components from the generative library. The programmer should be allowed to specify details or elect to supply some of his own implementations for specific aspects if desired.

The main steps necessary in generative programming are: domain scoping; feature and concept modeling; designing a common architecture and identifying implementation concepts; specifying domain specific notations for ordering systems; specifying the configuration knowledge; implementing the components; implementing the domain specific notations; and implementing the configuration knowledge using generators [2].

Existing Methods of Domain Engineering

Feature-Oriented Domain Analysis (FODA)

FODA is a domain analysis method that was developed at the Software Engineering Institute. The method was based on a detailed study of other domain analysis approaches. The idea of a domain in the context of this method relates to a family of systems. The main product of this method is a structured framework of related modes that document the results of the domain analysis.

The primary goal of this method is to develop products that are generic and widely applicable within a domain. FODA achieves this goal by applying two concepts: abstraction (also referred to as generalization) and refinement (also referred to as specialization). The idea is to abstract away any differences between applications in a domain. Domain products may then be created which are applicable to the entire system family. Specific applications may then be developed using refinements of the basic domain products. The refinements will reintroduce any factors that make each application unique.

FODA consists of three phases, the first two of which are much more effectively documented. The first phase is called Context Analysis. The goal of this first phase is to provide the context of the domain, which basically means to define its scope. The second phase consists of domain modeling. This phase looks at commonalities and differences between applications within a domain. Several types of models are produced. The last phase consists of architectural modeling. This consists of providing a software solution for creating new applications for the domain. Each phase will now be explained in greater detail.

As previously mentioned, the main purpose of context analysis is to define the scope of a domain. Relationships between the domain and external elements are analyzed, and variability of the relationships and the external conditions are evaluated. The final results of this analysis are documented in a context model. This context model consists of one or more structure diagrams as well as data-flow diagrams. The purpose of the structure diagrams is to portray how the target domain is related to other domains. Data flow diagrams show how data flows between the target domain and any other entities that it communicates with. Detailed information is gathered for each entity identified.

The Domain Modeling phase is comprised of three major activities including feature analysis, information analysis, and operational analysis.

The purpose of Feature Analysis is to capture and model the end-user's understanding of general capabilities or features of applications within a domain. The feature model is used to portray the common features and the differences between applications within a domain. Features are defined as "the attributes of a system that directly affect end-users". Features may be defined as optional, mandatory, or alternative. Also included in the feature model are composition rules which define semantics existing between features that are not expressed in the core feature diagram as well as any rationale which should be considered when choosing from alternative features. Full documentation of a feature model will include: a structure diagram showing a hierarchical decomposition of features that indicates which ones are mandatory, optional, or alternative, a definition of each feature, and any composition rules. This model will serve as a

communication medium between users and developers. The process for creating this model consists of the following steps: collecting source documents, identifying features, abstracting and classifying the identified features, defining the features, and validating the model.

During Informational Analysis (also called Entity-Relationship Modeling) the domain knowledge that is essential for implementing applications in the domain is defined, analyzed and captured. This knowledge is represented in terms of domain entities and their relationships and is made available for the derivation of objects and data definitions during operational analysis and architecture modeling. The model may be an entity relationship (ER) model, an object-oriented (OO) model, or a semantic network.

Operational Analysis (or Functional Analysis) identifies the behavioral and functional commonalities and differences of applications within a domain. The specification of function describes the structure of an application in terms of inputs, outputs, activities, internal data, logical structures, and data-flow relationships. The specification of behavior describes how an application responds in terms of events, inputs, states, conditions, and state transitions. An abstract model of the functionality of the family of applications is defined at the top level. As the abstract model is refined alternative and optional features are embedded into the model. Any issues raised during analysis or resolution are also captured.

The final phase of FODA is Architectural Modeling. The purpose is to provide a software solution to the problems defined in the domain modeling

phase. The model must address these problems in a way that the model can be adapted to any future changes in technology or in the problem itself. The proposed solution is to use architectural layering. The architecture is defined at different levels of abstraction so reuse may occur at any layer. The FODA methodology defines four layers and focuses on the top two: the Domain Architecture Layer and the Domain Utilities layer. The result is the development an application domain-oriented architecture. This is a high level design which packages functions and objects into software modules. Concurrent tasks are identified and communication and synchronization between tasks is defined using the DARTS (Design Approach for Real-Time Systems) notation. Finally, each task is designed as a sequential program [3,4].

Organization Domain Modeling (ODM)

Organizational Domain Modeling (ODM) is a formal, tailorable approach to domain engineering. The method was formalized by Mark Simos of Organon Motives Inc. and was funded by the ARPA STARS program. ODM is most successful when used to support domain engineering projects for domains which are mature, reasonably stable, and economically viable.

One issue, which is heavily emphasized by this method, is understanding stakeholders and their individual goals. As Simos says "Stakeholder issues, always a potential problem in any project, turn out to be critical risk factors in domain engineering, which by definition involves designing for multiple contexts of use." [5]. The interests of stakeholders are reconsidered at critical points throughout the domain modeling life cycle.

The overall goal of ODM is to systematically turn software artifacts from legacy systems into reusable assets that can be useful for future development efforts. ODM is applicable to both families of systems (vertical domains) as well as sub-portions of systems (horizontal domains). The method may therefore be useful both in re-engineering portions of legacy systems and in guiding the development of new systems.

The method is made up by three distinct phases: Plan Domain Engineering, Model Domain, and Engineer Asset Base. Each phase is divided into three subphases, each of which requires the performance of three tasks. The method is explained in more detail in the following paragraphs.

The Plan Domain phase focuses on understanding stakeholders, scoping the domain, and defining relevant domain boundaries. The first subphase, set objectives, is made up of determining the identities of relevant stakeholders, understanding their objectives as well as the overall project objectives, and prioritizing among the identified stakeholders/objectives. The second subphase, scope domain, includes tasks such as identifying and characterizing potential domains in areas of interest, defining the selection criteria by which a domain will be chosen, and finally selecting the domain to proceed with. The third subphase, define domain, consists of defining the domain boundaries through both rules and examples of systems which are to be included, identifying the main features of systems falling with in this domain, and analyzing the relationships between this and other domains. These scoping steps attempt to make the boundary decisions

explicit and public, helping to avoid later conflicts over what is and is not included.

The Model Domain phase is concerned with gathering and documenting relevant domain information. Acquiring domain information is accomplished by first planning the task, then collecting information from domain experts, system users, existing legacy documentation, literature studies, etc… This subphase is complete when the information is integrated and the most relevant system features have been identified. The next subphase involves describing the domain. This first entails developing a lexicon of domain terms that essentially captures the specific language of the domain. The next steps are to model the semantics of key domain concepts and then to model the variability of these concepts through identification and representation of features. The final subphase is refining the domain. This involves integrating existing models into a consistent overall model, modeling the trade-offs for why certain features are used or not used and finally clustering and experimenting with different combinations of features.

The final phase of ODM, Engineer Asset Base, consists of scoping, architecting, and implementing an asset base for the relevant domain. Scoping the asset base consists of correlating features with customers then prioritizing among them, and selecting which will be implemented. Architecting the asset base is accomplished by determining external and internal architecture constraints, and defining an architecture based upon these. The final subphase, implementing the asset base, consists of planning the implementation, implementing assets and finally implementing the infrastructure including asset retrieval and qualification

mechanisms. Traceability from features back to exemplar artifacts should be preserved so that developers have access to potential prototypes [1,6].

Family-Oriented Abstraction, Specification and Translation (FAST)

Family-Oriented Abstraction, Specification and Translation (FAST) is a domain engineering method which focuses on software families or product lines. A software family is a group of products that share common features and meet the needs of a particular market area. FAST was developed by David Weiss et al. at Lucent Technologies Bell Laboratories and was greatly influenced by the Synthesis method. FAST has been applied to over 25 domains at Lucent and has been shown to reduce development time and cost by 60 to 70% for new family members [1].

The overall goal of FAST is to create processes and assets for producing new members of a program family as fast and cheaply as possible. Steps to achieving this goal include finding appropriate abstractions for the family, creating a language to describe them, and creating the tools necessary to translate descriptions of family members into software and documentation deliverables. The FAST process attempts to provide guidance in each of these areas. FAST consists of two subprocesses: domain engineering and application engineering. During domain engineering, the software family is defined and an environment for producing family members is developed. Application engineering then uses this application environment to produce family members [7]. Feedback from use of the application engineering environment may suggest potential changes. The

domain engineering subprocess shall be explained in detail for the remainder of this section.

The domain engineering subprocess begins with collecting and documenting the knowledge pertaining to a particular domain through a method called the commonality analysis. The product of this method is the Commonality Analysis document, which is created through a series of moderated meetings with domain experts. The document consists of seven sections described as follows. It begins with an Overview, which briefly describes the domain and how it is related to other domains. The Overview is followed by a Definitions section that records key technical terms and their meanings. The third section is Commonalities. Commonalities are assumptions that are true for every member of a product family. Examples may include common attributes or functionality. The fourth section lists the Variabilities. Variabilities describe how individual members of a product family may differ. Examples include optional attributes or functions. This is followed by Parameters of Variation. This section captures the possible values of the variabilities including the specification of legal values, any default values, and the binding time for each value. The sixth section, Issues, captures any significant problems that were encountered by the team. When issues are resolved, the alternatives, chosen solution, and any applicable discussion are all recorded. The final section, Scenarios, captures examples used when describing commonalities and variabilities (usability and variability scenarios) [8]. Team members then review the CA document once an iteration is complete. The review inspects the content and structure of the document. This review is followed by

another review performed by engineers who are knowledgeable of the domain but who did not participate in the making of the document.

The next step in the domain engineering subprocess is to define the Decision Model. This decision model should consist of the set of all requirements and engineering decisions that must be resolved by an application engineer in order to construct a new member of the product family. The model lays out these decisions in an appropriate order. This model is derived from the Commonality Analysis document.

Following the definition of the decision model is the design of the Application Modeling Language (AML). This is what the application engineers will use to specify a new product family member; therefore any decisions identified in the decision model must be expressible in the AML. These descriptions are then generated into working applications and documentation. Either a compositional or a compiler approach may be used for the AML. If a compiler approach is used, only the abstract modules and the parameters of variation are specified in a domain. On the other hand, if a compositional approach is used, then a domain design specifying the architecture of the family is created. If multiple components interact, then a compositional mapping between the AML and the components specified in the design must be created.

The final step in the domain engineering subprocess is to actually implement the family/domain. This consists of designing and creating the application engineering environment. Both generation and analysis tools may be created. Generation tools take the AML specification as input and produce code

and documentation. Analysis tools may be used to analyze member specifications and produce feedback concerning consistency and or completeness of the specification, performance estimates, comparisons of different models, etc. Libraries of common assets are also specified, including code and documentation templates. Following the design specification, appropriate tools and their supporting data should be found or built, and any applicable templates should be written in a suitable language. Depending on whether the compiler or compositional approach was used for the AML, either a compiler or a compositional mapping that composes applications from library templates must be implemented. Finally the application-engineering environment should be documented [9].

Software Design Automation (SDA)

Software Design Automation (SDA) is a method for developing tool-supported formal specification languages or application generators that is being developed by the Pacific Software Research Center at Oregon Graduate Institute (PacSoft). It is intended to be applied to mature, stable problem domains. SDA uses mathematical models to express the problem requirements of a particular problem domain. A key concept behind SDA is to use Denotational Semantics to capture the complete specification of the domain specific language. Denotational Semantics is a type of language definition that expresses the meaning of a phrase in terms of its constituent parts. Another key concept in SDA involves the use of Monads. Monads are a way to structure semantics that expresses abstraction over notations or compositions. Functional languages are used to formally capture a

language specification. They capture the user's view of the domain as a data type and are used to model the solution space of the domain. They are then used to implement the semantics-based interpreters from the problem space to the solution space. When the interpreter has been built, a specification can be written in the domain specific language to run with the interpreter to produce a new component.

SDA is comprised of the following three phases: analyze the domain, define the language, and implement the generator and support products. These phases are applied in an iterative fashion. When and how many times to iterate is dependent upon the domain. The following several paragraphs will give an overview of the first two phases of the SDA process with an emphasis on phase one. The third phase will not be covered since it is not applicable to this project.

