Run-Time Variability in Domain Engineering for Post-Deployment of User-Centric Software Functional Completion

1st Year Ph.D. Report
November 2003

Alexandre Manuel Tavares Bragança
Alexandre.Braganca@i2s.pt
Departamento de I&D
I2S Informática – Sistemas e Serviços, S.A.
Trv. Ribeiro de Sousa, 40, 4250-409, Porto

alex@dei.isep.ipp.pt
Departamento de Engenharia Informática
Instituto Superior de Engenharia do Porto / Instituto Politécnico do Porto
Rua Dr. António Bernardino de Almeida, 431, 4200-072 Porto

Thesis supervisor:
Ricardo Jorge Silvério de Magalhães Machado
rmac@dsi.uminho.pt
Departamento de Sistemas de Informação / Escola de Engenharia / Universidade do Minho
Campus de Azurem, 4800-058 Guimarães

Abstract: Domain engineering is a systematic approach aimed at capturing the domain knowledge in the form of reusable assets that can be applied in the development of new products. In order to achieve reuse in a domain, domain knowledge must be captured in terms of commonalities and variability. Commonalities enable reuse of software components in a domain. They result from software features that are common across applications in a domain. Not all common features are mandatory in all applications of a domain. This (feature) variability enables the differentiation of applications in a domain. It is, then, necessary for variability to be identified, represented and implemented. Some domains do not allow for variability to be resolved only at pre-deployment time. Unfortunately, methodologies and tools currently do not fully address these requirements. Their support for pos-deployment, or run-time, variability is very restricted.

In our work we propose to address this problem. This problem can be further detailed in two issues: one is how to select and compose already existing components at run-time; the other is how to resolve variability totally at run-time. Regarding the first issue, our research is based on work done in pre-deployment time. We propose that the configuration and composition of variable software parts at run-time can be based on composition techniques similar to a component based approach. In order to enable this there must be a proper run-time execution environment support. We will address this issue by proposing the necessary run-time execution environment characteristics. This kind of variability can be solved at run-time by the end-user selection of the appropriate variation in a variation point from already existent software components. For the second issue, the end-user is required to specify the functionality of the variability point because in high dynamic domains it's not always possible to have pre-built components with all possible alternative functionalities for a variation point. For this issue we propose the adoption/building of domain-specific languages with appropriate characteristics that will enable the functional completion of the software by the end-user.

Keywords: domain engineering, domain-specific languages, feature modeling, component composition, run-time execution environments, end-user programming, software reuse.

Table of Contents

1. Introduction

The development of software systems is still a very hard and difficult engineering process. In fact, the main aim of software engineering, according to Fritz Bauer is "The establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines" [Naur *et al.* 1969]. To achieve these goals Pressman states that software engineering encompasses a set of three key elements: methods, tools and procedures [Pressman 1994]. In the context of these elements, various software engineering paradigms have been proposed and used. Examples are the waterfall model the spiral model or Rational Unified Process (RUP) [RUP].

All these paradigms aim at provide sound engineering principles. Even if many of these paradigms have been widely adopted there is still very hard to, for instance, make accurate predictions of a software project delivery date. If the duration of the project is not accurate then the project is not economically feasible. In order to maintain the economically feasibility of the project normally the product outcome will be less reliable and efficient. It may even be economically worst because of maintenance cost that came out of the poor reliability and efficiency of the product.

Recently more pragmatic approaches have been proposed like Extreme Programming [Beck 1999]. One such pragmatic approach is based on the intuitive concept of reuse. The reuse approach is based on building new software systems reusing already existing and proved artifacts. With this approach software engineering projects become more predictable. Particularly, predictions of costs and delivery dates become more accurate. Software reliability can also improve because of the reuse of already tested and proved artifacts.

In the past, reuse has been adopted in the industry with relative success. Examples are the use of class libraries like wxWindows [wxWindows] or object-oriented frameworks like Java [Java]. These all have the benefit of provide the programmer with the possibility of reuse code that deals with programming needs like implementing graphical windowing systems or data containers structures like arrays and lists. However, these are all reuse of software of generic nature. The advantages of software reuse can be much more if exploited in specific domains.

Product families and product lines aim at promote reusability within a given set of software products [Bosch 2000]. Software product lines have achieved substantial adoption by the software industry. The adoption of Product line software development approaches has enabled a wide variety of companies to substantially decrease the cost of software development, maintenance, and time to market and increased the quality of their software products [Bosch 2002].

To accomplish reusability among various software products there must be common characteristics among them. Normally this means that the various software products must share the same domain. Therefore, an organization that has built several software systems in a domain also has acquired very good knowledge of such a domain. This knowledge can be used when building new software systems in the same domain. A fundamental technical requirement for achieving successful software reuse is the systematic discovery and exploitation of commonality across related software systems [Prieto-Diaz 1990].

By capturing the acquired domain knowledge in the form of reusable assets and by reusing these assets in the development of new products, the organization will be able to deliver the new products in a shorter time and at a lower cost [Czarnecki 1998]. Domain engineering is a systematic approach to achieving this goal. As such, domain engineering is the foundation for emerging

product line software development approaches [Foreman 1996].

So we can say that reuse has to do with finding commonalities among software systems within a domain. Nonetheless, to build diverse software systems within a domain we also need to specify variability. Domain engineering focuses on supporting systematic and large-scale reuse by capturing both the commonalities and the variability of systems within a domain to improve the efficiency of development and maintenance of those systems. As such variability is one of the key aspects of domain engineering.

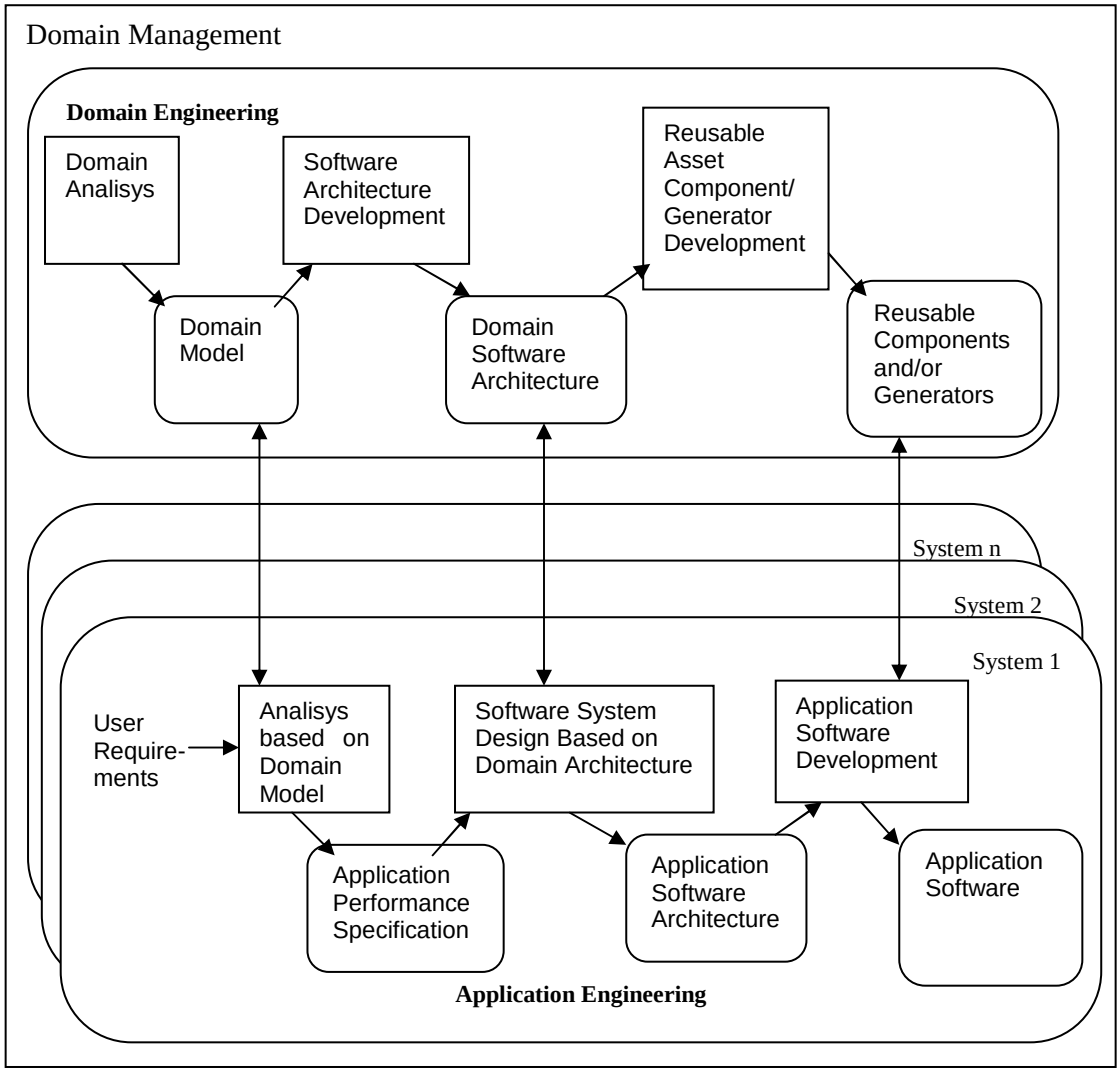

Figure 1. Domain Engineering vs. Application Engineering [SEI]

Figure 1 depicts the life cycles of domain engineering and application engineering based on the Software Engineering Institute (SEI) [SEI] at Carnegie Mellon University.

