On The Impact of Domain Dynamics on Product-Line Development

Haitham S. Hamza
Department of Computer Science and Engineering,
University of Nebraska
Lincoln, NE 68588-0115
hhamza@cse.unl.edu

Abstract: Software Product-line engineering aims at enabling systematic software reuse by allowing several related software applications to be developed using common assets. A key activity in product-line engineering is to identify common concepts among different products and their variation space, and to exploit these concepts to develop reusable assets. These concepts are also used to design a domain-specific language and a set of associated tools to facilitate the development of the different products in the family. However, when domains evolve, in response to paradigm shifts and technological advances, concepts may need to be modified or removed, or even new concepts might be added to the system. Such changes will not only affect the developed domain language but also the associated tool set. Adapting domain languages and associated tools, if possible, may become expensive and time consuming. In this paper, we first investigate the impact of domain dynamics on product-line development, and then we outline our going research for developing an approach to reduce such an impact.

1. Introduction

Software community has long realized the need for systematic software production techniques to overcome software complexity and to increase productivity [4]. However, unlike many other engineering disciplines, software development is inherently a *people-oriented* activity that is hard to be fully automated for mass production [6]. One promising approach to enable systematic software development is *Product-line engineering* techniques (also known as product-family engineering).

Product-line engineering aims at enabling systematic software production by providing the required assets and tools to evolve related software applications from a base system. This base system contains generic structures of features common to all products in the family. The concept of developing a *family* of products by exploiting their commonalities have been under investigation since early 70's [13] [14]. However, the increasing complexity of software systems coupled with time-to-market constraints, have renewed much interest in investigating the different aspects of this topic over the last few years. Thus, several methods and techniques for engineering software product-line have been developed, e.g. [1] [10] [22] [16].

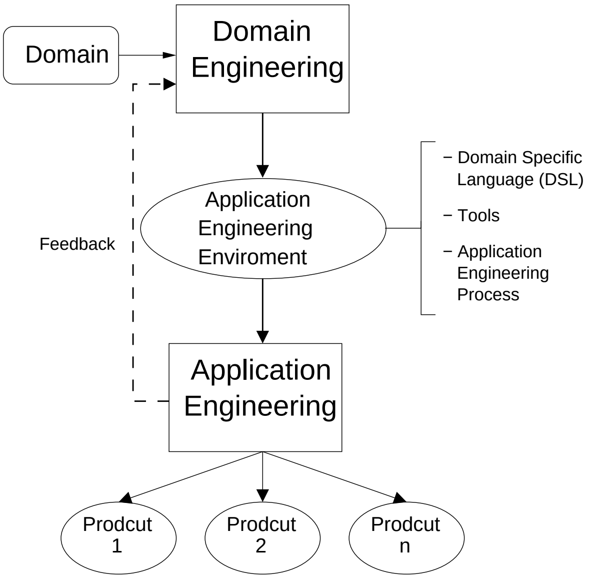
Domain
Domain Engineering
Feedback
Application Engineering Enviroment
- Domain Specific Language (DSL)
- Tools
- Application Engineering Process
Application Engineering
Prodcut 1
Prodcut 2
Prodcut n

Figure 1: A generic structure of product-line engineering development [1].

A key common step in most product-line engineering techniques is to identify common concepts among all products in the family, and to define the variation space of these concepts. These commonalities and variabilities play a central role in identifying the domain scope of the product family, and in evolving the base system of the family. In addition, commonalities and variabilities are important for the design and the development of domain-specific languages and tools needed for engineering the different products in the family.

However, when domains evolve, due to paradigm shifts and technological advances, the identified commonalities and variabilities may change, as concepts may need to be modified or replaced, or even new concepts may need be added to the system. Such changes, therefore, need to be reflect into all developed languages and tools. The ripple effect of domain evolution may greatly undermine the principle of adapting product-line techniques as maintaining domain languages and tools may overshadow the benefits gained from the systematic reuse.

In this paper, we first investigate the impact of domain dynamics on product-line development activities. Then, we illustrate our current research on developing an approach to reduce such an impact using the notion of software stability model [5].