Analyzing the domain is broken down into five main activities beginning with capturing a written definition of the domain. The purpose is to informally document information relevant to the domain, giving the language design team an initial understanding, and to identify domain experts for later feedback and asset validation. This documentation should include a high level problem statement, a model of the surrounding system architecture including the interfaces of neighboring domains/entities (for use in defining domain boundaries), an initial set of domain requirements including expected behavior and constraints of any instances, and a workflow analysis, which among other things captures common notations and concepts used by engineers.

The second activity, which actually runs concurrently with the first one, involves the formulation of a formal domain model. This is essentially the problem view, as the user perceives it. It is important to capture an initial model early on in order to focus the information gathering activities. Mathematical structures should be used to abstract and formalize domain concepts when possible. The domain model can be captured directly as a parameterized functional language data type. Specification of the parameters should then determine an instance of the domain.

The next major activity is to define the solution model. The solution model should partition the domain into the components and connectors necessary to provide the needed functionality. Since the problem domain is assumed to be mature and stable, there should be solutions in existence that can be analyzed. Interfaces to functions or modules should be used to specify legal interactions. These should be accompanied by descriptions of how the modules interact. Finally the overall system architecture should be documented.

Capturing the interface to the legacy environment is the fourth major activity. Analyzing the run time environment of any legacy system that exists will ensure that the generated code will be able to integrate properly with it.

Analyzing the domain ends by validating all models developed so far. A preliminary validation of the models may be achieved by showing that a solution can be calculated for a particular problem described in the domain. This example problem should be taken from domain experts who should observe the process of how the solution is obtained.

The second major phase of the SDA process is to define the domain specific language. The goal of the first half of this phase is to capture the initial language definition. One portion of the language definition is to specify its type system. This includes specifying all atomic entities and entity composition rules. A user feedback plan should also be developed so that structured feedback can be received from potential users as the language develops. The workflow analysis and user view domain model, which were captured in phase one, should be used when developing the core syntax. The language must be expressive over all domain entities and solution parameters allowing application engineers to specify new members of a product domain.

The second half of defining the language is to formalize the semantics. The purpose is to produce an interpreter that captures the external semantics of the language formally in a denotational style. The syntactic terms of the language should be given semantic descriptions. The language should contain a collection of phrases, each of which specifies some function. Monads should be used to capture any useful abstractions for structuring the solution. Semantics of any new language are based upon semantics of existing similar languages [9].

Proposed Method for Modeling the Problem Space of a Variable Domain

Introduction

There are various methods of domain engineering already in existence. Several of these have been discussed in detail in previous sections of this report. Although there is some overlap in the processes presented by these methods, each in some way offers an individual contribution to better understanding the problem analysis phase of a new domain. The first item that is missing from all of them is any consideration that the problem is likely to be embedded in a distributed heterogeneous environment. While consideration for this likely outcome will take place in a larger degree during the solution phase of domain analysis, preparations should begin when examining the problem. Additionally, many of the methods advocate use of models that are rather outdated. None of them recommend the use of UML. The Object Management Group unanimously adopted UML in 1997 as a standard [12]. Since then UML has quickly emerged as the industry standard for modeling enterprise software systems in domains ranging from finance and manufacturing to health and telecommunications [16]. The method proposed by this report attempts to integrate ideas from existing domain engineering methods with the use of relevant UML diagrams, as well as giving early consideration to concerns present in distributed heterogeneous systems.

Method

This proposed method consists of ten distinct steps. In the following paragraphs, each of these steps will be explained in detail. Each will then be further illustrated through the use of an example application. The application that will be used is a financial system that will compute net income for either a manufacturing or a wholesaling business. This system should include relevant sources of revenue and expense when computing the final figure. Since accounting is not the primary focus of this illustration, the sample problem will be somewhat simple in nature. A summary outlining the basic steps of the process can be found at the end of this section.

Step I: Describe the Problem Domain

The goal of step one is to gain an initial understanding of the new problem domain. This is important because it gives everyone involved in the project an initial understanding of what needs to be accomplished. This should begin with the development of a problem statement. Although, this may not be very detailed at this point in time, it is important to get the overall goal down on paper at the beginning of this process. The next item to be produced is a general description of the capabilities that applications falling within this domain should possess. This should include any desired properties of the system family that have not yet been captured in the problem statement. The final item to be produced is a list of any existing applications that would fall under the description of this domain.

Exhibit 1 shows the output of this first task for our example net income domain.

Exhibit 1: Step I in the Net Income Domain

Problem Statement:

To create a financial system that is able to determine earnings for either a manufacturing or wholesaling client.

Description of General Capabilities:

The systems should be able to compute net income when given applicable revenue and expense figures. These figures should be fed by a separate general ledger system or input by a user if a general ledger system does not exist.

Existing Applications:

There are no existing applications that would fall within this new domain.

Step II: Identify Stakeholders

The goal of step two is to produce a list of people who will be involved with the project in any significant way. These people may have oversight responsibilities or they may be a resource for better understanding the domain. This group of stakeholders and domain experts may include some or all of the following groups of people: senior management, project management, end users, customers, other service recipients, other service providers, investors, developers, and regulators. Be sure to include developers currently working on the project as well as those who may have worked on any legacy systems that are currently in place. This second group of developers may be knowledgeable about available components that may be reused. If the project is one that will have an overseas audience via the Internet it is important to include users from those locations. This will allow the project team to become aware of any additional or different needs that this user group may have.

For our net income domain four groups of people were recognized as important stakeholders. Senior management and project leadership were included to ensure that they remain aware of ongoing progress. Application engineers are important as a source of information about any previous projects that may be related to the current project. They will also be the key people later developing the software solution. Financial analysts or accountants are the end users of this system. They are also the people who can provide information regarding the different types of revenues and expenses that our domain will need to consider.

Exhibit 2: Step II in the Net Income Domain

Potential Stakeholders and Domain Experts:

Senior Management
Project Leaders
Application Engineers
Financial Analysts/Accountants

Step III: Expand Domain Definition

Step three consists of conducting a series of moderated meetings with the types of stakeholders identified in the previous step. A variety of tasks should be accomplished as an outcome of these meetings. The first task is to identify the overall project objectives. These should consist of the overall goals by which the success of the project will be measured. A variety of stakeholders will have input regarding the contents of this list. The second task is to expand the description of the domain. The domain experts should be able to assist a great deal. Any performance constraints and other types of non-functional requirements that all systems in this domain must meet should be included in this section. The third

task in this step is to define the boundaries of the domain. Any other domains or entities that instances of this domain will communicate with should be noted in this section. This section will help define what types of applications are and are not included in the project. Giving examples of specific applications that do and do not fall with the new domain may be helpful. The final task in step three is to generate a good list of potential sources of domain information. The following are some sources that may be helpful: information regarding applications that have previously been developed and fall into the domain (i.e. requirements analysis documents, existing code, etc.); developers who may have worked on related applications; other domain experts not previously identified; users of existing applications; people or documents that have information about entities/other domains with which applications from this domain must communicate; other existing sources of information including textbooks, standards, etc. This list is not intended to be a comprehensive list of information sources, but should serve as a starting point.

Figure III shows the work products that were created when the process was applied to our net income domain.

Exhibit 3: Step III in the Net Income Domain

Project Objectives:

1. Finish all phases of domain analysis in the time specified by senior management.
2. Produce quality work products at all stages from planning through product development.
3. Produce complete, high quality documentation as specified by the process.
4. Successfully implement the project while remaining at or under the specified budget.

Domain Description:

This domain is made up of financial systems that will compute net income when given applicable revenue and expense figures. These figures should be fed by a separate general ledger system or input by a user if a general ledger system does not exist. The system should return the correct earnings figure to the user in a speedy manner (less than n seconds).

Domain Boundaries:

This domain will only be concerned with earnings derived from manufacturing or wholesaling activities. This domain will not capture earnings for businesses whose revenues and/or expenses vary in nature from that of a wholesaler or manufacturer. For example, banking clients will not be accommodated. The domain may interact with a general ledger system. If a general ledger system exists, it will retain responsibility for recording individual accounting entries. The domain will also include a user interface to capture user entries.

Potential Sources of Information:

1. Financial Analysts/Accountants – May have knowledge regarding features and financial rules.
2. Application Engineers – May have applicable knowledge from past developments of related applications. May also have knowledge regarding system requirements at various sites.
3. Textbooks – May have formal definitions of key terms.
4. GAAP (Generally Accepted Accounting Standards) – This accounting standards guide may have guides on how to calculate income.

Step IV: Acquire and Document Relevant Domain Information

The objective of step four is to gather and document relevant domain information. The tasks identified in this step should be repeated as many times as necessary to gain a thorough understanding of the problem. The first logical task is to interview the people previously identified in step three. The interviewer should keep detailed notes of information gathered. Important items to document include the definitions of common words or expressions particular to the domain as well as capabilities that will be required for applications that fall within the domain. The next task is to consult other information sources identified in step three to fill in any knowledge gaps about the domain. The third task is to create a dictionary of common terms. Exhibits 4a and 4b contain an example of such a list for the net income example.

Exhibit 4a: Domain Dictionary for the Net Income Domain

Dictionary of Common Terms:

ADVERTISING AND PROMOTION – The cost associated with the practice of bringing to the public's notice the good qualities of something in order to induce the public to buy or invest in it.

BAD DEBT EXPENSE – Cost associated with writing off money that is owed to you that you cannot collect.

COST OF GOODS SOLD – The direct cost to the business owner of those items which will be sold to customers.

COST OF GOODS MANUFACTURED – Includes all expenses directly associated with the manufacturing of goods.

COST OF GOODS WHOLESALED – Includes costs associated with the purchase of goods for resale.

DEPRECIATION - is the amount of expense charged against earnings by a company to write off the cost of a plant or machine over its useful live, giving consideration to wear and tear, obsolescence, and salvage value.

Exhibit 4b: Domain Dictionary for the Net Income Domain (continued from 4a)

Dictionary of Common Terms:

DIRECT LABOR – The cost of workers who transform the materials into a finished product at some stage in the production process.

DIRECT MATERIALS – Those materials that can be feasibly identified with the product.

GENERALLY ACCEPTED ACCOUNTING PRINCIPLES (GAAP) - Term used to describe broadly the body of principles that governs the accounting for financial transactions underlying the preparation of a set of financial statements. Generally accepted principles are derived from a variety of sources, including promulgations of the Financial Accounting Standards Board and its predecessor, the Accounting Principles Board, and the American Institute of Certified Public Accountants. Other sources include the general body of accounting literature consisting of textbooks, articles, papers, etc.

GENERAL LEDGER - is the set of accounting records that show all the financial statement accounts of a business.

MANUFACTURING OVERHEAD – is the costs associated with providing and maintaining a manufacturing or working environment. For example: renting the building, heating and lighting the work area, supervision costs and maintenance of the facilities. Includes indirect labor and indirect material.

NET INCOME – is the company's total earnings, reflecting revenues adjusted for costs of doing business, depreciation, interest, taxes and other expenses.

Task four is to identify common features for applications that will fall within the domain. These features should be classified based on whether they will appear in every application created (common features) or whether they will appear in only some of the applications (variable features). For example, in the net income domain all applications will need to capture sales revenue, but only some will need to capture other revenue. Any composition rules and/or mathematical formulas representing relevant business knowledge should also be documented at this time. An example from the net income domain would be to note how net income is composed of revenue and expenses. Exhibit 5 shows the

common and variable features as well as the composition rules that were identified for the net income example.