In the figure it is clear that domain engineering has to do with *engineering for reuse* and application engineering with *engineering with reuse*. In Figure 1 it is also clear that the application engineer must reuse artifacts from domain engineering to instantiate a new application in the domain. This new application will have common functionalities with others in the domain but will also have

differences that make it a particular instance of that domain. This means that the notion of variability and the methods, techniques and technology used to achieve variability are one of the most important issues in domain engineering.

Variability in software is achieved fundamentally by the following techniques [Svahnberg and Bosch 2000; Gurp 2003]:

- *Inheritance*, is used when the variation point is a method that needs to be implemented for every application, or when an application needs to extend a type with additional functionality
- *Extensions and extension points*, are used when parts of a component can be extended with additional behavior, selected from a set of variations from a variation point
- *Parameterization, templates and macros*, are used when unbound parameters or macros expressions can be inserted in the code and later instantiated with the actual parameter or by expanding the macro
- *Configuration and module interconnected languages*, are used to select appropriate files and fill in some of the unbound parameters to connect modules and components to each other. By configuration is meant the process in which source code is selected from a code repository and put together to form a particular product. Module interconnection languages are one way of describing configurations
- *Generation of derived components*, is adopted when there is a higher level language that can be used for a particular task, which is then used to create the actual component

There are also more recent techniques and methods that came from the academia but have limited adoption in the industry, such as: aspect oriented programming [Kiczales *et al.* 1997], subject oriented programming [Ossher *et al.* 1994] or generative programming [Czarnecki 1998]. These techniques tend all to resolve the variability issues at compilation-time. Thus variability is solved mostly at pre-deployment and deployment (installation) time.

There are some cases in witch solving variability at compilation-time and even at deployment time is not a satisfactory solution.

Let's take, for instance, the case of a system aimed at the insurance industry. One common situation in this domain is the need that the insurance company has to market new insurance products. Products in the insurance market are very complex. A new insurance product may imply that new and variable data may be necessary to register for each new insurance policy of that product. New rules for risk assessment and claim processing may also be needed. These are some of the many variability points that such a system may need to cope in the case of the market of a new insurance product. We can say that such a software system may have significant change in behavior for each insurance product. Insurance companies also need to market new products very quickly in order to respond to market needs.

In the described scenario if we have, for instance, 10 insurance companies using the software system and an average of 20 insurance products for company in the case of software variability solved at compilation-time we must have 10*20=200 variations of the software system.

Even if technically one could manage such a number of diverse versions of the system one significant problem remains: when an insurance company needs to market a new product it is necessary for the software engineer (in the software house) to build the new variation point in the system (i.e., a new application). This will encompass the engineering process that takes time, and also time to deploy the new application. Such a time frame may not be feasible because of the constraints of the time to market for the new product.

In the study we made so far it is possible to identify and resume the limitations regarding variability in domain engineering in the following three problems:

(1) Variability is mostly taken into account before or during application deployment. As observed this is not always the best solution.

(2) At the same time, and also resulting from the previous problem, the domain engineering methodologies do not take into account the possibility of the application customer having a special role in a pos-deployment phase. This special role could enable the variable points of the application to be resolved at run-time.

(3) Finally the techniques used to achieve variability do not take into account the technology heterogeneity of most real cases. For instance, in most techniques it is implicit the adoption of object-oriented languages.

In this thesis we will face the dynamic run-time software variability problem, in particular the three major problems described above. For such we propose the adoption of a domain-specific language and framework.

Thesis statement

The adoption of domain-specific languages in domain software engineering can significantly improve product line variability and flexibility by (a) enabling an effective technique for implementing run-time variability; (b) enabling the possibility of a pos-deployment kind of software engineering; and (c) enabling variability in a technically heterogeneous product line.

Regarding the first problem, we will evaluate to what extent it is feasible to specify the variability of an application at run-time using the proposed technique. In order to do so a domain-specific language and framework will be evaluated in a real case of an insurance software product line. A pragmatic approach will be used because this evaluation will be done in a real case. Feedback from end-users, software engineers and domain experts will be registered and results from this real case will provide further insight into the viability of our propose regarding run-time software variability. These results can then be compared with documented results from other techniques of software variability.

Our approach raises the second problem mentioned above. Who will solve the run-time variability? The end-user? The end-user in a special role of *a kind of* software engineer? Our approach should try to answer these questions. In order to do so it is necessary to revise the software engineering methods, particularly the domain engineering methods, so that dynamic variability and its method's implications can be "formally" integrated in a methodology. Regarding this problem we propose the adoption of existing and proved methodologies with the necessary adaptations.

There are diverse domain engineering methodologies such as Feature-Oriented Domain Analysis (FODA) [Kang *et al.* 1990], Organization Domain Modeling (ODM) [Simos *et al.* 1996] or Draco [Neighbors 1984]. There are also various similarities between the methodologies. For instance, it is very common to adopt feature diagrams as a mean to do domain analysis (i.e., problem space analysis). The limitation of such methodologies is more evident at the solution space. Methodologies often do not specify how to instantiate a domain architecture, specially the case of run-time features. Documented examples normally regard only object-oriented languages and achieve variability of software at compile-time [Deursen *et al.* 2002; Jaring *et al*. 2002]. This has to do with the third problem. To our knowledge, there are few document examples of variability in real product line applications. We believe that most real product line applications are technically very heterogeneous. This is also the reality of the product line that is the case study of this thesis. Addressing the variability problem in a heterogeneous environment raises different challenges. We

also propose that domain-specific languages can have a significant impact in addressing this problem. Because our case study can be classified as an heterogeneous system we expect our thesis to give some significant contribute also in this field.

In the remainder of this report, section 2 presents the context of our work. It also addresses and discusses related work and research. Section 2 is mostly a synthesis of the state-of-the-art of the thesis field that resulted from the planned work for the first year. Section 3 details the proposed approach in a technological and methodological level. In section 4 we present our working plan for the next two years. In this section we cover the research context, i.e., the research method. We also present the case study and how we will be doing research in an enterprise environment. This section also covers how we intend to validate our work. Finally, section 5 presents the expected contributions.

2. Related Work

Our research work has to do mainly with the field of domain engineering. Nevertheless this is a vast field of research since it is linked with other fields and various techniques such as application engineering, software reuse, product lines, automatic programming, architecture design and software frameworks just to name a few.

The major focus of our research is variability in software. Basically it means how to analyze, design and implement variability. Particularly our work focuses on how to cope with extremely dynamic applications such as the ones described in the previous section.

2.1 Domain Engineering

One can say that domain engineering started with the work of Dijkstra regarding *structured programming* and the notion of *programming for reuse* [Czarnecki 1998].

The next major reference is the work of Parnas on program families [Parnas 1976]. Parnas stated why one should study program families instead of individual programs. He also stated that a set of programs is considered a family when it is the case that in order to study this set, it is necessary to study the common properties among the elements of the set first, and then study the properties of the individual family members. He also stated that in a program family one should first study the commonalities (common features) and then the variability (diverse features) of each program.

The work of Neighbors is also of major importance. He introduced the first domain engineering methodology, named Draco, in his Ph.D. [Neighbors 1980]. In his thesis he argues that many software systems are very similar and so should be built out of reusable software components. He also states that for reuse to be successful it is necessary to reuse analysis, design and code and not only code. Neighbors states that “the concept of domain analysis is introduced to describe the activity of identifying the objects and operations of a class of similar systems in a particular problem domain”.

Draco uses domain-specific languages, prettyprinters, source-to-source transformations and software components. Draco is based on the assumption that we can specify a program in a high level domain, i.e., the problem domain, and them transform that program successively into others domains, until we get a program in an executable domain. Achieving this we have a solution to the initial problem in an executable format. We can say that Draco is a transformational system.

The method is based on the definition of several domains with the respective domain-specific language and transformations between the domains. The *initial* domain is the domain of the problem or the business domain. A program in the domain is specified using this domain-specific language.

One can say that the Draco system was a precursor of domain engineering methodologies. It is also accurate to say that Neighbors work influenced almost every methodology or technology in the field of domain engineering and also other fields. In particular more recent works are also based on transformation technology. Examples are Intentional Programming [Simonyi 1995] and GenVoca [Batory and O’Malley 1992], which are presented in section 2.1.3.

There aren’t so many well-documented domain methodologies. There are also less documented applied case studies. The Software Engineering Institute at Carnegie Mellon University is one of

the exceptions. In fact, in 1990, SEI published a technical report regarding Feature-Oriented Domain Analysis [Kang *et al.* 1990]. This technical report presents a feasibility study regarding Feature-Oriented Domain Analysis. To our knowledge this is the first method witch claims it self to be a domain engineering method. One major advantage of this methodology regarding others is that it has much public available documentation. Another advantage is that much of this documentation regards applied cases.

The primary goal of the method is to provide a basis for understand and communicate about the problem space addressed by software in a domain. In order to achieve this goal the method is based on the examination and study of a class of related software systems and the common underlying theory. The result should be a reference model that describes the class of software systems. The method also proposes a set of architectural approaches for the implementations of new systems. This means Feature-Oriented Domain Analysis, as the name implies, is focused on analysis of the domain, i.e., the analysis and representation of the problem.

Given the central focus of the method is domain analysis it encompass basically three phases:

- Context Analysis: defining the extent of a domain for analysis;
- Domain Modeling: describing the problems within the domain that are addressed by the software;
- Architecture Modeling: creating the software architecture(s) that implements a solution to the problems in the domain.

Each of the phases of the domain analysis method is composed of several activities. The results of these activities are documents that describe domain knowledge. These documents define the scope of the domain, describe the problems solved by software in the domain and describe architectures that can implement solutions.