The reminder of the paper is organized as following. Section 2 reviews the basic concepts of product-line engineering. Domain dynamics and their impact on product-line engineering activities are discussed in Section 3. Sections 4 and 5 present the proposed approach. The paper concludes in Section 6.

2. Product-Line Engineering

A typical product-line engineering process consists of two main activities (See Figure 1) [1]: *Domain Engineering* and *Application Engineering*:

- ***Domain Engineering.*** Domain engineering focuses on developing reusable assets required to evolve the different products in the family. A key activity in domain engineering is *domain analysis*. Domain Analysis (or DA, for short) [12] is the activity of capturing the common features among the different products in product family, and parameterizing the variation space of these features. In addition, DA methods provides guidelines for exploiting identified commonalities and variabilities into reusable assets for future reuse. Several domain analysis methods have been proposed in the literature, e.g. [3] [9] [10] [15] [17] [19] [20] [21] [2] [11]. The main product of the domain engineering activities is the development of application engineering environments (See Figure 1). The application engineering environment consists of the necessarily domain-specific models and languages, and the required tools to support the engineering of the different products in the family.

- ***Application Engineering.*** Application engineering activities focus on the process of developing the different products in the family. These activities include the generation of the product specification and the selection, customization, and integration of the appropriate assets to implement a specific product in the family.

3. The Problem

In this section, we explain the concept of domain dynamics, and then investigate the impact of the evolution of the domain on the activities of product-line engineering.

3.1 Domain Dynamics

Domains can evolve overtime to reflect paradigm shifts or technological advances. This fact, however, is neglected in most existing product-line engineering methods. Most existing DA techniques, for example, identify a set communalities and assume that these communalities are *"stable"*, i.e. they are enduring in the domain. This assumption, however, may not always hold. In fact, several features that appear to be common to the products of a given domain may become *"unstable"* over time, and sometimes they even disappear as new technologies emerge. As a result, domain models developed based on these "unstable" commonalities become unstable too, and hence, they may require significant modifications over time.

To illustrate the concept of stable and unsable communalities, consider the example of developing a family of telecommunication routers. In telecommunication networks, a source and a destination are connected through some intermediate routers. These routers are dynamically configured in order to establish the required connection. A router consists of several modules including: the *switch unit* and the *control unit*. Switch unit has input ports and output ports for receiving and sending signals, respectively. The control unit consists of control circuits and algorithms to configure the switch unit.

A possible domain model that captures the commonalities among electronic routers is given in Figure 2. The model can be used to develop different types of routers by adding/removing additional components to the model shown in Figure 2, and by adjusting the parameters of the different features and components in the model. For example, one can adjust the size and the type of the "Memory" to develop different models of the same type of routers. Also, one can change the type of the router by adding/removing the appropriate components. For example, we can develop an ATM (Asynchronous Transmission Mode) router (Figure 2), or replace the ATM switch with an IP (Internet Protocol) switch to develop an IP router.

However, when the domain evolves, the model shown in Figure 2 may require considerable changes. To illustrate this, consider for example the OEO (Optical-Electronic-Optical) converters unit shown in Figure 2. OEOs are indeed one of the main building blocks for any electronic router. However, with the continuing advances in optical technologies and devices, future routers are expected to operate in the optical domain, where no OEOs are needed. In such a case, the

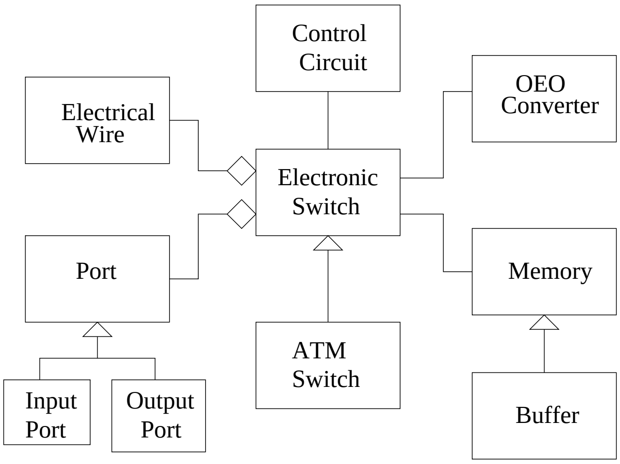

Figure 2: Portion of the domain model of an electronic router.

domain has not changed per se, we are still interested in developing telecommunication routers but with different type of technology. In complex architectures, accommodating such evolution may be very costly as several changes may be required.