Exhibit 5: Common and Variable Features for the Net Income Domain

Features of Applications within the Net Income Domain:

1. Net Income – Common feature
2. Revenue – Common feature
3. Sales Revenue – Common feature
4. Other Revenue – Variable feature
5. Cost of Goods Sold – Common feature
6. Cost of Goods Wholesaled – Variable feature
7. Cost of Goods Manufactured – Variable feature
8. Purchases – Variable feature
9. Direct Materials – Variable Feature
10. Direct Labor – Variable feature
11. Selling and Administrative Costs – Common feature
12. Sales Salaries – Common Feature
13. Advertising and Promotion – Variable feature
14. Depreciation – Variable feature
15. Bad Debt Expense – Variable feature
16. Other Expenses – Variable feature

Composition rules:

1. Net income must include revenue and expenses.
   Net Income = Revenue – Expenses

2. Revenue must include sales revenue and may include other revenue.
   Revenue = Sales Revenue + Other Revenue

3. Expenses must include cost of goods sold and selling and administrative expense, and may include other expenses.
   Expenses = Cost of Goods Sold + Selling and Admin. Expense + Other Expenses

4. Cost of goods sold may include either cost of goods manufactured or cost of goods wholesaled.
   Cost of Goods Sold = Cost of Goods Wholesaled | Cost of Goods Manufactured

5. Cost of goods wholesaled must include purchases.
   Cost of Goods Wholesaled = Purchases

6. Cost of goods manufactured must include direct materials, direct labor, and manufacturing overhead.
   Cost of Goods Manufactured = Direct Materials + Direct Labor + Manufacturing Overhead

7. Selling and administrative may include one or more of the following: sales salaries, advertising and promotion, depreciation, and bad debt expense.
   Selling and Admin. Expense = Sales Salaries + Advertising and Promotion + Depreciation + Bad Debt Expense

The final task for step four is to document any knowledge regarding communication that takes place between the new domain and any outside domain or entity. The type of communication that takes place as well as any established communication interfaces that must be used should be documented. If the interfaces have not yet been determined, this fact should be noted. This is also a good time to note any concerns that may arise from the distributed nature of any outside entity. For example if frequent communication must take place with a mobile entity, the problem may need to take limited connectivity or additional security concerns into account.

Exhibit 6 shows the result of this task for the example net income domain.

Exhibit 6: Communication with outside Entities for the Net Income Domain

Communication Between Entities:

The system will receive relevant revenue and expense figures from either a general ledger system or a user. Since the general ledger system may vary for each client, there is no communication interface that can be noted at this time. This is a point of variation that must be planned for. The interface to the general ledger may also change for the same client in time. A user interface must also be created. It must enable the user to enter relevant revenue and expense figures when no general ledger system is present, and view the results of the net income calculation.

Step V: Model the Domain

The goal of step five is to model the new domain. The models will enable everyone involved in the project to better understand different aspects of the problem. An extended version of the FODA feature diagram is used as well as use case diagrams, sequence diagrams, and collaboration diagrams from the Unified Modeling Language [12]. The domain dictionary should be updated to reflect any new terms or new detail about existing terms that is discovered during the modeling phase.

The first task is to create a feature model similar to the one used in the FODA method of domain engineering. This uses an extended feature diagram in order to capture common distributed computing concerns. Like FODA, each feature is represented as a box. These features are then arranged in a hierarchical manner. Each feature is decomposed until it is present at the level of interest to the user. The leaf nodes of the feature diagram are of particular importance. Assuming that the system will be implemented as a collection of distributed components, each leaf node represents the correlating component that will implement its specified functionality. The box has a solid outline if the leaf node feature must be present in order for the system to run correctly. On the other hand, if the leaf-node feature represents system functionality that is not essential, its box is outlined in a dashed manner. All non-leaf node features are represented with solid box outlines.

Generally, each feature has a filled circle above it if it is a common feature across all applications within the domain. While, if the feature is variable/optional across different applications, the circle is not filled in. There is one exception to these rules that may occur. Sometimes an optional feature may have mandatory children. In this case, the optional parent feature has an unfilled circle while any children have filled circles. An example of this situation occurs in the net income domain. A client may be a manufacturer or a wholesaler. Therefore, the client will incur either a cost of goods wholesaled expense or a cost of goods manufactured expense. If a client is a wholesaler and incurs cost of goods wholesaled, then this must include a purchases expense.

A solid triangle means that one or more of the decomposed features must be present. A hollow triangle means that exactly one of the more detailed features is present. Any communication found to be necessary between leaf-node features (components) should be represented by a dashed line. Exhibits 7a and 7b contain the feature diagrams applicable for the net income example. Exhibit 7a begins with the business domain and further decomposes this into wholesalers and manufactures. From there the two sub-trees that begin with net income are very similar. Feature modeling enables us to represent variation in such a way that we can condense this first feature diagram into the one contained in exhibit 7b without losing any relevant information. This feature diagram begins with net income since this is the focus of our problem domain.

In the net income domain, all components (leaf-node features) must be present in order for net income to be correctly computed, so all of the boxes are solid. Also, this is not a domain where the features would need to communicate with each other so no dashed lines are present. This diagram can be easily derived from the information captured in the fourth step.

Exhibit 7a: Detailed Feature Diagram for the Net Income Domain

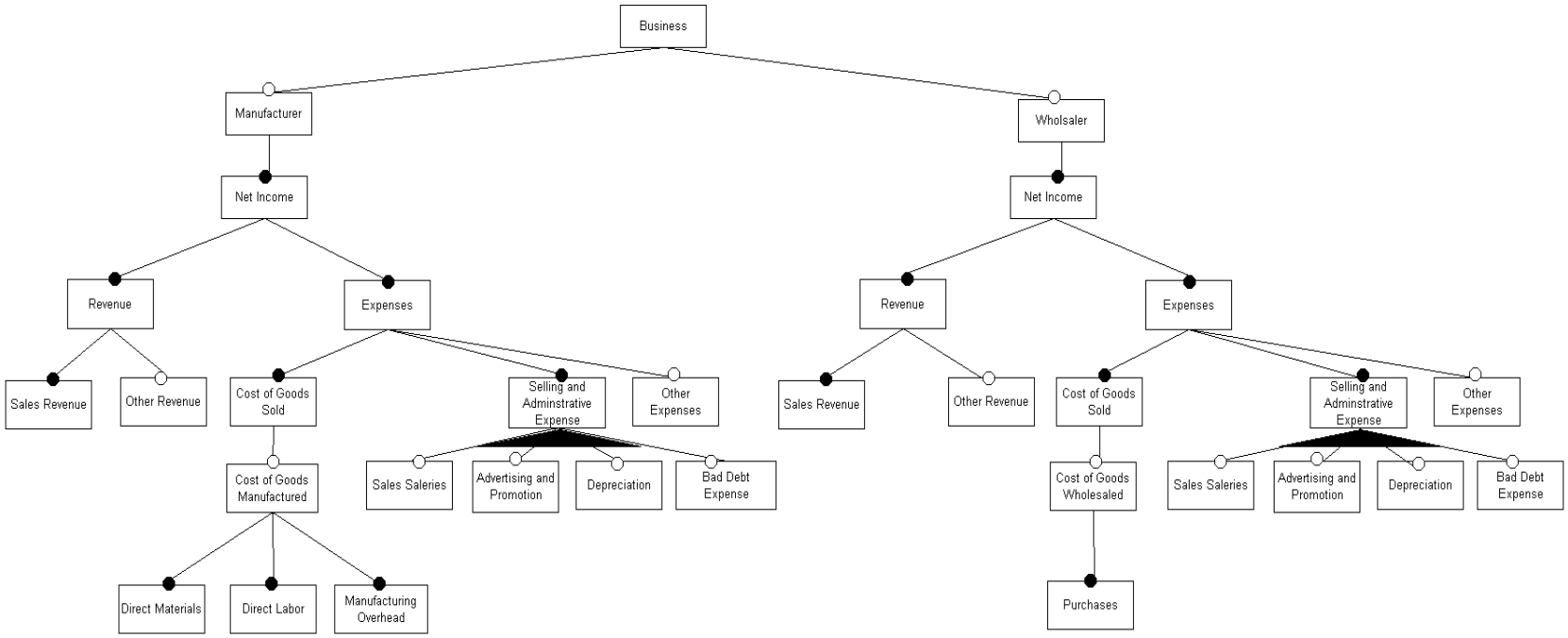
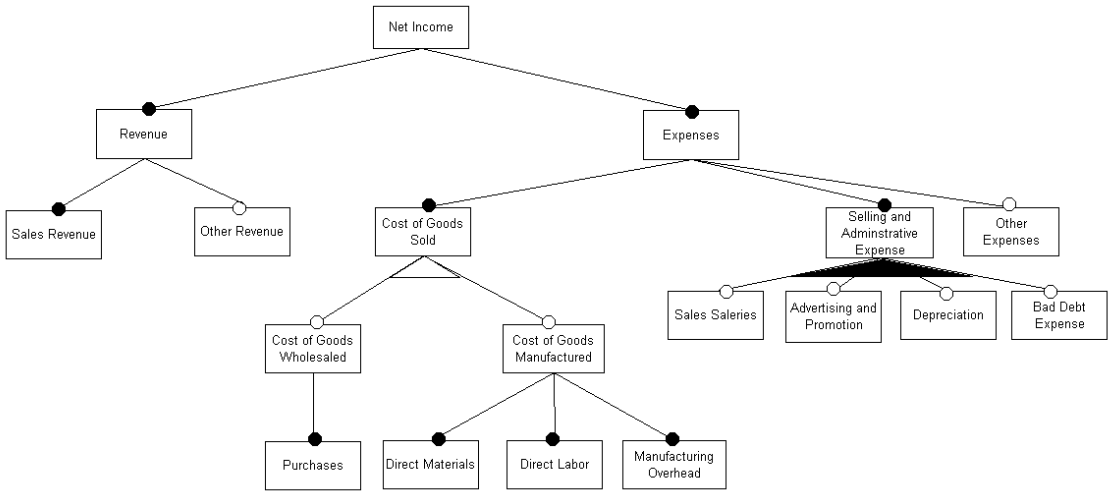


Exhibit 7b: Condensed Feature Diagram for the Net Income Domain



The second task for step five is to create a series of use cases that show how any actors (outside entities or applications from other domains) will interact with applications within this system. Use cases are a good tool for showing the functionality of the system as it is perceived by outside users (actors). The number necessary will depend upon complexity of the project. Exhibits 8a, 8b, 8c and 8d show the four possible use case diagrams for the net income domain. Exhibit 8a shows what happens when net income is calculated for a manufacturing client using a general ledger system. Exhibit 8b is the same as 8a except now the client is a wholesale company. Exhibit 8c calculates net income for a manufacturer when there is no general ledger system available. Exhibit 8d is the same as 8c except the client is a wholesale company.

Exhibit 8a: Use Case Diagram 1 for the Net Income Domain

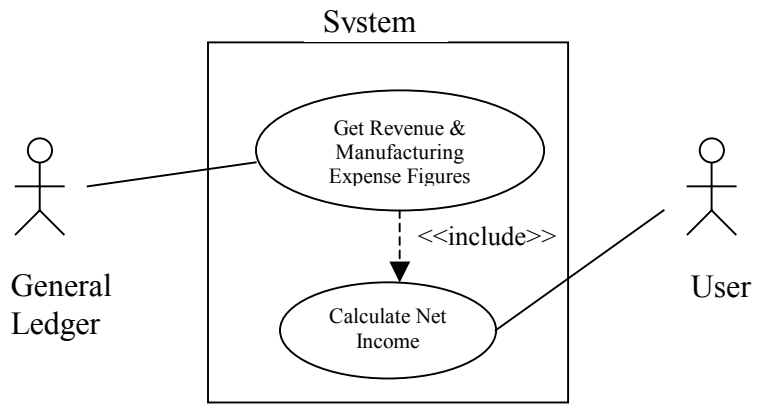


Exhibit 8b: Use Case Diagram 2 for the Net Income Domain

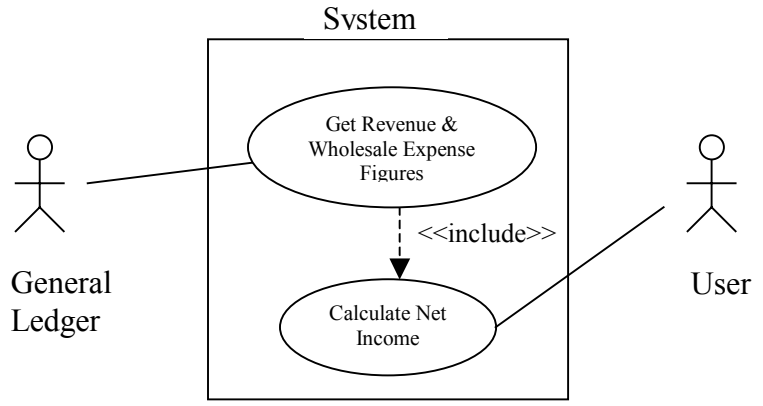


Exhibit 8c: Use Case Diagram 3 for the Net Income Domain

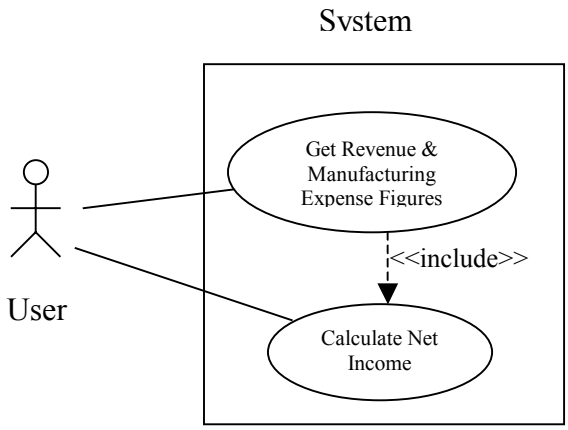
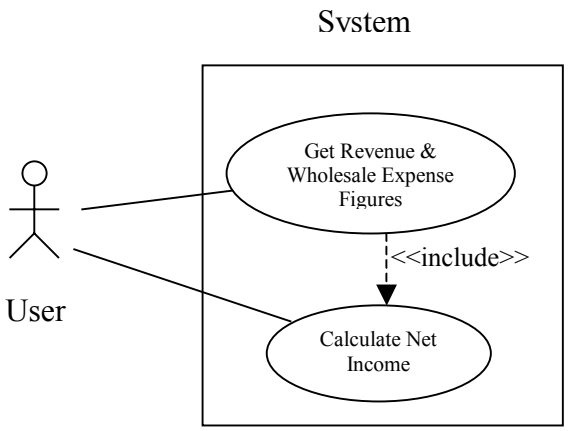