As we can see the method has one phase for architecture modeling, witch could mean it also addresses the generation of (or support for) software solution. As we will see later this is not entirely true. In fact, the original method is very vague in how to evolve from the problem representation into the solution space.

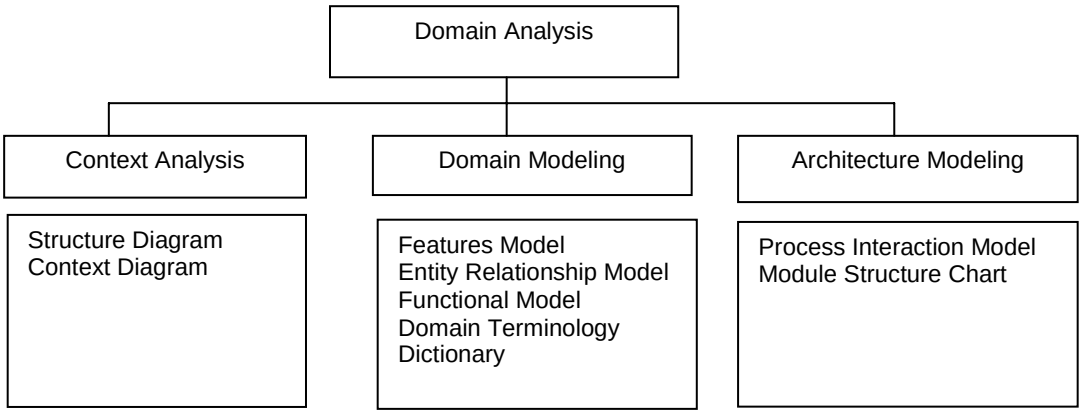

Figure 2. Phases and Products of Domain Analysis [Kang *et al.* 1990].

The method also defines the possible roles of participants in the domain analysis process: end user, domain expert, domain analyst, requirements analyst and software designer. These roles can be further classified by their 'relation' with the method. The end user and the domain expert are sources to the method. The domain analyst is a producer. Requirement analyst, software engineer and also the end user are consumers. Is to be noted that the end user can be a source and a consumer of the method.

Figure 2 represents the phases and products of Feature-Oriented Domain Analysis. The first two phases undoubtedly correspond to the domain analysis phase in Figure 1. One could say that the software modeling phase is related to the software architecture development phase in Figure 1. Nevertheless this is not the case because, as we will see, the architectural models resulting from architectural modeling are not sufficient to be the base for software component generation, i.e., reusable software artifacts.

One very important aspect of methods is the roles people, or systems, play. If we refer again to Figure 1 we can imagine a requirements analyst and a software designer using the products of a domain analysis when implementing a new system in the domain. In this scenario we can also imagine a domain analyst using the feedback from the implementation of new systems to further extend and evolve domain analysis. In Draco, for instance, there are four human roles: system builder, domain builder, domain user and system specialist.

In domain engineering, the domain products (i.e., the ones presented in the first half of Figure 1) represent the common functionality and architecture of applications in a domain. These are generic and should be reused in the development of new systems in the domain. The generic nature of the domain model implies that there is variability in the possible implementations of applications (systems) in the domain. The development of new systems in the domain requires refinements in the products of domain engineering so that the specificity of new system can be achieved. When we add specificity we are removing the variability of the domain model, i.e., we are selecting one of the possible choices of implementation. The process of removing generality - or adding specificity - in order to build a new system needs some mechanism to implement variability.

As already mentioned, the products of domain engineering are artifacts that can be reused in building new systems in the domain. These artifacts can be abstractions of functionalities or designs (i.e., architecture) to be reused in the development of new systems in the domain. Product frameworks and product lines are based on the reuse of such abstractions [Bosch 2000]. Thus, one can say that domain engineering should be used in the development of product lines and product frameworks.

In a domain there are common parts that represent invariants of the domain. If we are talking about product frameworks or product lines we can say these common parts are software components that implement invariant functionalities (abstractions). In order to differentiate between diverse products in a product line or different applications using a framework we need the referred variability mechanism. On the modeling phase there is also the need to represent variability. Some usual design concepts that can represent variability are: aggregation/decomposition, generalization/specialization and parameterization.

Aggregation/decomposition and generalization/specialization are modeling concepts very familiar to object oriented programmers. With aggregation, grouping several abstractions creates a new abstraction. Decomposition is the inverse of aggregation. When we decompose one abstraction into its components we are refining that abstraction.

When we create one abstraction by using the commonalities between abstractions we are generalizing. With generalization abstractions lose specificity. Specialization is the inverse of generalization. An abstraction is specialized when we add features to the abstraction. Specialization is also a refinement.

Parameterization is a technique in witch software is adapted/configured by substituting the values of the parameters in the software.

In Feature-Oriented Domain Analysis these techniques are used in order to model feature variability in a domain. As the name implies, features are the core concept of Feature-Oriented Domain Analysis. We can think of a feature as a characteristic of a concept. The method uses features to represent characteristics of concepts of the domain. Some of these features are invariant in the domain. Others can vary and there may also be rules in the selection and composition of features in a domain. The use of features in domain modeling is very used because the terminology used is very close to the end user and the domain experts. As such domain models that use features can be easily understand.

Feature Oriented Reuse Method (FORM) is an evolution of Feature Oriented Domain Analysis [Kang *et al.* 1998]. In this evolution features became the central concept of domain engineering and feature models are used not only in requirements engineering but also in the design phase (architectures) and in the building of software components. This means that FORM extends the utilization of features from the domain problem space into the decision and the solution space.

Several more work and research has been done in the field of domain engineering. One other major domain engineering methodology is Organization Domain Modeling (ODM) [Simos *et al.* 1996]. To our knowledge ODM is in structure similar to Feature Oriented Domain Analysis but its process is far more elaborated and detailed.

Another reference is Reuse-Driven Software Engineering Business (RSEB) [Jacobson *et al.* 1997]. This method is based on Object-Oriented Software Engineering process (OOSE) [Jacobson *et al.* 1992]. It extends OOSE with a more reuse-oriented process. This method also adds an interesting approach: the adaptation of an existing object-oriented analysis/design method used for application engineering in order to use it in 'engineering for reuse'. The RSEB extends OOSE with architectural constructs for families of related applications built from reusable components. It is also interesting because it uses UML as the base notation [UML].

FeatuRSEB is a more complete example of the integration between domain engineering and application engineering [Griss *et al.* 1998]. As the method name implies it is a merge between Feature Oriented Domain Analysis and RSEB.

There are other areas that related to domain engineering. As we saw, architecture modeling is a very important activity in domain engineering. One area of particular interest is how to describe a software architecture [Garlan and Shaw 1990]. Architecture Description Languages (ADLs) are used to describe the components, connectors, and information about their interactions that compose a system. There are several languages and tools that can be classified as ADLs [Medvidovic 1997]. This is also a very active field of research that is very strong related to domain engineering. For instance, the concept of software architecture is central to the Domain-Specific Software Architecture (DSSA) method [Hayes 1994]. We can see DSSA as an application of the concept of software architecture in a domain. From Figure 1 we also can see that a software architecture for the domain is one of the outputs of domain engineering.

The Object Connection Architecture (OCA) was presented in [Peterson and Stanley 1994] as a method that uses the outputs of Feature-Oriented Domain Analysis to build a generic design for the domain. This generic design encompasses software components that conform to the software architecture model proposed for structuring software systems in OCA.

Software patterns are another field of active research and practice. Patterns can be considered a micro-architectural view of a system. Most work in patterns is from a component or design reuse perspective. Software patterns are widely used in the development community because they normally originate from best practices [Fowler 2002].

There are also other methods and techniques that can relate to domain engineering because of the focus they put on reuse. Examples are the OOram method [Reenskaug *et al.* 1996] and the work on software frameworks [Fayad and Johnson 1999].

The main focus of our research is the concept of variability in domain engineering. Variability is used to differentiate applications in a domain. In contrast, commonalities are shared by applications in a domain. For a domain to exist there must be commonalities between applications. For a domain to be useful variability must be identified, represented and implemented. In the next three sections these tree main topics of our research will be detailed.

2.1.1 Variability Identification

Variability identification is one of the first steps in domain engineering. In fact, commonalities must be first discovered in order for a domain to exist.

A domain analyst using diverse sources of information performs the initial analysis of a domain. The sources of information and particularly the process of analysis are better documented and more precise depending on the domain engineering methodology. For instance, in Draco, this process is not explicitly and precisely documented.

In Draco, the analysis of a domain should result mainly in a domain-specific language, components of that domain (objects and operations of the domain) and rules for transformations and refinements. A domain is specified by giving its syntax, guidelines for printing (prettyprinter), rules for simplifying relations between objects and operations and semantics in terms of other domains already known to Draco. Draco starts with domains that describe executable computer languages. The idea in Draco is to refine the program from the problem domain into an executable domain.

Being one of the first domain engineering methodologies, Draco is also a domain methodology very oriented because it is based on the central concept of domain languages and transformations between them. As such the final output of domain engineering is always a domain-specific languages and tools. Because of that the initial analysis phase of the method is very oriented towards identifying language constructs.

The Draco domain engineering method is based on four human roles:

- *Draco system builders* which are the builder of the mechanism and also the designers of the specification languages for the different domain parts;
- *Domain builders* which encompass the domain analyst who tries to discover the objects and operations of a domain and the domain designer who accepts the results from the analysis as a base to design a new domain language;
- *Domain users* which encompass the system analyst who uses an available Draco domain as a framework for his analysis of a specific problem and the systems designer who accepts

the analysis of a specific system from the systems analyst and uses a domain language to describe the system;

- *Draco system specialist* who refines the specification of a problem into an executable target language by navigating through the modeling domains of Draco.