3.2 Impact of Domain Dynamics on Product-Lines

Experiences reported with adapting product-line engineering techniques suggest that, productivity can be increased if domain specific languages and specialized tools are developed based on the concepts of the domain itself [1] [23]. This is because, generic modeling languages (e.g. the Unified Modeling Langauge) need to be customized and/or extended to reflect the concepts of the domain and to accommodate the parameterized variability space of products in the family; a task that, if possible, can become complex, error prone, and of course expensive.

However, when the domain evolves, a chain of changes may be required in order to adapt the developed domain language and its associated tool sets. These modifications, if possible, may become complex and time consuming, error prone, and of course expensive.

4 The Proposed Approach

In this section, we present the underlying principle of an approach that we develop in order to alleviate the impact of domain dynamics on product family development activities.

4.1 Approach Overview

In order to reduce the impact of domain dynamics on the different activities in product-line engineering processes, we suggest that:

> “domain analysis methods, and in particular the techniques used for identifying commonalities and variabilities, should isolate the common *“core concepts”* of the domain form concepts that are likely to change overtime, *even if these concepts appear to be “common” to all the products in the family*”.

For example, in the analysis of electronic routers, we may identify *“storing”* as a common concept among all routers. Whether developing electronic or optical routers, the concept of “storing” is still valid, although the realization of the concept can be quite different[1]. Thus, the new

---

[1] For example, in optical routers; the notion of Buffers cannot be explicitly realized because of the current lack of

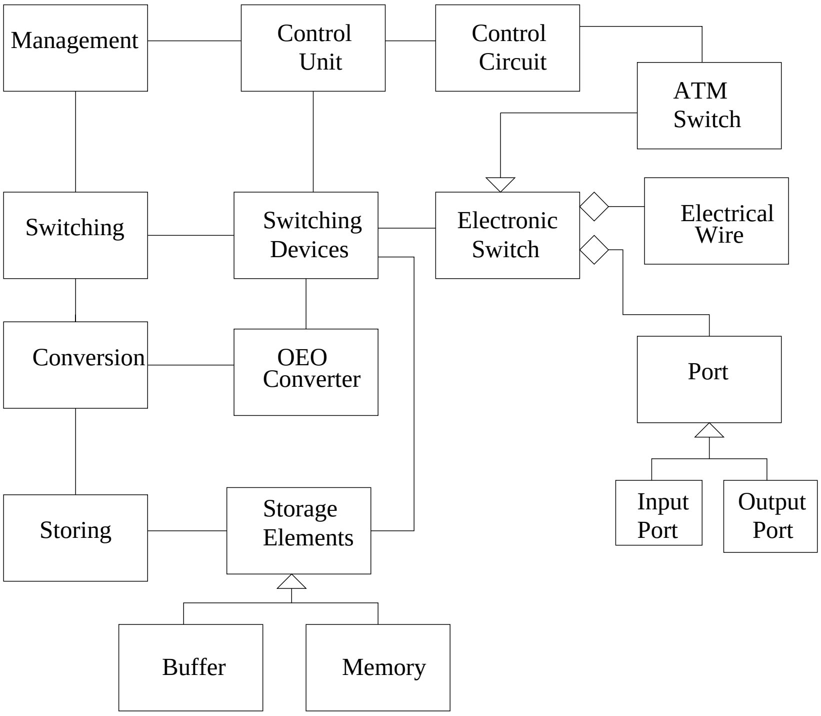

Figure 3: Portion of the domain model of an electrical router applying the new vision.

commonality model contains “Storing” as a component. Should the type of the router change, or new storage technologies evolve, the domain must not be reanalyzed from scratch.