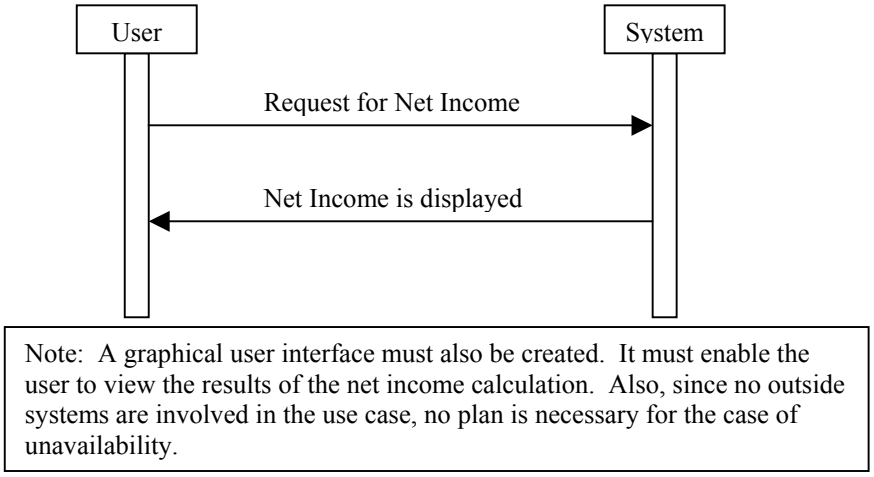
Exhibit 8d: Use Case Diagram 4 for the Net Income Domain



The next task for step five is to create one or more sequence diagrams to show how this new domain would communicate with other domains/entities. Sequence diagrams are good at showing how a set of messages is sent over time. At a minimum, one sequence diagram should exist for every use case identified by the use case diagrams. If the use case is more complex, it may have a number of alternative paths through it depending on data values given. If such alternative paths exist, a separate sequence diagram should be created for each one. Each sequence diagram should include a text description of what should take place if any of these outside systems become unavailable. Also, if the method by which the system will communicate with these outside entities is unknown this should be documented as a part of the diagram. This is done to take into account potential communication uncertainty between geographically dispersed systems. Five of the sequence diagrams that apply to the example net income domain are shown in

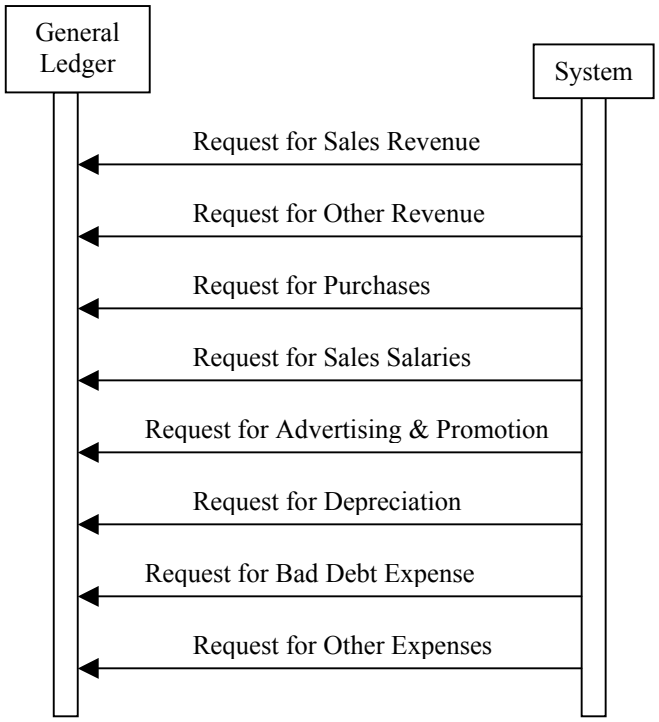
exhibits 9a, 9b, 9c, 9d, and 9e. These five sequence diagrams show the potential variations that exist when all of the optional components (shown in Exhibit 7b) are present. If the net income example was comprehensively being modeled as it would in the real world, it would be necessary to complete a separate sequence diagram for each potential variation in the presence of optional components.

Exhibit 9a: Calculate Net Income



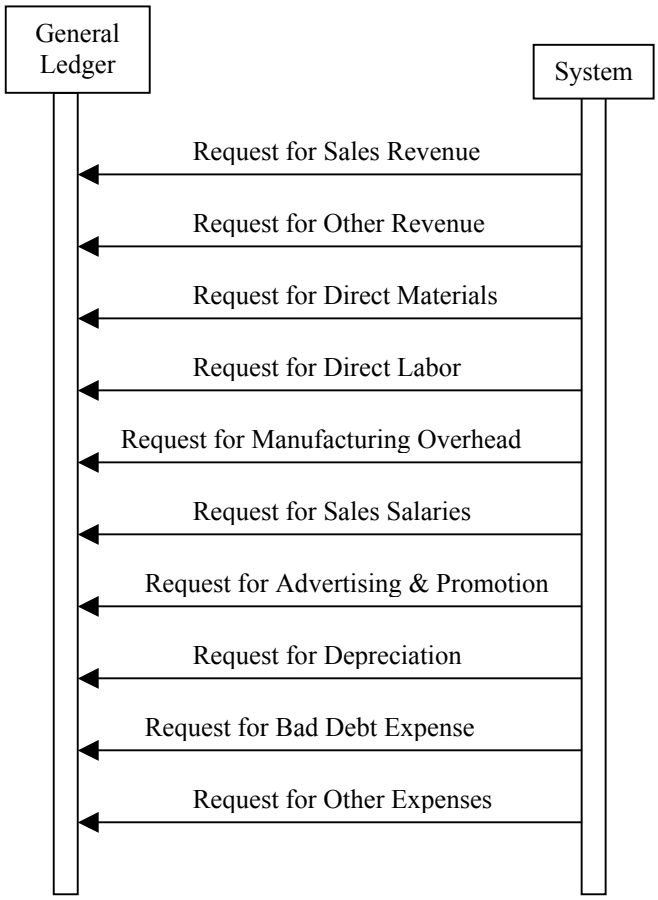
Note: A graphical user interface must also be created. It must enable the user to view the results of the net income calculation. Also, since no outside systems are involved in the use case, no plan is necessary for the case of unavailability.

Exhibit 9b: Get Revenue and Wholesale Expense Figures from the General Ledger



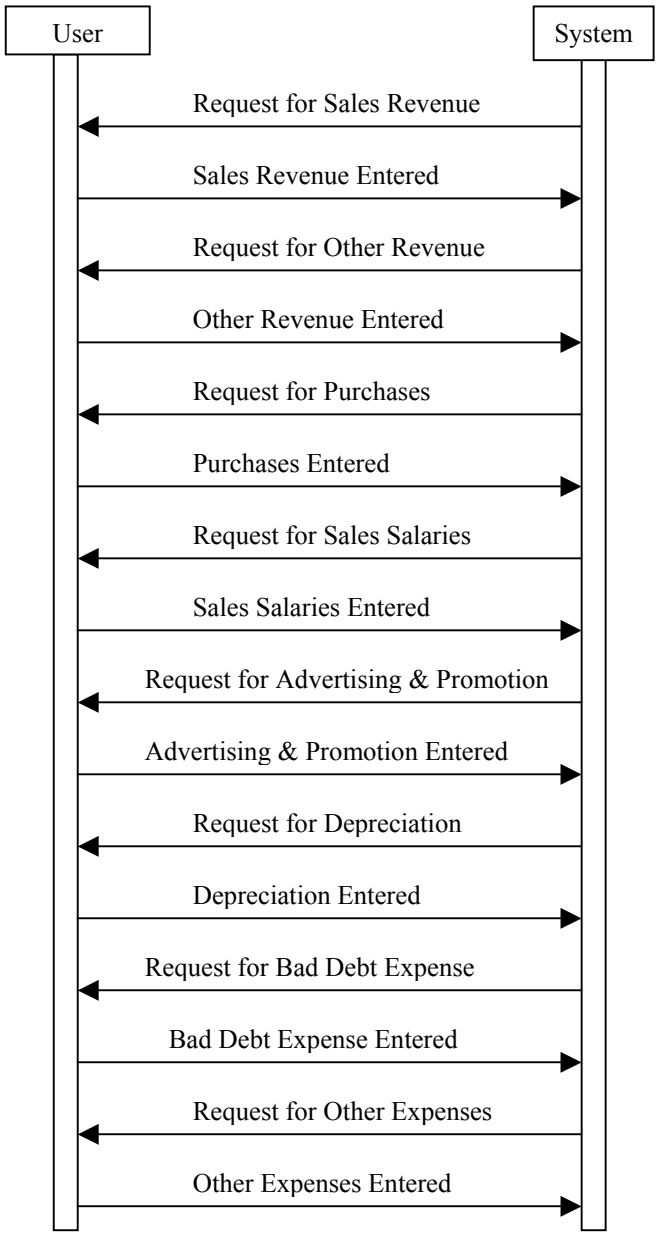
Note: If the general ledger becomes unavailable, the user will be prompted to manually input each figure (exhibit 9d).