These roles fit very well with the domain-engineering life cycle as described by SEI and illustrated in Figure 1. The roles of Draco system builders and Domain builders related to domain engineering and the roles of Domain users and Draco system specialist relate to application engineering.

We can see from the description of the major roles of Draco that the domain analyst is basically discovering the objects and operations in the domain. This information is then passed into a domain designer that designs a language for the domain. The new domain language is construct by the domain designer based on domains already know to Draco. All these domain models integrate a library of domains.

The Draco method targets situations where typically an organization is building systems in a domain. At the application engineering level as applications are build they can use the Draco system or not. Even if applications are build outside Draco the knowledge that results from the process can be an input for the domain analyst. We can say that one major input of knowledge about the domain is the actual organization experience in building applications in the domain. The Draco method suggests there can be other sources of information such as documents about the domain. The method also states that after the identification of the objects and operations of the domain the domain designer can specify the syntax of the domain language. What the method doesn't state is how to do this. Draco also doesn't specify clearly how to identify the objects and operations of the domain. These objects and operations represent in fact the commonalities of a domain. The variability in Draco is the way we can combine these operations and objects. Since these combinations are in fact only limited be the grammar of the language the results of the domain analysis identify a very wide scope of variability. In fact all the possible programs we can build with the domain language.

Other domain methods take a more detailed approach into domain analysis. For instance in Feature-Oriented Domain Analysis there is a first phase called context analysis. In this phase the domain analyst interacts with users (of possible applications of the domain) and domain experts to establish the bounds of the domain and also the proper scope for the analysis. In this phase the analyst also gathers sources of information for performing the analysis.

The objective of the initial phase of the method is the definition of the scope of a domain in terms of the probability that the domain will give usable domain products. The relationships between the domain and the external elements are evaluated. The degree of variability of the domain is also evaluated. The availability of domain sources (experts, documentation, etc.) is also used to scope the domain.

As depicted in Figure 2 the documentation resulting from context analysis is the structure diagram and the context diagram.

The structure diagram is used to show the relations between the domain and other domains. This type of diagram includes higher, lower and peer level domains regarding the domain in study. Higher domains are domains that include the domain. Lower level domains, or sub-domains, are domains that are in the scope of the domain but are well understood. All other domains (peer domains) that interface with the target domain should also appear in the diagram.

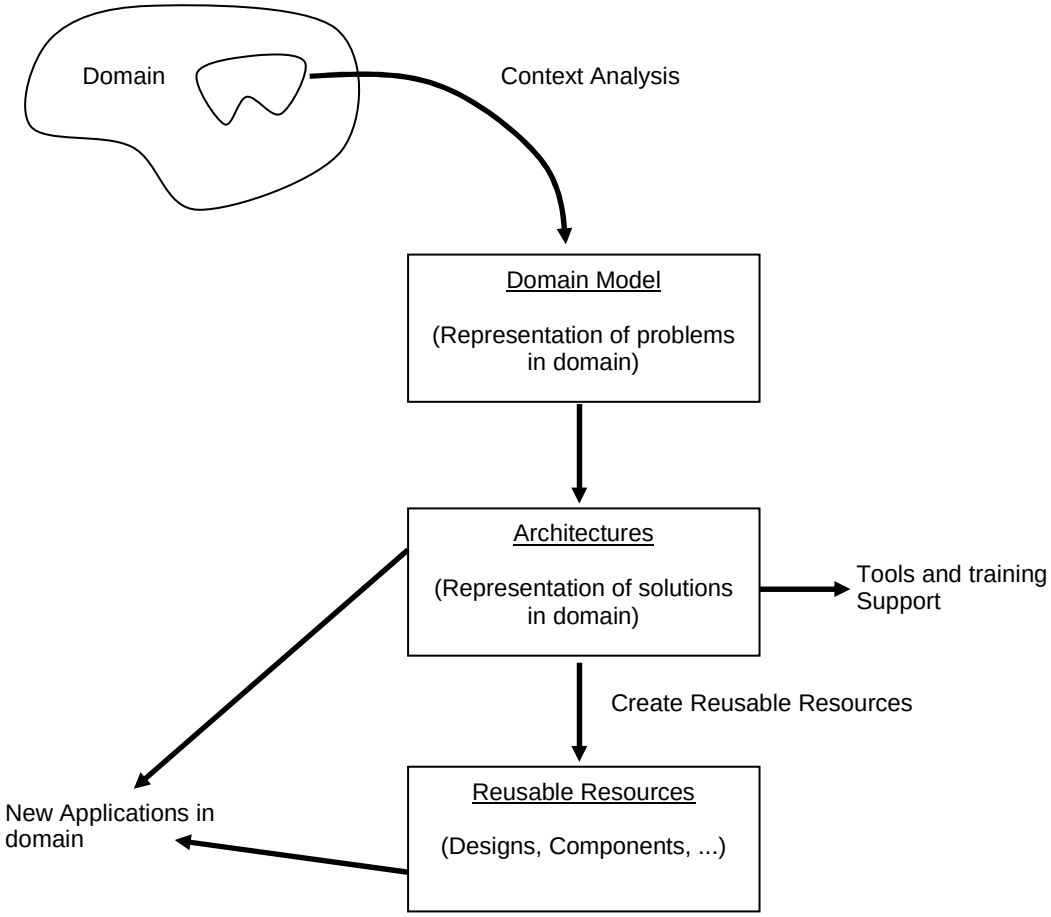

Figure 3.Domain Engineering Supports Software Development [Kang *et al.* 1990].

The context diagram is basically a top-level data-flow diagram of the interfaces the domain has with other domains or entities. The particularity of this data-flow diagram is that the variability of the data-flows across the domain boundary must be indicated. If the variations are due to different features of the applications in the domain this fact must be described. Because features are only introduced in the phase of domain modeling this means that context analysis and domain modeling may be done in parallel. Entities that appear in the context analysis must be described.

Domain experts, end-users and documents and applications of the domain are all sources of knowledge that the domain analyst should use in the context analysis.

The method uses aggregation and generalization to capture the commonalities of the applications in the domain in terms of abstractions. Refinements are used to capture the differences between applications. Parameterization is used to specify the context of the refinements. As such, one can say that the result of the method is a group of abstractions of a domain and a series of refinements of each abstraction with parameterization. When a new refinement is introduced in the domain, the context in witch the refinement is made must be defined in terms of parameters.

Parameterization is the technique used by Feature Oriented Domain Analysis to select the refinements of the domain abstractions. With the refinements new applications in the domain can be specified.

Parameterization is based on identifying the factors that result in different applications in a domain. The method classifies these factors according to:

- The *capabilities* of applications in a domain from the end-user's perspective;
- The *operating environments* in witch applications are used and operated;
- The *application domain technology* (methods and techniques specific to the domain) based on witch requirements decisions are made;
- The implementation techniques.

These factors represent different aspects of a domain that can be parameterized when building a new application. As the name of the method imply, features are a major concept used in domain engineering. The method uses *features* to model (parameterize) the capabilities of applications from the end-user perspective.

The method advocates that the initial level of abstractions should be high. This maximizes reuse of the products of the domain. Also, the refinements that make abstractions more specific should be delayed as much as possible. This maximizes reuse but also minimize the added value (or productivity) of using a generic abstraction. The method provides abstractions (components) at different levels of refinement so that they can be reused as needed.

We shall see in the next section that features and feature modeling are key tools in representing variability in domain engineering.

2.1.2 Variability Representation

To allow effective modeling of domain knowledge in order to support the engineering of applications in the domain variability must be represented and documented. In Figure 3 we can see that domains models should support the definitions of architectures (reference architectures in the domain) witch are then used in the building of applications in the domain. In the process of application building, the variability aspects (i.e., features) of the domain models will normally disappear. They disappear because the system engineer will select what variations will be used in the new application. We will see that features are an effective concept for representing variability and that feature modeling is used to document that variability.

Features and feature modeling are extensively used in Feature-Oriented Domain Analysis method to model variability. In the domain modeling phase the domain analyst uses the information sources and the other products of the context analysis to support the creation of a domain model.

In Feature-Oriented Domain Analysis after having scoped the domain in context analysis it is now necessary to identify the commonalities and variability of the problems addressed by the applications in the domain. This is done in the domain modeling phase. To achieve this the modeling phase has three major activities: feature analysis, entity-relationship modeling and functional analysis. This means that apart from feature modeling the method also uses other tools to model the domain knowledge. Let's see how this domain knowledge, and in particular variability, is captured and modeled in Feature-Oriented Domain Analysis.

Feature analysis allows the domain analyst to capture the diverse capabilities of the applications in the domain according to the end-users. This is a very productive analysis tool since it models the

problem space from the end-user's viewpoint. Thus, feature diagrams normally don't include technical capabilities. The viewpoint of the user is normally centered in the services or functionalities provided by applications and operating environments in witch they run.

Because, as we will see, the feature diagram captures the commonalities and variability of the applications in the domain, the features presented in the feature model are used to generalize and parameterize other models.

Features are characteristics of the system that make sense to the end-user. The feature diagram depicts characteristics of the system from the viewpoint of the end-user. The features (characteristics) that appear in the feature diagram should affect directly the end-user. For instance, programming technical features should not appear in a feature diagram.

Figure 4 presents an example of a feature diagram. A feature diagram looks like an inverted tree. The structure of the relationships between features is represented by the connectors and visual indicators that can be used in the diagram. For instance, in Figure 4, air conditioning is an optional feature, as denoted by the circle in the end of the feature line.