Figures 3 and 5 shows portions of the domain models of an electronic and an optical routers, respectively, using the new approach. It is worth noting that the two architectures have several elements in common. In particular, they share all the *core concepts* of the domain such as “switching”, “management”, “conversion”, and “storing”.

Now, consider a simple domain evolution scenario, where a new optical device that can *simultaneously* switch and convert optical signal has become available. An example of such a device is the Wavelength Exchanger Optical Crossbar (WOC) proposed in [8]. In such a case, all the elements related to the conversion concept can be removed from the model (See Figure 4). It is worth noting that such a change does not affect the other components in the model.

The above approach for developing domain models leads naturally to a new approach for constructing domain-specific languages and engineering tool sets for developing the products of the family. In particular, we suggest that domain-specific languages should *directly map the core concepts of the domain into language constructs.* These constructs provide appropriate extension points for binding the required features of a given product, which will be identified during the requirement specifications in the process of application engineering.

4.1 Separation of Concepts

When domains evolve, some common core concepts may need to be removed and/or new concepts may be added. In large and complex domains, modifying the domain model may lead to major

---

optical technology. Physically, signals are stored in the optical domain by being delayed through multiple fiber-delay loops (FDLs).

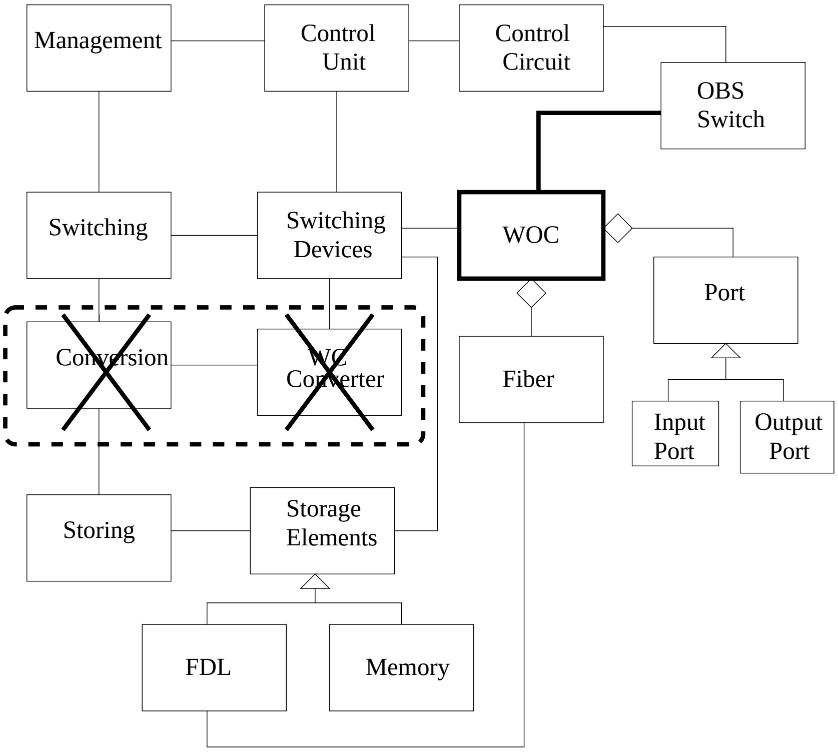

Figure 4: Portion of the domain model of an optical router under domain evolution.

changes due to the ripple effect of some of the changes. In the worst case, the overall model may collapse forcing a new domain engineering process. To avoid this, one may think of the domain model as a collection of *concept units* instead of a set of individual concepts. For example, in Figure 5, two units are highlighted (the dashed boxes), one unit encapsulates concepts related to the “switching” functionality, whereas the other unit consists of the concepts related to the “conversion” process. Adapting the domain model for future changes may be achieved by only adding and removing appropriate concept units. This further abstraction of the domain model can reduce the complexity of the model and hence minimize the impact of domain evolution on the model and all associated activities.

5 Realizing the Approach

To realize the above approach, two main questions need to be answered:

1. How to identify the core common concepts of a domain? and
2. How to group different concepts into units that can be added and removed with minimal impact on the rest of the model?