Exhibit 9c: Get Revenue and Manufacturing Expense Figures from General Ledger



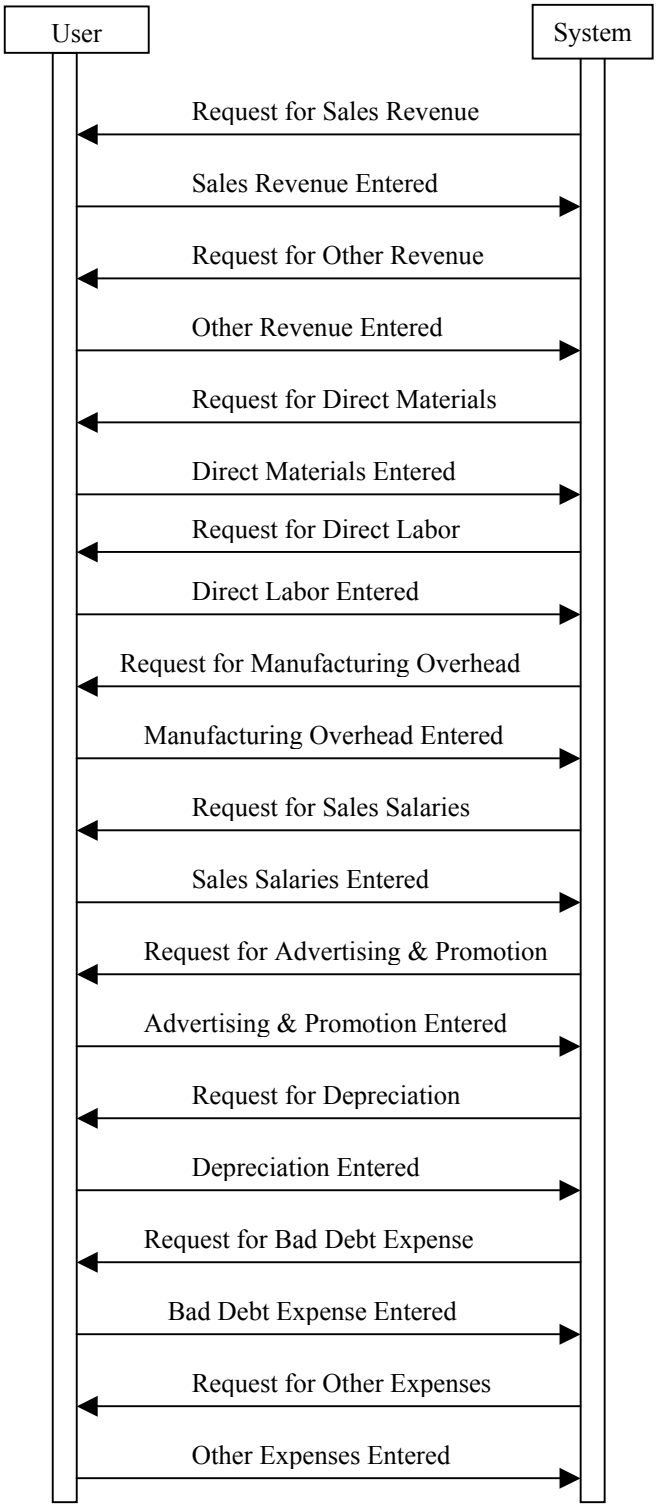
Note: If the general ledger becomes unavailable, the user will be prompted to manually input each figure (exhibit 9e).

Exhibit 9d: Get Revenue and Wholesale Expense Figures from the User



Note: A graphical user interface must also be created. It must enable the user to enter the relevant revenue and expense figures. Also, since no outside systems are involved in the use case, no plan is necessary for the case of unavailability.

Exhibit 9e: Get Revenue and Manufacturing Expense Figures from User



Note: A graphical user interface must also be created. It must enable the user to enter the relevant revenue and expense figures. Also, since no outside systems are involved in the use case, no plan is necessary for the case of unavailability.

The next task for step five is to consider including one or more collaboration diagrams. These diagrams are useful for showing the objects and links that are meaningful within an interaction between different entities and domains, or between various components. Since at this point in the problem solving process we do not have design details regarding parameters and local variables that will be used in the actual implementation, we will actually use a simplified version of the diagrams that just identify what type information is being passed. Our goal is to document the business logic involved in the process including any decisions, processes or computations. We will use the actual business terminology that is consistent with the contents of the domain dictionary. This will be the same level of detail that we used for the sequence diagrams. As with the sequence diagrams, for a real life problem domain, it is necessary to create a collaboration diagram for each possible feature combination.

Exhibits 10a, 10b, 10c, and 10d cover four different scenarios that occur for the example net income domain. These four examples show the potential variations that exist when none of the optional components (shown in Exhibit 7b) are present. While clients are not likely to switch from being manufacturers to wholesalers and vice versa, availability of the general ledger system may come and go over time in a distributed system. Exhibit 10a shows what happens when the client is a wholesaler and the general ledger is able to supply revenue and expense information. Exhibit 10b portrays what happens when the client is again a wholesaler but this time the general ledger is unavailable. In the case of exhibit

10c, the client is a manufacturer and the general ledger is available. Finally in 10d, the client is a manufacturer and the general ledger is once more unavailable.