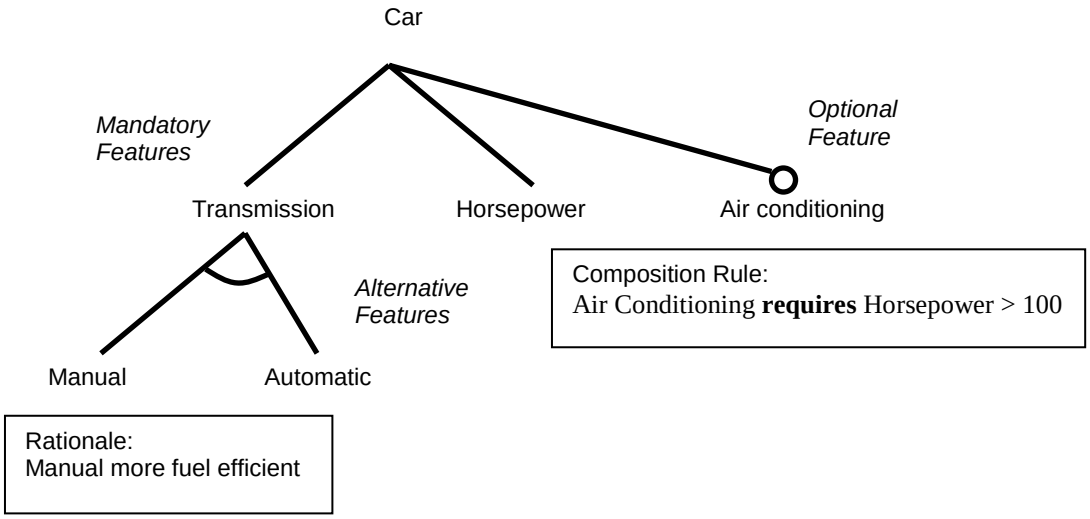

Figure 4.Possible Feature Diagram for a Car [Kang *et al.* 1990].

There can be rules regarding the combination of features that cannot be specified only by visual indicators. In this case the method uses what is called composition rules. In Figure 4 there is a composition rule that says the air conditioning feature, when present, requires that the feature horsepower have a value greater than 100. Composition rules are used to define the semantics existing between features that are not expressed visually in the diagram.

Feature diagrams are a rich and useful tool regarding their expressiveness for documenting variability (mainly) from the end-user perspective. Features, and feature diagrams, are the basis for the specification of variation points in the final architecture of a system.

Apart from the visual diagram features also have a textual description. The proposed form for describing features is presented in Figure 5.

```
Name: <standard feature name>
Synonyms: <name> [FROM <source name>]
Description: <textual description of the feature>
Consists Of <feature names> [ { optional | alternative } ]
Source: <information source>
Type: { compile-time | load-time | run-time }
Mutually Exclusive With: <feature names>]
Mandatory With: <feature names>]
```

Figure 5. Textual description of a feature [Kang *et al.* 1990].

When the domain analyst is building the feature diagram and defining features he is doing this activity within the context of the information he gather from the end-users, domain experts and various documents about that domain. The choices he makes regarding optional and alternative features should be well documented or else future selection of features could be done in different knowledge context. Thus this knowledge is described in an "issues and decisions" documented attached to the features diagram. The "issues and decisions" has the original rationale of the major issues and respective possible decisions of the features diagram.

The proposed form for documenting issues and decisions according is presented in Figure 6. The "Raised at" statement indicates the component during the refinement of which the named issue was raised. The "Applies to" statement identifies the components that resulted by the decision.

```
Issue: <issue-name>
Description: <a textual description of the issue>
Raised at: <component name>
Decision: <decision name>
Description: <a textual description of the decision>
Rationale: <a textual description the rationale behind the decision>
Constraints/Requirements: <a textual description of any new constraints caused by, or any new
requirements derived from the decision>
Applies to: <component name>
```

Figure 6. Textual description of an "issue and decision" [Kang *et al.* 1990].

Features have to be 'transformed' into software constructs that realize the variation points. There are different moments when this is possible. These moments are binding times, when the feature is realized in terms of software. The method describe three possible binding times for the realization of optional or alternative features:

- *Compile-time*: features that are decided when the system is built and do not change. This kind of features should be realized at compile-time of the system (package) for efficiency reasons.
- *Load-time*: features that are defined only at the beginning of the execution of the system. These features remain stable during the execution of the system. This usually originates what is called 'table-driven' software.
- *Run-time*: features that can change during the run-time of the system. The method gives as example menu-driven software. One example of such is a word processor that can have the auto spelling checker feature active or not.

According to [Czarnecki 1998] these binding times are incomplete. In reality there may be other binding times, e.g. linking time or first call time (that is very important for just-in-time compilation). We can then generalize the binding time concept according to the specific times of the systems in the domain. For instance that can be specific times like debugging time and testing time. It is possible also to conceive special times in the life cycle of applications like off-line time or emergency time.

Apart from the binding time (when to instantiate the feature or component) of a feature there is also the problem of the binding local, i.e., the location of the feature (*where* to instantiate the feature or component). For this reason the concept of binding site was introduced to cover both situations [Simos *et al.* 1996].

Features modeling can be applied in diverse aspects or factors of a domain. Features can be classified in the same categories mentioned for the factors of parameterization: operating environments, capabilities, domain technology and implementation techniques.

Because features capture domain knowledge from the end-user perspective it's very natural that most features are in fact capability features. Features related to capabilities can be further categorized into three areas [Myers 1988]:

- functional features: these area basically services that are provided by the applications;
- operational features: these are related to the operation of applications ;
- presentation features: these are related to what and how information is presented to end-users.

These are just the most common categories of features. It is possible that more categories of features exist in a given case. These new categories can be identified in the analysis of a domain. The method does not discard this possibility.

One of the objectives of features is that they be used for the construction of software components. This has to do with moving from the problem space into the solution space. Implementation techniques must be used according to the analysis. The implementation techniques vary mainly according to the binding time of the feature. For instance, stable features with compile binding time can be build/packaged with preprocessor techniques or application generators and run-time features can be implemented as menu options.

Domain knowledge is not restricted to features of the domain. There is also the need to capture knowledge about the commonalities of the domain. There is the need to know what entities, objects or functionality exists in the domain. In order to do so new tools are needed.

As we saw feature modeling enable the capture of domain knowledge from the perspective of the end-user. This normally regards capability features.

Apart from then end-user perspective there is the need to capture more precise domain knowledge from an implementation perspective. Feature Oriented Domain Analysis does this using a kind of Entity-Relationship model. The purpose of this model is to represent the domain knowledge explicitly in terms of domain entities and their relationships, and to make them available for the derivation of objects and data definitions during the functional analysis and architecture modeling.

The Entity-Relationship model is based on Chen's method [Chen 1976] with the adoption of generalization and aggregation concepts from semantic data modeling that are used as predefined relationship types [McLeod 1978; Borgida *et al.* 1984].

The basic building blocks of the Entity-Relationship model are entity classes and *consist-of* (aggregation) and *is-a* (generalization) relationships. Entity classes represent classes of objects of the domain. *Consist-of* and *is-a* are kinds of relationships that can occur between entities. These kinds of relationships are of major importance in the method because they can be used to identify commonality and differences between entities and so provide the basis for the development of components that can be reused and parameterized. The model also provides for the definition and use of other types of relationships that may be of importance in the domain.

Given the fact that the entity-relationship model contains domain knowledge from the implementation perspective it is the base for identifying and derive objects and components. The method makes no assumptions regarding implementation technology. Object-oriented programming or other methods and techniques can be used.

In order to build an entity-relationship model for the domain various sources of domain knowledge can be used. This gathering of information permits the identification of the entities and their relationships and also the attributes of both. This activity will result in the identification and understanding of the major domain concepts and their terminology. This information should be recorded in domain terminology dictionary.

Functional analysis is the activity in the method witch purpose is to identify the functional commonalities and differences of the applications in the domain.

The feature model and entity-relationship model are used as guidelines in developing the functional model. The mandatory features and the entities are the basis for defining an abstract functional model. In this point the model will have the common functionalities in the domain. The alternative and optional features are embedded into the model during refinement. If during the process other factors are encountered that can cause functional differences they are defined using the "issues and decisions" approach. Issues and decisions are then used in the refinements for parameterization.

The method purposes two forms for describing the functional aspects of a domain: *Activitycharts* and *Statecharts*.

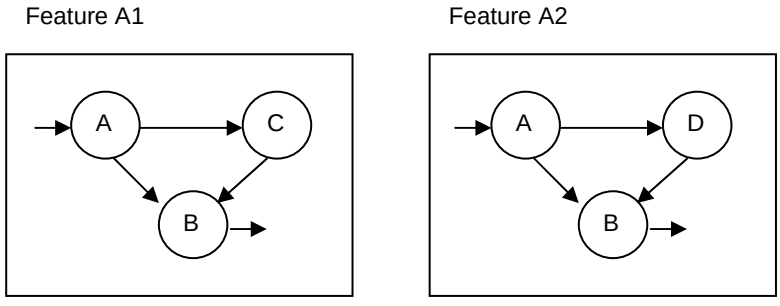

Figure 7. Developing separating components for each alternative.