To answer the above two questions, we developed an approach that applies the concepts of *Software Stability Model* (SSM) [5] to identify the common and stable concepts of the domain. In addition, in order to separate and encapsulate concepts into different concept units, the approach uses the mathematical theory of *Formal Concept Analysis* (FCA) [18] along with a set of quantitative metrics, similar to those we proposed in [7].

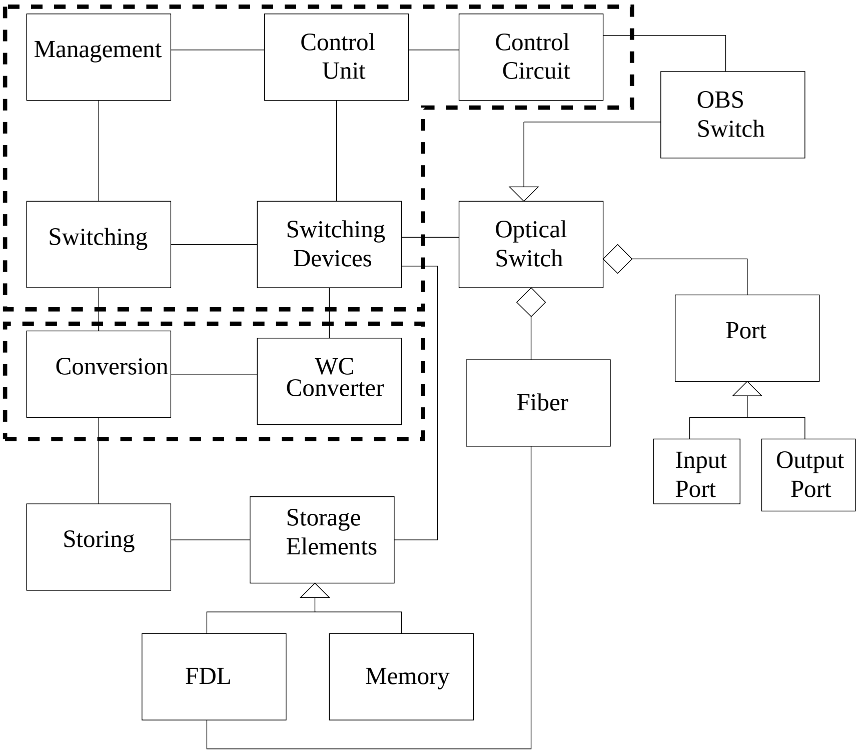

Figure 5: Portion of the domain model of an optical router.

5.1 A Brief Background

The proposed approach is based on two main notions: the *Formal Concept Analysis* (FCA) theory [18] and the *Software Stability Model* (SSM) [5]. FCA is a mathematical framework that can be used to represent and analyze data and their relationships [18]. A formal concept is defined as a pair (O, F), such that $\beta(O) = F$ and $\alpha(F) = O$, where G is a set of objects; F is a set of features; $O \subseteq G$; and $F \subseteq M$. $\beta(O)$ is an operator that is defined as the set of features shared by all the objects in O. Similarly, $\alpha(F)$ is defined as the set of objects that share all the features in F.

SSM is a generic layered approach for modeling software that classifies the classes of the system into three layers: *Enduring Business Themes* (EBTs): contains the enduring and core knowledge of the underlying business; *Business Objects* (BOs): contains classes that map the EBTs of the system into more concrete objects; and *Industrial Objects* (IOs): IOs are classes that map the BOs into concrete objects. In a banking system, for example, one possible EBT is “ownership”; without an “Ownership”, there is no account. “Account” is a BO, while a “SavingAccount” is a concrete “Account”, and hence it is an IO.

5.2 Main Activities

In the following, we provide a brief overview of the main activities in our approach:

- **Phase 1: Analysis Phase.** In this phase, first the common generic requirements of the different products are analyzed using existing analysis and requirements engineering techniques. This step produces a set of common Functional and Non-Functional Requirements. Next, the domain is analyzed using the concepts of SSM (See Section 5.1) in order to identify the EBTs, BOs, and IOs of the domain. In addition, a use-case model is developed and use-case scenarios are identified and specified. Finally, a set of core concepts in the domain are identified. Each core concept is matched with a set of use cases that realizes this core concept. For example, the *“Security”* concept can be matched with a set of use cases that includes the *VerifyPassword* and the *UserLog* use cases.