Exhibit 10a: Collaboration Diagram 1 for the Net Income Domain

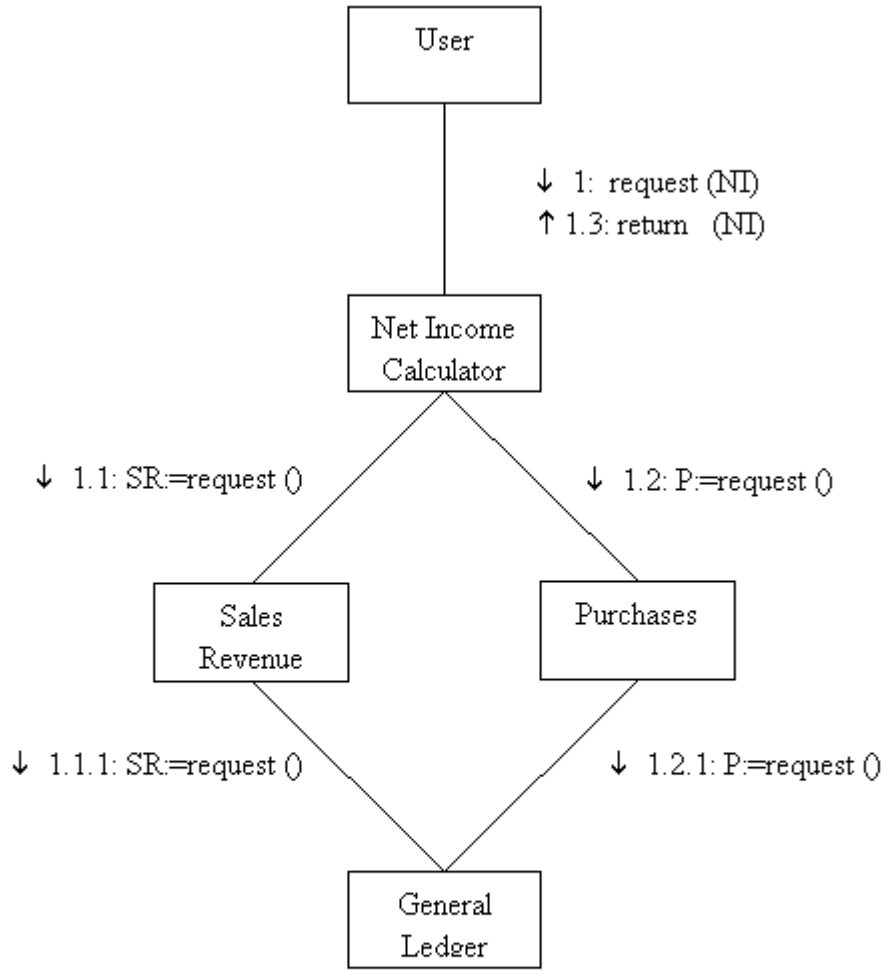


Exhibit 10b: Collaboration Diagram 2 for the Net Income Domain

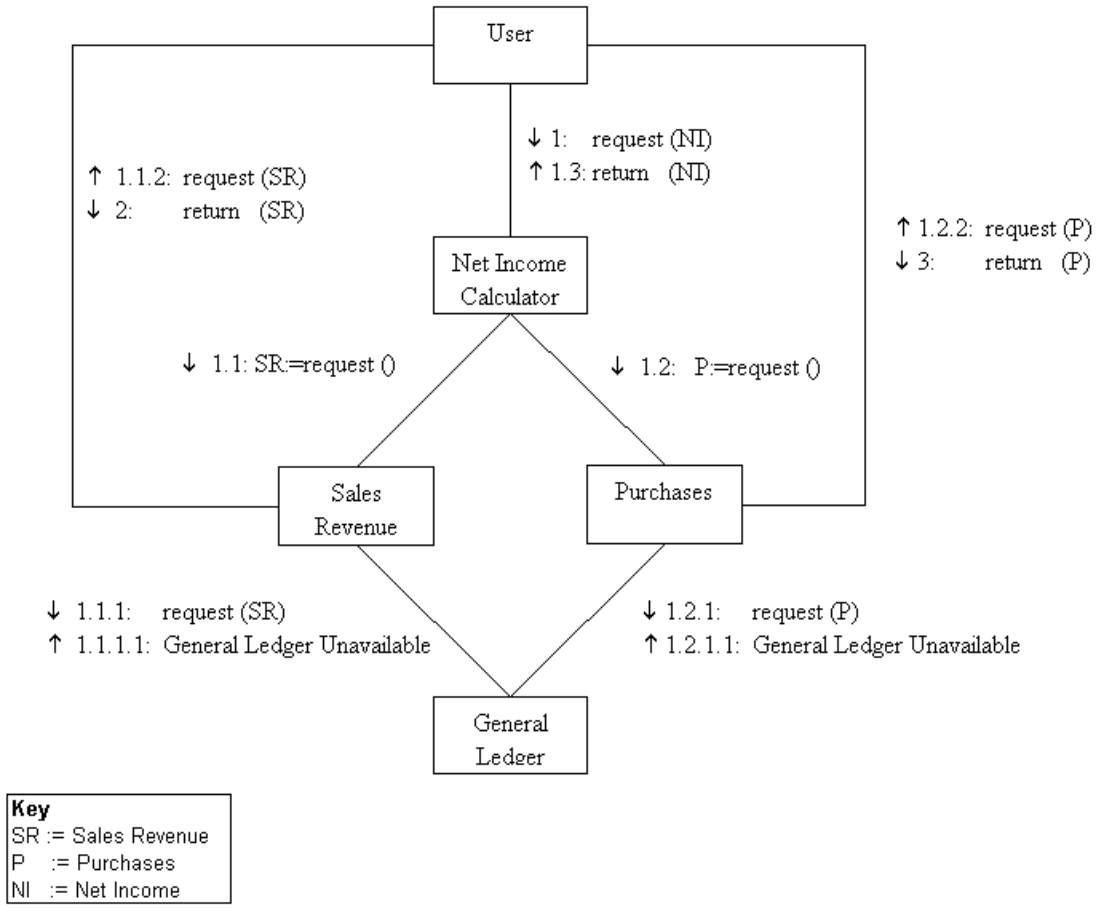


Exhibit 10c: Collaboration Diagram 3 for the Net Income Domain

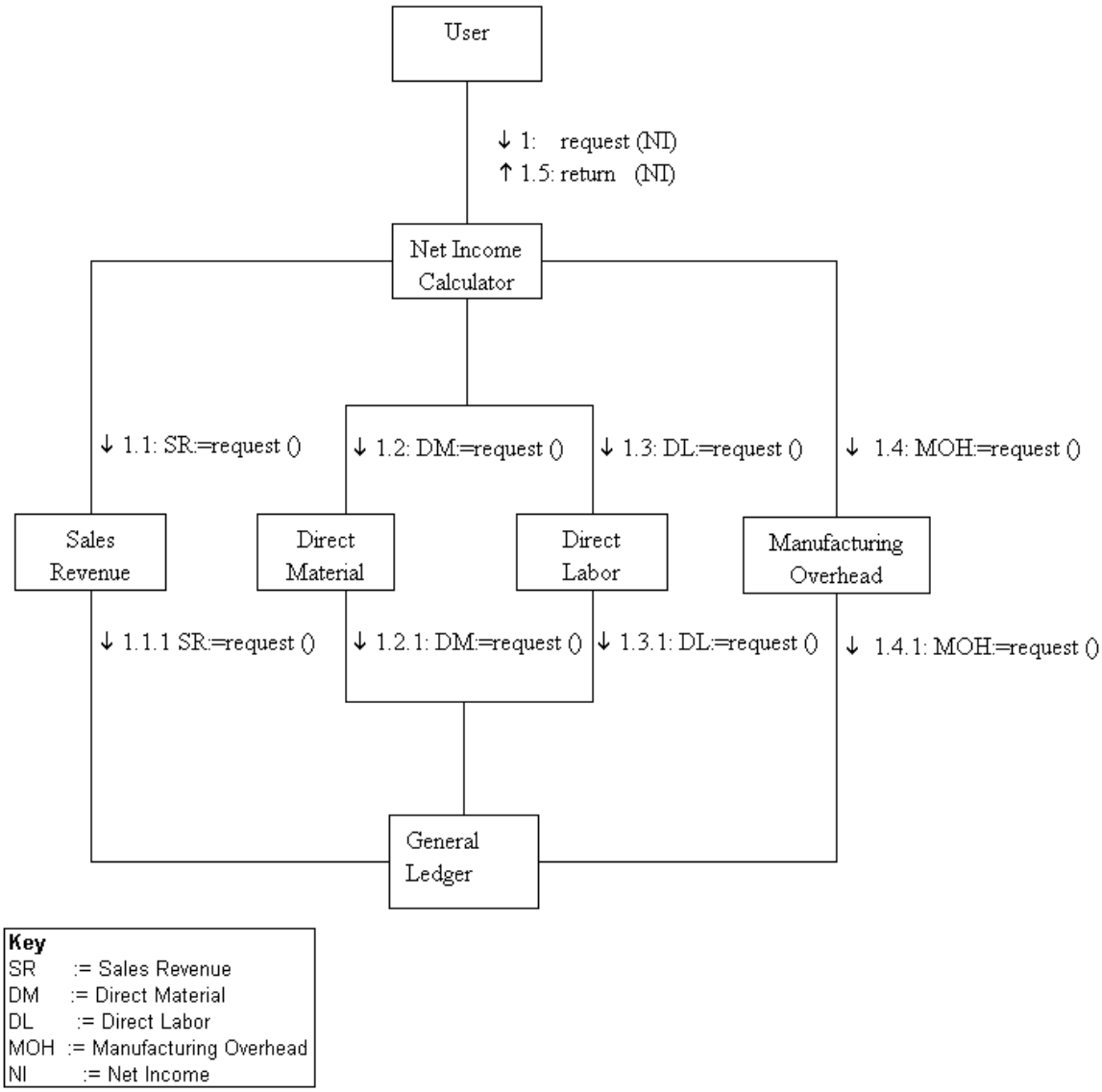
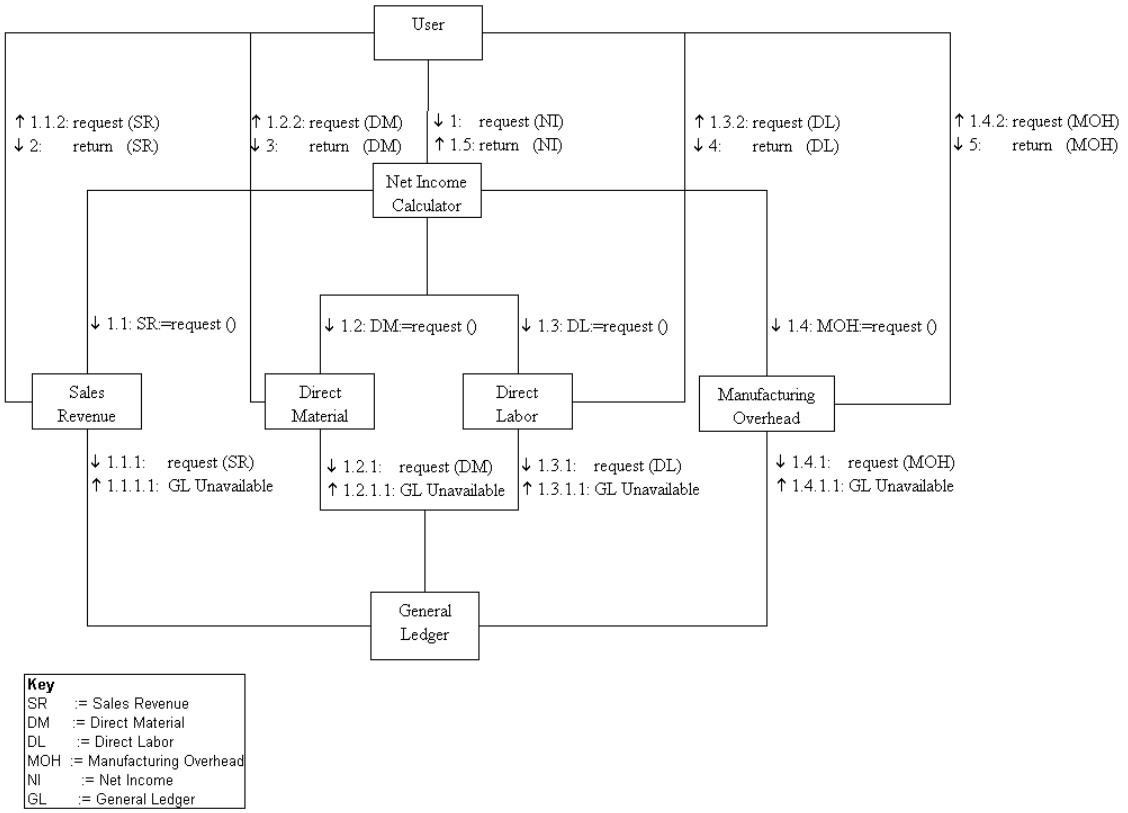


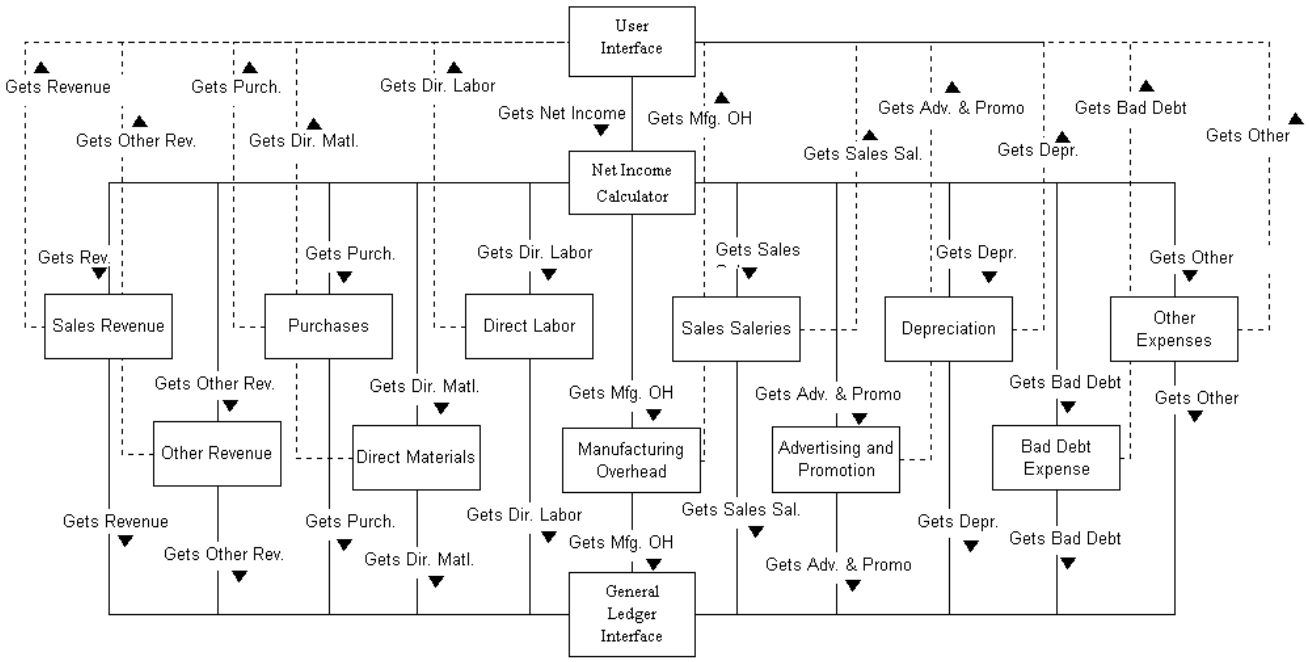
Exhibit 10d: Collaboration Diagram 4 for the Net Income Domain



The final task in step five is to create a domain model that shows the structural relationships between different entities and objects within the domain. These objects may consist of all features and other potential components that are known at this time. This model will show how the various objects of the system fit together to make up the overall architecture. Communications between objects are represented using association lines that are labeled with meaningful titles illustrating the relationships between the two objects. In general association lines will be solid, but we have represented them with both solid and dashed lines to make the diagram easier to read. This model helps to show the composition of the

overall domain and should be used to illustrate relevant dictionary terms. The objects in this diagram become candidates for components in the design phase. Only objects inside the system should be shown in this model. Any interactions with outside entities can be included by using interface components. Exhibit 11 shows the domain model for the net income domain. The dashed lines show communications between the user interface and various feature components. Any association lines that appear to go through a feature component are actually intended to go behind it. For example, there is no association between Other Revenue and Sales Revenue.

Exhibit 11: Domain Diagram for the Net Income Domain



Step VI: Validate Models, Dictionary, and Domain Descriptions

The goal of step six is to validate the models developed in step five, the domain dictionary, and the updated domain description with the relevant stakeholders. As a part of this validation process, look for potential variants in the models. This may help identify applications that should fall into the problem domain that have not yet been considered. This idea of looking for variants was proposed by Michael Jackson in his book about problem frames [13]. As he suggests, we will look for four different types of potential variants.

The first potential variant is a description variant. This type of variant usually delays the time when a decision is bound. For example, instead of deciding whether a client is a manufacturer or a wholesaler before building the system, perhaps we would want to let the user decide which one he is every time he asks for a net income calculation. The system could then go out to some sort of a description file to find out what types of revenues and expenses are associated with whichever type of client the user picked. Since the type of organization in the example being considered does not typically change day to day, this variant will not be used for the net income example.

The second potential variant is an operator variant. Operator variants specify the circumstances under which the behavior of a system must change. This may include changing default behaviors or assigning rules for overriding default behaviors. In the net income example, the system first looks to a general ledger system for revenue and expense figures. If the general ledger is not available, the user is prompted to input these values. This behavior is specified in

case the client has some sort of a general ledger printout available, but the system is unable to connect to the general ledger itself (perhaps a portion of the network is down). This is an example of an operator variant. For this example, the sequence diagrams already contain provisions for this scenario.

The third type of variant is the connection variant. This type of variant deals with examining the reliability of connections. Since many systems are distributed in nature, it is necessary to specify how the system will behave if suddenly some portion of its resources become unavailable. This could be applicable to communications from the domain to outside entities or even between components of the domain itself if they do not physically reside on the same machine. The same example that was used for the operator variant in the net income example would also apply here since the required behavior change is based on connection availability.

The fourth and final potential variant is the control variant. This is relevant when control is shared by two entities, domains, or components. When this occurs it is important to consider which object should have control at various times, under various circumstances.

All of the models prepared in step five should be updated to reflect new information learned during this variant analysis. Additional diagrams may be required or adjustments to existing diagrams may suffice. Once the domain has been expanded to include any potential variants that were identified and all of the major stakeholders have agreed on the revised contents of the models, domain definition and domain scope, it is time to move onto step seven.

Step VII: Create Decision Model

The only task for step seven is to create a decision model to be used in new development efforts within the domain. This decision model should describe the process that an application engineer would need to follow to specify the requirements of a new family member using the components and variabilities that have been identified in the existing models. It is important to include every decision that must be made in the model. This allows an application engineer to be able to specify parameters for a new application without first becoming an expert in the domain. After following the decision model, the application engineer should have a list of what features/components will need to be included in the application created for his particular problem. This information will be used to specify the problems parameters using an application domain specific language. The ADSL is described in step seven.

The decision model can be created easily from the feature diagram. Since all mandatory features must be included in any final application, they do not need to be included in this decision model. Any optional features may or may not be present in each application, so they should be included in the decision model. If there is some logical order in which the decisions should be made for a particular domain, then it is important to lay them out in that order for the decision model.

There are two different styles that may be used to create the decision model. The first style is to use an activity diagram from UML. This type of diagram is useful for showing what types of feature decisions may be made concurrently as well as explicitly stating the order that any decisions/processes

must follow. The second style is to use basic flow-charting notation to mark starting and ending points, decisions, and processes. This is useful to emphasize the feature decisions that must be made and to present them in a straightforward manner. One or both of these types of decision models may be used for a problem domain.

Exhibit 12a contains an activity diagram style decision diagram, while exhibit 12b contains a decision diagram in the flow-chart style for the net income domain.