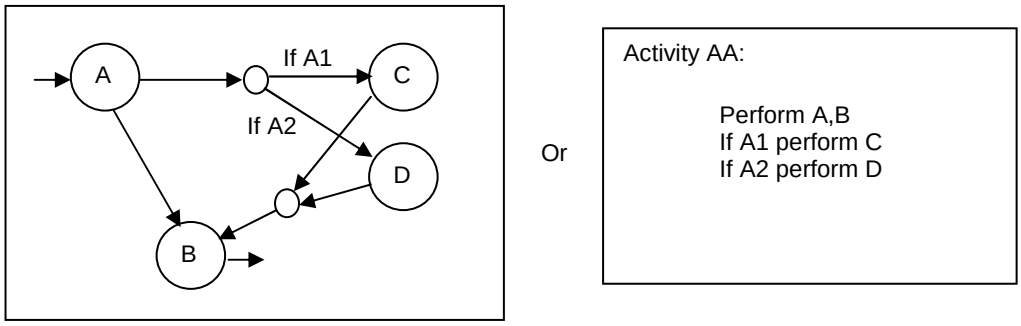

Figure 8. Parameterization for adaptation to each alternative.

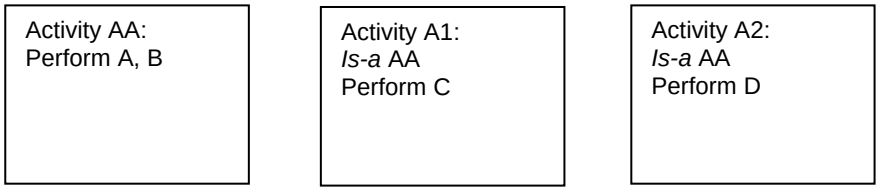

Figure 9. Each alternative is an instantiation of the general component.

Activitycharts describe functions of the domain in terms of inputs, outputs, activities, internal data, logical structures, and data-flow relationships. Statecharts specify the behavior in terms of events, inputs, states, conditions and state transitions.

The activity of functional analysis should start by the specification of a top-level abstract model of the common functionalities in the domain. This top-level model is then refined by the application of features and issues and decision. Every feature should be referenced when refining the functional model.

The method purpose three different ways to incorporate features and issues and decisions into the model:

- by developing separate components (refinements) for every alternative (Figure 7);
- by developing one component, but with parameterization for adaptation to each alternative (Figure 8);
- by defining a general component and developing each alternative as an instantiation of the general component (with an inheritance mechanism) (Figure 9).

This model should be built after the feature and entity-relationship diagrams. Feature, entities and relationships can be used to support the activity of functional analysis. For instance, alternative features in the feature model may be used to identify generic functions. Alternative features are specializations of a more general feature, and the functionality corresponding to the general feature is defined as a generic function witch is inherited by the functions implementing the alternative features. Also the generalization/specialization relationships (i.e., *is-a* relationships) of the entity-relationship model can be used to identify generic objects and the functionality associated with the generic objects.

These results from the domain modeling phase will be used in the requirements analysis when engineering a single system in the domain.

All the aspects mentioned so far regard mainly the problem space, i.e., modeling the knowledge of the domain. This knowledge must be used to build solutions. The advantage in domain engineering is that this knowledge is used in building a framework of artifacts that can be reused in application engineering in the domain. All the invariants of the domain can be build and even the possible variants. In each new application in the domain the invariants are already build and it is very possible that we only need to select witch variants to include in the new application. We are now dealing with the solution space, i.e., the building blocks and tools of the domain that can already be build and will be reused by the applications in the domain. This issue will be discussed in the next section.

2.1.3 Variability Implementation

We can think of applications in a domain as being concrete instances of generic architectures of the domain. Architectures represent solutions in the domain. They represent the structure of the applications in the domain. Architectures are also the base for building reusable artifacts that are used in the building of applications. These reusable artifacts are, for instance, software components that implement domain functionalities. When building an application in the domain the architecture of the application is an instance of a reference domain architecture and it uses software components that fit the architecture. The software components can be select from the features of the new application. Common features imply common components that are used in all applications in the domain. Variable features require support for variability in the solution space.

As depicted in Figure 2, Feature-Oriented Domain Analysis also encompasses an architecture modeling phase. The focus of this phase is shifted to the design of solutions in the domain. In this context the primary goal is to provide a base architecture to support the systems in the domain and also the building of software components to be reused (when building these systems).

The architecture model is a high-level design of the applications in a domain. Therefore, the method focuses on identifying concurrent *processes* and domain-oriented common *modules*, and on allocating the features, functions, and data objects defined in the domain model to the processes and modules. The packaging of functions and objects into modules must be done considering the processing time of the features (i.e., compile-time, activation-time, and run-time) that each module implements.

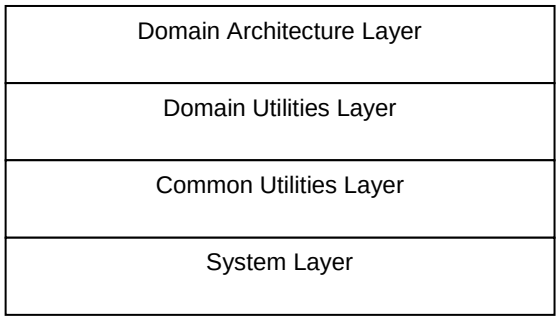

Figure 10. Architectural Layers in Feature-Oriented Domain Analysis [Kang *et al.* 1990].

In order to more easily cope with future changes in technology or others problems a layered architecture approach is adopted. The architecture is defined at various levels of abstraction so that reuse can occur at the level appropriate for a given application.

In terms of architecture an application is a collection of programs (i.e., processes) that can be compiled separately and executed in parallel. These processes can be defined based on the functional analysis. Each process must be designed as a hierarchy of modules with the allocation of functions and data objects defined in the data-flow model. Then, domain-oriented common modules that can be used across the applications must be identified to increase the reusability.

The method proposes the layered approach presented in Figure 10. At the top, the *domain architecture layer* is represented as a model showing the concurrent domain-processes and inter-connections between them. This model is called a process interaction model and is represented using the DARTS (Design Approach for Real-Time Systems) methodology [Gomaa 1984].

The *domain utilities layer* shows the packaging of functions and data objects into modules and the inter-connections between them. This is called module structure charts and is represented using the Structure Chart notations [Yourdon *et al.* 1978] following the DARTS methodology.

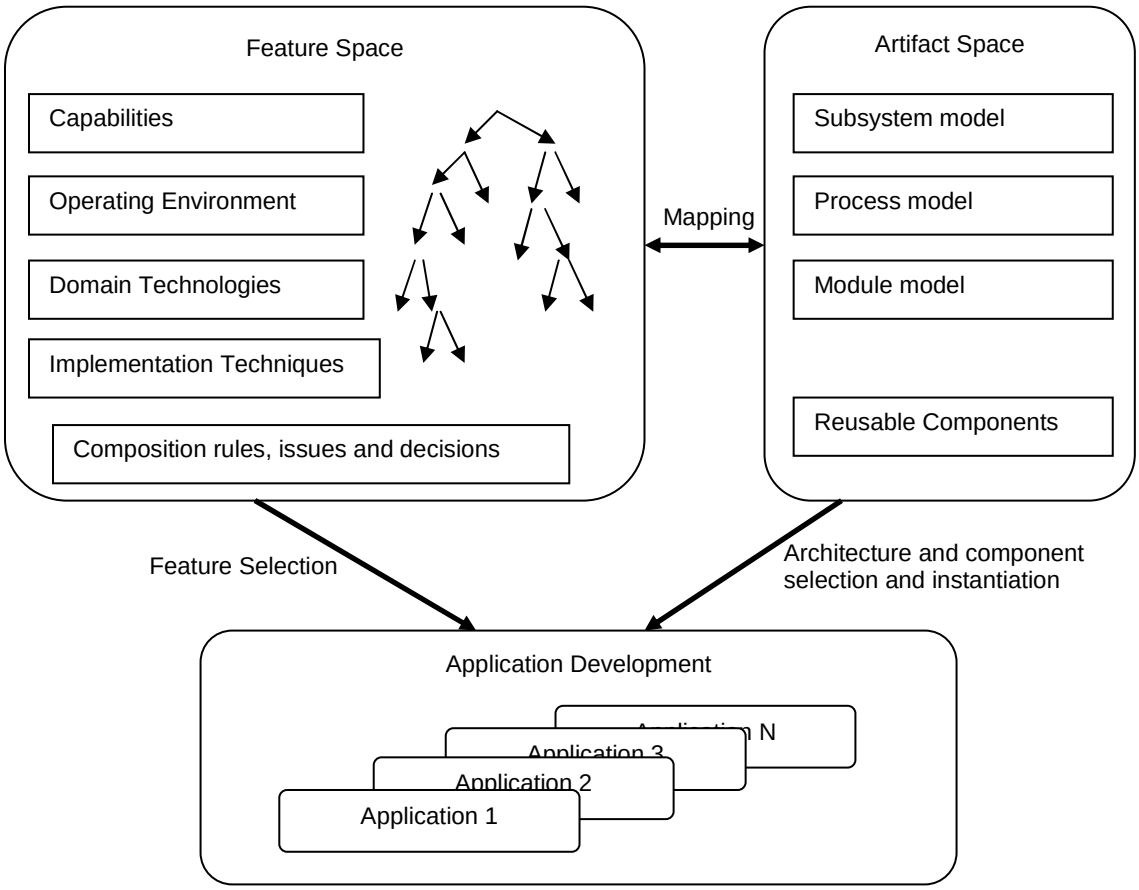

Figure 11. Feature Oriented Reuse Method [Kang *et al.* 1998].

The *common utilities layer* contains modules that can be used across different domains. Normally programming aspects that are common to applications of diverse domains are realized in modules in this layer (examples are synchronization and communication aspects). Aspects that regard the operating system or the programming languages are part of the *system layer*.