- **Phase 2: Formal Concept Analysis Phase.** In this phase, we first construct the formal context of the domain with G being the set of the EBTs; BOs, and IOs of the domain, and M being the set of all the use cases identified in Phase 1. In the formal context, an X is placed in the intersection between a element in G and a use case in M, if the element is a participant in this use case. Next, we generate the formal concepts of the system [2]. It should be noted that, not all the generated formal concepts are relevant concepts. For example, some formal concepts may violate the structure of the SSM, and hence, they must be eliminated. The elimination of irrelevant concepts is accomplished by identifying a set of *filtering rules*. A *rule* is a predefined constrain that should be satisfied by each concept for it to be relevant to the domain. One rule, for example, may emphasize that a relevant formal concept must contain one or more EBTs. Any concept that violates this rule must be eliminated.
- **Phase 3: Concept Encapsulation Phase.** This phase quantifies the relationships between the core concepts (Phase 1) and the computed formal concepts (Phase 2) of the domain. The result of this phase is a set of formal concepts that decompose the domain into stable units. This phase consists of two main steps. The first step decomposes the domain into a set of *“Clusters”*. Each cluster is a collection of formal concepts that realizes one of the core concepts in the system. A cluster “realizes” a given core concept, if and only if, the union set of the use cases of all the formal concepts in the cluster forms a *super set* for the set of the use cases of the given core concept.

  It may be noted that, for a given core concept there can be several clusters that realize this concept. Thus, the second step in this phase is to select the one cluster among all possible clusters to realize a given core concept. To achieve this, we quantify the relevance of each cluster in the domain with respect to a given core concept using the following four quantitative metrics:

  1. *Coverage Percentage:* measures the percentage by which a cluster covers a given core concept. This metric assigns a negative weight that is proportional to the relative size of the use case sets of the cluster and the core concept. Clusters with less redundant use cases can reduce the possibility of overlapping clusters in the final decomposition of the domain. The less this overlap is, the simpler the model becomes;
  2. *Coupling Index:* measures the average interaction between a given cluster and all the core concepts of the domain. The objective here is, again, to reduce the overlap possibility between the set of clusters if the final decomposition of the domain;
  3. *Stability Index:* measures the level of stability of a given cluster. That is, how enduring the cluster is in the domain under consideration;
  4. *Quality Factor:* computes the overall quality of a given cluster with respect to each core concept by assigning weights to each of the previous three metrics. The weight of each metric is adaptable and depends on several factors, such as the nature of the domain and its evolution rate.

---

[2] http://www.st.cs.uni-sb.de/∼ lindig/ (online)

Based on these metrics, we can identify the best cluster to realize each core concept in the domain. Effectively, we have decomposed the domain into a set of clusters, each can be manipulated separately.

One of the main challenges in the proposed approach is that, some of the core concepts may not be separable as they significantly cross-cut other concepts in the system. To solve this problem, we introduced the notion of *"Autonomous"* concepts and *"Distributed"* concepts[3]. The former are concepts that can be isolated and encapsulated into stand-alone units, whereas the later are concepts that cross-cut other concepts, and hence, they cannot form a stand-alone unit. Based on this classification, we developed two approaches to handel each of these two concept types when evolving the model of the domain.

6. Conclusions

In this paper, we define and discuss the impact of *domain dynamics* on the different activities of product-line engineering. We investigate how the evolution of the domain can adversely affect domain models and other components that are based on these models. To avoid this problem, we suggest an approach that identifies the *"core"* common concepts of the domain instead of just considering the common concepts among the products of the domain as in most existing product-line engineering methods. These core concepts are used to develop a core domain-specific language that can be easily adapted when the domain evolves. This work still in its preliminary stages and further investigation and validations are certainly needed. However, we believe that the proposed approach holds the promise to alleviate the impact of domain evolution on the different activities and products of the product-line engineering paradigm.