Exhibit 12a: Activity Diagram Style Decision Model for the Net Income Domain

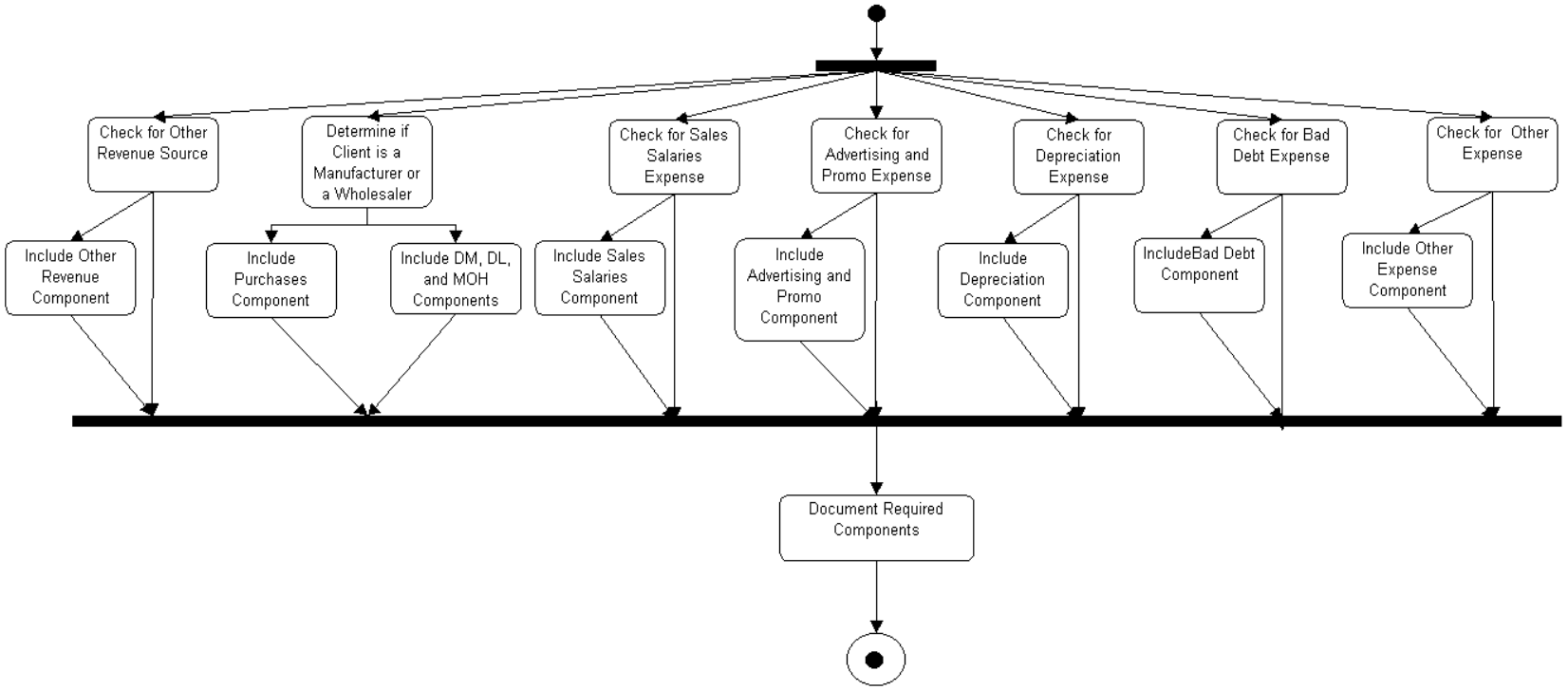
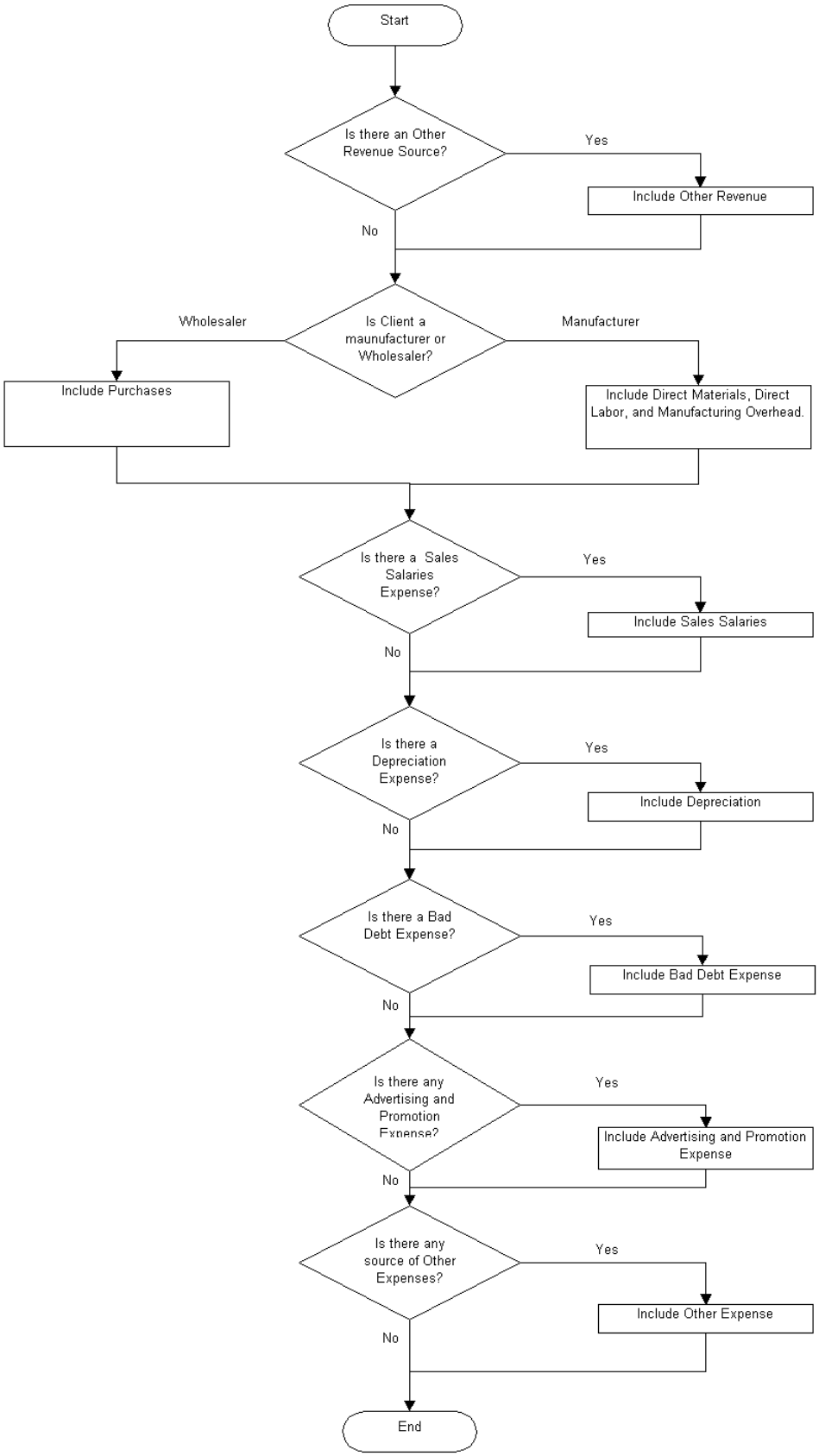


Exhibit 12b: Flow-Chart Style Decision Model for the Net Income Domain



Step VIII: Create ADSL

The goal of this step is to create an Application Domain Specific Language. The ADSL allows us to express the parameters of a specific application in a consistent way for a particular domain. This is useful because, if the language created is consistent, then it is possible to send a text description of a new application to a generator and have that application be automatically produced. This is the goal of generative programming. The difficult part is to find a language with sufficient expressive power to specify a new application that does not take too long to develop. For the purposes of this report's proposed method, it is sufficient to express the contents of the feature diagram in an ADSL.

Another advantage to using an ADSL is that once base level (i.e. ones with all optional features being present) collaboration and sequence diagrams have been developed, it may be possible to write programs that will generate additional diagrams representing the different potential feature combinations. This possibility is explored further in the future research directions section of this report.

We will use a slightly altered version of the ADSL method proposed by Deursen and Klint [14]. Each feature is named and followed by ":" and a feature expression. According to Deursen and Klint the feature expression can consist of:

- an atomic feature,
- a composite feature: a feature whose definition appears elsewhere,
- an optional feature: a feature expression followed by "?",
- mandatory features: a list of features enclosed in *all* (),

- alternative features: a list of feature expressions enclosed in *one-of* (),
- non-exclusive features: a list of feature expressions enclosed in *more-of* (),
- a default feature value: *default* = followed by an atomic feature,
- and remaining features of the form…, indicating that a given set is not completely specified.

The ADSL proposed by Deursen and Klint requires two additions to enable it to represent the extended feature diagrams that this report uses. The first addition deals with how to represent a non-essential feature. This shall be designated by a * preceding the feature name. To illustrate this notation, we will assume for a moment that Bad Debt Expense is a non-essential system component. The other extension is to represent communication between two components. If the communication is one way, we will say: talks-to (component A, component B). This represents a one-way communication from component A to component B. On the other hand if the communication is in both directions we say: talks-with (component A, component B). The components included may be either optional or mandatory components. If a talks-to() or talks-with() statement is included with an optional component for the specification of a particular application that does not contain that optional component, the talks-to() or talks-with() statement should simply be ignored. For the sake of illustration we will assume that Other Revenue must send a message to Other Expenses and that Advertising and Promotion must be able to send and receive messages from Sales Revenue.

Another revision that we shall make to the Deursen and Klint ADSL method involves changing the definitions of composite features and mandatory features. Although Deursen and Klint say that mandatory features are enclosed in the *all*() notation, they actually use *all*() with both mandatory and optional features inside the parenthesis in the example presented in their paper. We will eliminate this inconsistency by clarifying that the *all*() notation will actually be used to define a composite feature. Further, each atomic feature will be designated to be a mandatory feature unless it is explicitly noted to be optional.

We also have changed the definitions of the *more_of()* and *one-of()* expressions to only allow choices between a group of optional features, not a group of unspecified feature expressions. This is necessary to disallow certain illegal feature combinations. The first example of this would be the use of the *one-of*() with a list of mandatory features. If the list included one mandatory feature then the *one-of*() expression is unnecessary and if it contained more than one mandatory expression it would be logically incorrect. It is also incorrect to use the *more-of*() expression with a list of mandatory features. More than one mandatory features can be represented without the use of this expression

As a final revision, we will remove the ability to use the form … to define a feature expression without specifying its complete contents. Since this language is intended for use with a generator, it is better to explicitly specify feature expression contents. The revised informal definition of a feature expression is presented in Exhibit 12.

Exhibit 12: Revised Feature expressions

A feature Expression may consist of:

- an atomic feature which is defined by the domain dictionary
- a composite feature: a feature expression that is defined by a list of feature expressions enclosed in *all* ()
- an optional feature: a feature expression followed by "?"
- mandatory feature: an atomic feature or a feature expression followed by an "!"
- alternative features: a list of optional features enclosed in *one-of* ()
- non-exclusive features: a list of optional features enclosed in *more-of* ()
- a default feature value: *default* = followed by an atomic feature
- a non-essential feature: a feature expression preceded by "*"
- one-way communication: two or more mandatory or optional features enclosed in *talks-to* (). The first feature sends messages to each of the remaining features.
- two-way communication: a list of mandatory or optional features enclosed in *talks-with* (). All of the features listed may communicate with each other.

The ADSL is formally defined in Exhibit 13 using Backus-Naur Normal Form notation. The FEATURE mentioned comes from the domain dictionary.

Exhibit 13: ADSL in Backus-Naur Form

<constraint-exp> ::= <atomic-feature> : <feature-exp>

<feature-exp> ::= <composite-feature> | <optional-feature> | <default-feature> |
<mandatory-feature> | <alternative-feature> | <non-exclusive-feature> |
<non-essential-feature> | <one-comm> | <two-comm>

<atomic-feature> ::= FEATURE

<composite-feature> ::= all(<feature-list>)

<optional-feature> ::= < feature-exp>?

<mandatory-feature> ::= <atomic-feature> | < feature-exp>!

<alternative-feature> ::= one-of(<optional-feature-list>)

<non-exclusive-feature> ::= more-of(<optional-feature-list>)

<default-feature> ::= default = <atomic-feature>

<non-essential-feature> ::= * < feature-exp>

<one-comm> ::= talks-to(<feature-list>)

<two-comm> ::= talks-with(<feature-list>)

<feature-list> ::= < mandatory-feature-list> | <optional-feature-list> |
< mandatory-feature-list>, <optional-feature-list> |
<optional-feature-list>, < mandatory-feature-list>

<mandatory-feature-list> ::= <mandatory-feature> | <mandatory-feature>, < mandatory-feature-list>

<optional-feature-list>::= <optional-feature> | <optional-feature>, <optional-feature-list>

Exhibit 14 shows the impact of the changes on the ADSL for the net income domain. From this exhibit, it is possible to recreate the feature diagram for the net income domain. For example we know that it would begin with the net income feature that is made up of two mandatory features: revenue, and expense. Applying each expression one after another, the entire diagram can be recreated.