As we saw Feature-Oriented Domain Analysis presents some guides into what should be a domain architecture. Nevertheless it doesn't present much information regarding the process of going from the problem space into the solution space. This issue is addressed in FORM.

In FORM features became the focus of the method. As such the method establish more sound grounds for the mapping between the problem space and the solution space based on features. In Feature-Oriented Domain Analysis features are used mainly for the description of the problem space by the end-users. In FORM domain designers and application engineers also use features. The designers use features to build architectures and the engineers use features to build applications in the domain. Figure 11 presents the mapping between features and artifacts/components in FORM.

As with Feature Oriented Domain Analysis the FORM method also entails four major aspects related to features: capabilities, operating environment, domain technologies and implementation techniques. The major difference is that FORM focuses all the feature aspects and not only the capability features. By doing so the method captures not only the features of the domain applications in terms of functionalities but also the features in terms of implementation details. We can imagine these different kinds of features as layers. There can be dependencies between features in different layers. For instance, the selection of one functional feature may imply the use of one specific implementation feature.

The idea beyond the focus on features in domain engineering is that they are used from the analysis phase into the architecture and component building phase. By doing so, in the application engineering process the requirements phase can be done by selecting domain features of interest for the new application. This selection will guide the selection of the application architecture and the reuse of software components.

The adoption of features through all the engineering process raises one major problem: how to connect, or map, between the 'traditional' feature model used to describe the problem to a decision space and ultimately a solution space (components for reuse in application building)

As depicted in Figure 11 the artifact space has also layers. These layers represent reference architectures in the domain at different levels of abstraction. The method uses functional features mainly to identify required components, while non-functional (technical or implementation) features are used to partition components or to select type of connectors between components.

The subsystem level defines the overall system structure by grouping functions into subsystems that can be allocated to different hardware. This process is guided by matching capability features (witch are, by definition, top level features) to the required functions blocks (subsystems). Functions should be grouped in a manner that promotes cohesion so that related functions stay in the same subsystems. Because components of a subsystem will normally be allocated into a machine all the related functions should stay in the same subsystem. Also because of that, communication between subsystems should be carefully thought. There can be synchronous or asynchronous communications mechanisms between subsystems according to the needs. One

common architecture design pattern is the use of different subsystems to user interface functionalities and database functionalities.

The design of the subsystem level has to be done with a careful design of the interfaces between the different subsystems in order, for instance, to maintain an acceptable degree of performance.

The reference architecture at this level of abstraction is represented by the different subsystems (witch one normally represents a top level capability feature) and the connections between them. These connections can be synchronous or asynchronous.

Figure 12 presents an example of a subsystem model. We can see that in this example the user interface and database management system where considered external subsystems.

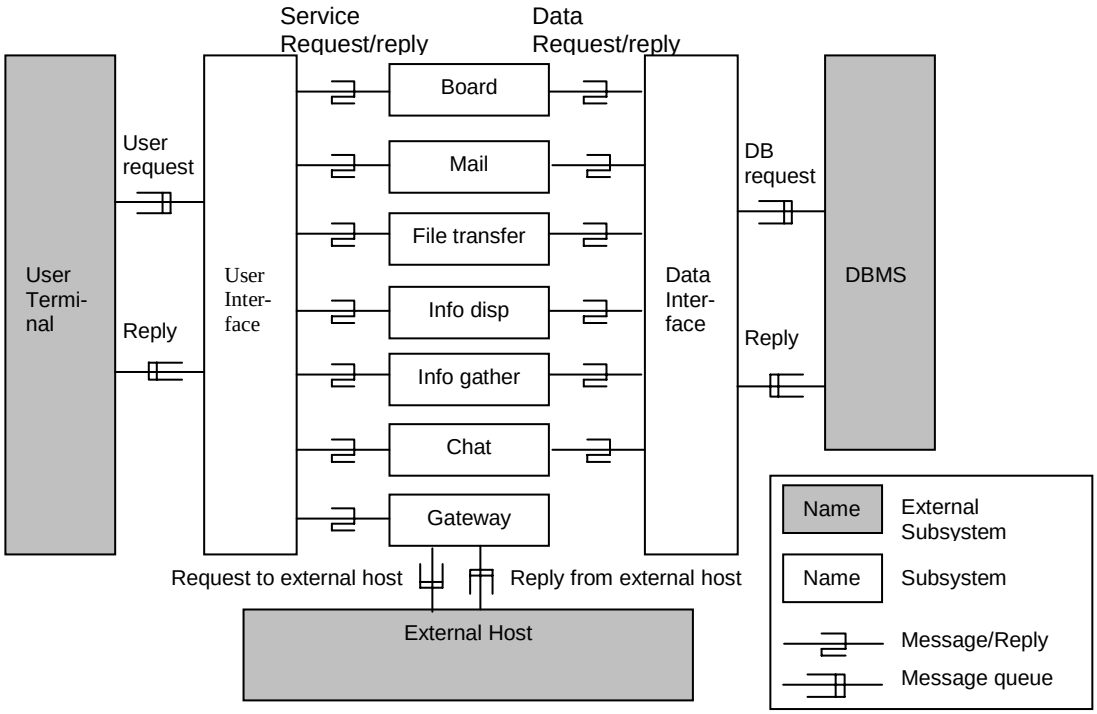

Figure 12. Example of FORM subsystems model [Kang *et al.* 1998].

It is very useful to connect between the artifact space and the feature space so that their mappings become more explicit. In the case of Figure 12 parallel to the subsystems model there should be a corresponding feature model. The feature model should have explicit representation of the set of features that are the basis for the subsystem model. In this case Board, Mail, File transfer, etc. are subsystems that map to top-level features (or sets of features).

The process level represents the dynamic behavior of each subsystem. The behavior of a subsystem is described in terms of processes. Mainly what it does is allocate operational features of the capability feature (service feature) assigned to the subsystem to its processes. We can say that each process of the subsystem will 'implement' operational features of the capability feature of the subsystem.

Operational features should be allocated to processes according to the notions of cohesion and separation with regard to localization of data, localization of control and execution frequency. Processes are also categorized as resident or transient with respect to their duration of activation, and multiple and simple with the respect to whether they can be forked. Figure 13 presents one example of a process model.

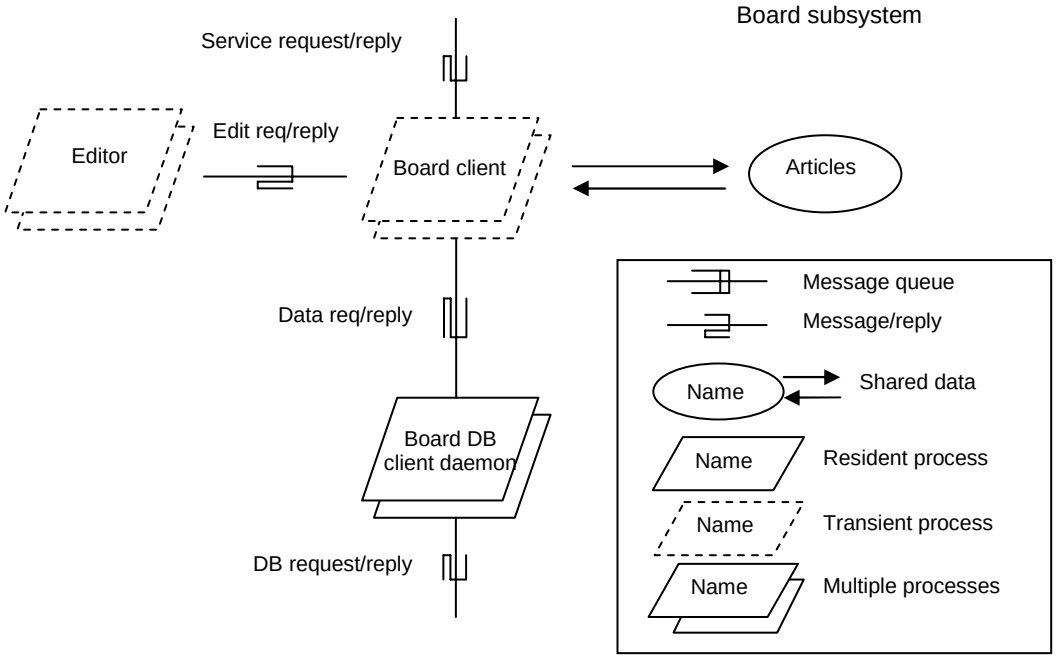

Figure 13. Example of FORM process model [Kang *et al.* 1998].

The lower level of the architecture model is the module level. This level is developed using features at all levels of the feature model.

In FORM modules are in fact the software components to be reused. Every process from the process model should be described in terms of modules. As modules reflect 'selection' of features at different levels we can say that they normally satisfy a set of features. In fact a module contains an abstract specification that satisfy the features. So there can be several concrete software components that match the specification of one module. The implementation of a module can be done in several ways according to diverse reuse strategies. For instance, there can be pre-coded components or parameterized template components of even skeleton code components that need to be completed.

Figure 14 shows an example of how one module model looks like. In this example the model reflects the modules used to implement the Board process. As already stated parallel to the model there is always a feature model showing the select features. In the case of a module model there should be, normally, features of several levels (capability, operating environment, domain technologies and implementation techniques).

Unfortunately, to our knowledge, the FORM method doesn't add much more light into the process of design and implementation of reusable modules, i.e., components and the composition of them.