References

1. D.M. Weiss and C.T.R. Lai. Software product-line engineering: a family-based software development process. Addison Wesley, 1999.

2. G. Arango. Domain analysis methods. In Software Engineering, W. Schaefer and D. Diaz, Eds., New York: Ellis Horwood, 1994.

3. S.C. Bailin, et al.,'KAPTUR: knowledge acquisition for preservation of tradeoffs and underlying rationales, *Proc. 5th knowledge-based Software Assistant Conference,* 1990.

4. J.M. Buxton, P. Naur, and B. Randell, (editors) "Software engineering concepts and techniques *1969 NATO Conference of Software Engineering.* NATO Science Committee, Brussels, 1969.

5. M. E. Fayad, A. Altman, "Introduction to Software Stability", *Communications of the ACM,* vol. 44, no. 9, Sept. 2001.

6. J. Greenfield, and K. Short. "Software Factories: Assembling Applications with patterns, Models, Frameworks and Tools'" P*roc. of 18th ACM Conference on OOPSLA03,* pp. 16-27.

---

[3] Distributed concepts can be viewed as *aspects*. However, we prefer to use different name to avoid the confusion between the two notions, as each is handled differently.

7. H.S. Hamza, “Separation of concerns for evolving systems: a stability-driven approach,” *Proc. of 1st ICSE05 Workshop in Modeling and Analysis of Concerns in Software (MACS2005)*, pp. 57-61, 2005.

8. H.S. Hamza and J.S. Deogun,“Architectures for WDM Benes network with simalteneoues space-wavelength switching capabilities,” *Proc. 2nd IEEE Int. Conf. on Broadband Networks (BROADNETS 05)*, (To appear).

9. K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson, “Feature-Oriented Domain Analysis (FODA) Feasibility Study, *Tech. Report CMU/SET-90-TR-021*, SE I, Nov. 1990.

10. R. McCain, “Reusable software components construction: a product oriented paradigm, *Proc. 5th AIAA/ACM/NASA/IEEE Computers in Aerospace*, 1985.

11. H. Mili et al. Reuse-based software engineering. John Wiley Sons, Inc. 2002.

12. J. Neighbors, “Software Construction using components, P*hD Thesis, Dept. of Information and Computer Science*, U. of California, Irvine, 19981.

13. D.L. Parnas, “On the criteria to be used in decomposing systems into modules,” *Communication of the ACM*, vol. 15, no.12, 1972.

14. D.L. Parnas, “On the design and development of program families,” *IEEE Transactions on Software Engineering*, March,1976.

15. R. Prieto-Diaz, Domain analysis for reusability,” P*roc. 11th Annual International Computer Software and Applications Conference (COMPSAC)*, 1987.

16. P. Predonzani, G. Succi, and T. Vernazza. Strategic software production with domain-oriented reuse. Artech House, 2000.

17. W. Vitaletti, and E. Guerrieri, “Domain analysis within the ISEC rapid center, *Proc. 8th Annual National conf.* on Ada Tech., 1990.

18. R. Wille, Restructuring lattice theory: an approach based on hierarchies of concepts. In I. Rival, editor, Ordered sets, Reidel, Dordecht-Boston, 1982, pp. 445-470.

19. Reuse-Driven Software Process Guidebook, *Software Productivity Consortium*, VA, 1993.

20. Reuse Adoption Guidebook, *SPC-92051-CMC*, SPC, VA, 1993.

21. SofTech, Inc., Domain Analysis and Design Process, *Do. 1222-04-210/30.1*, DISA-CIM Software Reuse Program Office,1993

22. J. Greenfield and K. Short. Software factories: assebling applications with patterns, models, frameworks, and tools. Wiley Publishing, Inc. 2004.

23. R. Pohjonen, J.-P. Tolvanen, “Automated Production of Family Members: Lessons Learned,” *Proc. OOPSLA International Workshop on Product Line Engineering The Early Steps: Planning, Modeling, and Managing* , 2002. http://www.plees.info/Plees02/plees02.htm