Exhibit 14: ADSL for the Net Income Domain

Net Income : all (Revenue, Expenses)

Revenue : all (Sales Revenue, Other Revenue?)

Expenses : all (Cost of Goods Sold, Selling and Administrative Costs, Other Expenses?)

Cost of Goods Sold : one-of (Cost of Goods Wholesaled?, Cost of Goods Manufactured?)

Cost of Goods Wholesaled : Purchases

Cost of Goods Manufactured : all (Direct Materials, Direct Labor, Manufacturing Overhead)

Selling and Administrative Costs : more-of (Sales Salaries?, Advertising and Promotion?, Depreciation?, * Bad Debt Expense?)

talks-to (Other Revenue?, Other Expenses?)

talks-with (Selling and Administrative Costs, Sales Revenue)

Once the ADSL has been created, the model of the overall problem domain is complete. Using the ADSL and the diagrams and UML models that have been created, an application engineer should have all of the tools necessary to generate a design model for the solution to a specific problem within the domain.

Step IX: Validate Decision Model and ADSL

The task for step nine is to meet with domain experts and have them validate the decision model and the ADSL created in steps seven and eight. This

should be done in a session with several domain experts, representatives from the domain modeling team, and two application engineers aren't very knowledgeable of the domain area. The experts propose at least two realistic problems that would fall within the domain. The application engineers then attempt to follow the decision model to define features and express the application in the ADSL. The domain experts oversee this process as it is being carried out. This session should identify usability problems. Feedback from both the application engineers and the domain experts should be used to improve the model and/or the ADSL. Any necessary revisions should be made before moving on to step ten.

Step X: Get Final Signoff

This is the final step in the process. The one action item is to get the final signoff from all of the major stakeholder groups. This will most likely involve a formal presentation. The project team should demonstrate that they have successfully achieved all of the project goals. They should give an overview of important information learned about the domain. They should demonstrate that UML diagrams prepared adequately capture the systems behavior and that the decision diagram and ADSL can be used to adequately represent application variabilities. They should also present important metrics such as whether the project was completed in the time allotted to it and whether it stayed under its allocated budget.

The actual sign-off procedure will be done in a manner proposed by Ian Graham in his book on Object-Oriented Methods [15]. A signoff sheet will be circulated to each stakeholder previously identified. Each stakeholder may sign

off as being in agreement that the problem analysis phase has been completed in a way that the project is now able to move on to the solution stage. If they are not in agreement, they must sign that they are in strong disagreement with what has been achieved so far. Any signatures stating strong disagreement will prevent the project from moving forward. An open issues document should also accompany this signoff sheet. Anyone who has a strong divergent opinion over a specific aspect of the project can document his or her point of contention on this open issues sheet and still sign that they are in agreement with the project progress overall. Any open issues recorded must have a specific person assigned to resolve them and a deadline. This will allow everyone to go away from the meeting feeling that the efforts so far have been a success.

Summary

The summary form of the complete process is as follows:

1) Document an initial description of the domain. This description should include the following elements:
   a) A problem statement.
   b) A general description of capabilities of the proposed application family
   c) Examples of any existing applications which would fall into this domain

2) Identify potential stakeholders and domain experts (individuals and/or groups). Consider the following groups of people:
   a) Senior Management
   b) Project Management
   c) End Users
   d) Customers
   e) Other service recipients
   f) Investors
   g) Developers
   h) Regulators

3) Conduct a series of moderated meetings with representatives from each group of stakeholders and domain experts. The following tasks should be accomplished:
    a) Define the overall project objectives.
    b) Expand the domain description. Include any performance constraints that the new system family must satisfy.
    c) Define the boundaries of the Domain. Make a note of any known entities/applications that may interact/communicate with instances of this new domain.
    d) Identify potential sources of information about the domain. Consider the following:
        i) Information regarding applications which have previously been developed, that fall into the domain (i.e. requirements analysis, existing code, etc.)
        ii) Developers that may have worked on related applications.
        iii) Other domain experts not previously identified.
        iv) Users of existing applications.
        v) People/Documents that have information about entities/other domains with which applications from this domain must communicate.
        vi) Other existing sources of information including textbooks, standards, etc.

4) Gather/Document potentially relevant domain information.
    a) Interview domain experts.
    b) Explore other potential information sources.
    c) Create a dictionary of common domain terms.
    d) Identify features of applications within the domain. Consider the following:
        i) Commonalities across all applications
        ii) Variabilities between applications
        iii) Note any composition rules.
    e) Document any required APIs, which must be used to communicate with neighboring domains/entities. Document what type of communication takes place.

5) Model the domain. Any additional information learned regarding domain terms should be used to update the dictionary.
    a) Use extended feature models to document commonalities and variabilities of function for applications falling within the domain. If there is a need for leaf node features to communicate with each other, include this in the diagram.
    b) Create a series of use case diagrams that show how the user as well as any outside entities or domains will interact with applications falling within the new domain.
    c) Create one or more sequence diagrams to show how this new domain would communicate with other domains/entities. Include with each

diagram a text description of what should take place if/when any of these outside entities become unavailable. If the method by which these two entities will communicate is unknown, document this as a part of the diagram.

d) In addition to sequence diagrams, consider creating one or more collaboration diagrams.
e) Create a domain model that shows the structural relationships between different entities and components within the domain.

6) Validate models, dictionary, and the updated domain description with users and domain experts. As a part of this validation process, look for potential variants in the models. This may help identify applications that should fall into the problem domain that have not yet been considered.
   a) Make any revisions necessary.
   b) Make sure all parties are in agreement as to the scope of the domain.

7) Define a decision model to be used in new development efforts within the domain. This decision model follows the process that an application engineer would need to follow to specify the requirements of a new family member.

8) Create ADSL (Application Domain Specific Language). This language will be used by application engineers to specify new product instances.

9) Validate decision model and ADSL with domain experts.

10) Get final signoff from all major stakeholder groups.

Exhibit 15 contains a summarized list of the major artifacts created by this process. The step in which they are first created and later validated are also noted.

Exhibit 15: Artifacts Created

Artifact Description	Step Created	Step Validated
Domain Description	III	VI
Dictionary	IV	VI
Feature Model	V	VI
Use Case Diagrams	V	VI
Sequence Diagrams	V	VI
Collaboration Diagrams	V	VI
Domain Model	V	VI
Decision Model	VII	IX
ADSL	VIII	IX

Future Research Directions

The first area of future research deals with the domain dictionary. So far we have assumed that a document of relevant terms is kept. We have not specified how this is to be kept or how to use this document to describe aspects of the domain outside of the feature diagram. An interesting area of research would be how to use terms within the dictionary to compose sentences that might describe the UML models created in this process including the use case diagrams, sequence diagrams, collaboration diagrams, and the domain model.

The second area of potential future research was already briefly mentioned in the section of this report that described step VIII (the creation of the ADSL). As was briefly mentioned in that section, it may be possible to write programs that will generate additional diagrams representing the different potential feature combinations once base level diagrams have been created. The next two paragraphs describe two possible approaches to this problem.

One possibility is to develop all use case diagrams, the domain model, and a series of base level sequence and collaboration diagrams corresponding to the different use case scenarios, then write a program that creates the additional sequence and collaboration diagrams necessary to illustrate the different possible combinations of optional features. This series of base level diagrams would contain all of the possible optional components somehow annotated to show that they are variable.

The other possibility is to look into representing the entire domain as a set of four diagrams (use case, sequence, collaboration and domain). Each diagram

would contain all of the possible optional components. These components would need to be marked as variable perhaps using some notation from the ADSL. The goal would be to then use a UML interpreter to generate whatever specific models are needed for an individual system specified by an application engineer.

A good beginning reference for how to model variability in UML diagrams is Matthias Clauß's paper [17]. In this paper, Clauß presents extension notations to represent the location of the variability, the variants, and the relationship that assigns each variant to its variation point. These notations involve the use of stereotypes. Clauß states that this notation can be applied towards classes, components, packages, collaborations, and associations. His paper presents an example class diagram that uses his proposed notation.

The other logical area of future research is to move on to the design phase of domain engineering. The same type of process needs to be developed for this phase. It should combine aspects of current processes with UML diagrams and give consideration to concerns associated with distributed programming, as we have done for the problem phase in this report.

Conclusion

This report presents a ten-step detailed process that can be followed to effectively understand, document, and model the problem space of a new domain. This process combines important aspects of existing domain engineering processes into one method. It also adds several elements that current processes lack including: the use of updated modeling techniques (the use of UML) and giving early consideration to concerns arising from the distributed heterogeneous programming environment that is becoming more and more common today. For example, the definition of the feature model, which was first proposed by the FODA method of domain engineering, has been extended to take communication and availability concerns between components into account. The process also specifies a grammar that can be used to describe the contents of an extended feature model. The proposed method is explained step by step and is illustrated through the use of an example problem.

The end result of this process is a model of the overall problem domain. This model is comprised of the ADSL, its mathematical composition rules, and the various diagrams and UML models that have been created. Design engineers can use these artifacts to create a common architecture for the domain during the domain design phase of domain engineering. During domain implementation, this system architecture will actually be built along with a generator that will accept individual system descriptions in terms of the ADSL. The model of the problem domain will once again be used during application engineering when the

application engineer will use it to specify necessary parameters to generate a design model for the solution to a specific problem within the domain.

List of References

1. Czarnecki, K., and Eisenecker, U. “Components and Generative Programming” Invited talk, in Proceedings of the Joint European Software Engineering Conference and ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/FSE'99, Toulouse, France, September 1999).

2. Czarnecki, K., Eisenecker, U., “Generative Programming Methods, Tools and Applications”, Addison-Wesley, 2000.

3. Kang, K., et al. “Feature-Oriented Domain Analysis (FODA) Feasibility Study” (CMU/SEI-90-TR-21, ADA 235785). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1990.

4. http://www.sei.cmu.edu/str/descriptions/foda_body.html

5. Simos, Mark A., “Organization Domain Modeling: A Tailorable, Extensible Framework for Domain Engineering”. Proceedings of the 4th International Conference on Software Reuse (ICSR ’96), pp. 230 - 232.

6. Simos, M., “Organization Domain Modeling (ODM): Formalizing the Core Domain Modeling Life Cycle”. SIGSOFT Software Engineering Notes, Special Issue on the 1995 Sympsium on Software Reusability, Aug 1995, pp. 196 - 205.

7. Weiss, David M., “Software Synthesis: The FAST Process”, MultiUse Express, June 1994.

8. Ardis, M.,Daley, N., Hoffman, D., Siy, H., Weiss, D., “Software Product Lines: a Case Study”. Software Practice and Experience 30(7), June 2000 pp. 825-847.

9. Widen, Tanya, “Formal Language Design in the Context of Domain Engineering”. Master’s Thesis, Oregon Graduate Institute, June 1998.

10. Gacek, Cristina, “Exploiting Domain Architectures in Software Reuse” Proceedings of the ACM-SIGSOFT Symposium on Software Reusability (SSR'95), ACM Press, Seattle, WA, 28-30 April 1995, pp. 229-232.

11. J. Neighbors, “Draco: a method for Software Systems”, “Software Reusability”, Biggerstaff & Perlis, ACM Press 1989.

12. Rumbaugh, J., Jacobson, I., Booch, G. “The Unified Modeling Language Reference Manual”, Addison-Wesley, 1999.

13. Jackson, Michael. “Problem Frames: Analyzing and structuring software development problems”, Addison-Wesley, 2001.

14. Van Deursen, Arie, and Klint, Paul, “Domain-Specific Language Design Requires Feature Descriptions”, Journal of Computing and Information Technology, 2002.

15. Graham, Ian. “Object-Oriented Methods: Principles & Practice”, Addison-Wesley, 2001.

16. http://www.omg.org/

17. Clauß, Matthias, “Generic Modeling using UML extensions for variability”, OOPSLA 2001: Workshop on Domain Specific Visual Languages. http://www.isis.vanderbilt.edu/oopsla2k1/Papers/Clauss.pdf