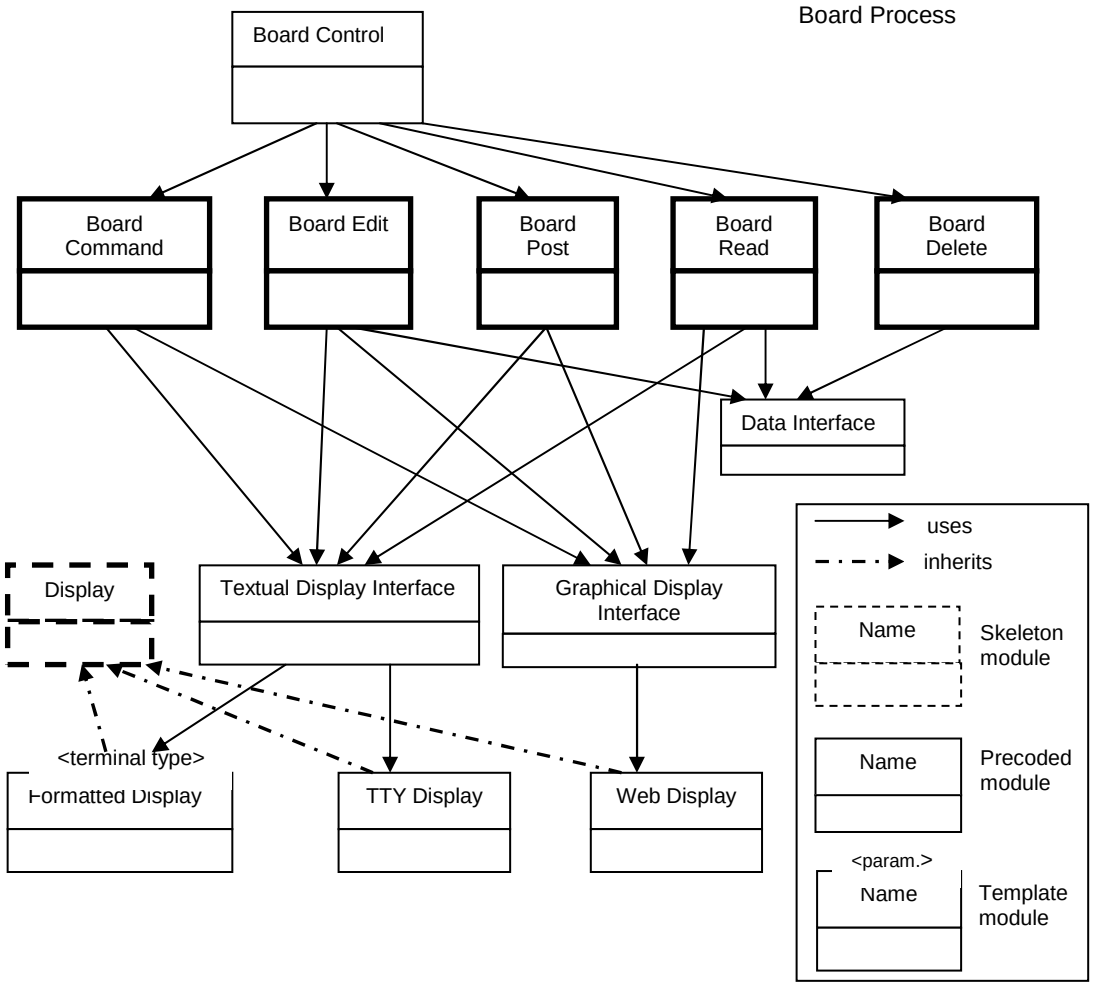

Figure 14. Example of FORM module model [Kang *et al*. 1998].

The Object Connection Architecture proposes a method to realize concrete designs and software components based on domain models as the ones resulting from Feature-Oriented Domain Analysis. It also relies on the notion of subsystem as the first level of partition of a system. Subsystems interact with witch other by means of *imports* and *exports*. We can say that imports and exports define the interface of the subsystems. Objects represent the behavior (and possible state) of real-world or virtual entities. Subsystems are composed of objects. A subsystem uses a *controller* to coordinate the activities of its composing objects. At a higher level there is the notion of an *executive* that coordinates the subsystems. *Surrogates* are a concept that OCA uses to represent logical or physical devices that interface with the system. The concept of *signature* is used to represent the interface of OCA components (objects, subsystems and surrogates). To support a separation between an object and its actual implementation each object has a *manager*, witch is basically a mediator between the object implementation and the clients of the object. The objective is to achieve a higher degree of independence from the implementation.

OCA describes the process of mapping from domain models to software components. Let's take for instance the case of the object concept. Objects are one of the more important parts of an OCA architecture because they represent the base functionality. They are the building blocks of subsystems witch in turn compose the architecture of a system. As mentioned, objects represent

entities, so they can be identified based on diagrams such the entity-relationship diagram of Feature-Oriented Domain Analysis. Objects that need to be represented are identified based on the selected features. For instance, if we don't select any feature related to an entity of the domain we don't need to define an object for that entity. According to OCA one way to discover the operations of an object is from the possible features of the object identified in the feature diagram. Other source can be the functional model. The possible composition of features can also guide the object definition. For instance, all mandatory features (descendent from a selected feature) need to be implemented in the object. Alternative features can be supported with different implementations of the object.

In an OCA architecture objects implement features. Objects can them be combined into subsystems. A system is then composed of subsystems. This composition concept is expanded in GenVoca.

Genvoca is a method that synthesizes software systems by composing components from reuse libraries. The GenVoca base approach to building software systems has some similarities with Draco because they both are transformation systems. This means they both rely on transformation mechanisms to produce the software system.

The GenVoca method is based on Genesis [Batory *et al.* 1988], a database management system generator and Avoca [O'Malley and Peterson 1992], a generator in the domain of network protocols. The similarities of Genesis and Avoca led to the appearance of GenVoca [Batory and O'Malley 1992].

The development of software using GenVoca is based on the composition of components thru layers of data types. These layers of data types represent features of the component. In this manner layer compositions can be described by expressions. Here is an example of a GenVoca expression for a component:

```
bag[concurrent[sizeof[unbounded[managed[heap]]]]]
```

This expression is defining a collection component with diverse features. A data type layer represents each feature. Each layer contributes to the component composition with classes, attributes and methods that implement the corresponding feature (or features). By stacking the different layers we can combine the different features that the component needs.

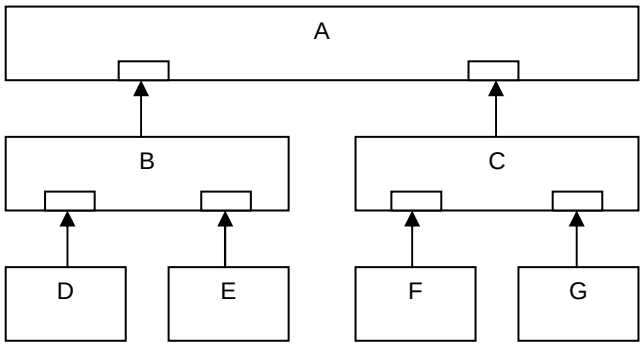

Figure 15. Tree representation of GenVoca expression A[B[D, E], C[F, G]] [Czarnecki 1998].

This combination of layers in order to achieve one certain behavior has some similarities with the inheritance mechanism of the object-oriented model. In the object-oriented model inheritance is used to add new behavior to a class by specializing its original features. We can see GenVoca layers in a manner similar to inheritance but with a larger scope. In fact, GenVoca layers are not confined to work with only one class; a typical layer is a combination of several classes. As we will see later there are also other differences from the inheritance mechanism.

Layers can be viewed as parameterized components. For instance, in the previous example, the sizeof layer has the unbound layer as parameter witch in turn has the *managed* layer as parameter. So, layers can (and normally) have other layers as parameters. In the previous example all but the heap layer have other layers as parameter.

In general a GenVoca layer can have more than one parameter. Lets take for instance the following expression: A[B[D, E], C[F, G]]. If we make a visual representation of this expression we get a tree, like the one in Figure 15. As we can see layers D and E became parameters of layer B witch is a parameter of layer A. Layer A has also C as a second parameter. C is composed of layers F and G.

In order for the GenVoca composition of layers to work we need to specify how the layers can be composed. For that, for each layer it's necessary to specify the interfaces the layer expects as parameters and the interfaces it exports. These interfaces are called *realms*. A realm is basically a collection of class and method signature declarations. A layer belongs to a realm if it exports all the classes and methods declared in the realm (the layer can export more classes and methods than the ones declared in the realm).

Using this concept of realm, we can express, for instance, the following GenVoca domain model:

```
R={A[x:S, y:T]}
S={B[x:U, y:V]}
T={C[x:U, y:X]}
U={D, F}
V={E}
X={G}
```

In this model R, S, T, U, V and X are realms. A, B, C, D, E, F and G are layers. The expressions state what realms are exported and imported from the layers. The first expression states that the realm R is exported from layer A witch imports as parameters realms S and T. The model also states that realm U is exported from the layers D and F witch do not import any realms. Figure 15 represents a valid layer composition according to this model.

In addition to realm parameters a layer can have also constant and type parameters. Realm parameters are called vertical parameters because they affect the all hierarchy of layers. Constant and type parameters are called horizontal parameters because they only affect the layer in witch they are parameters.

In order to better understand the concepts we will present an example of a GenVoca component composition based on [Czarnecki 1998]. This example uses an extension to C++ called P++. P++ extends C++ with new language features that enable the specification of the GenVoca modeling concepts [Singhal and Batory 1993].

Figure 16 presents an example of how realms can be declared in P++. The figure presents two realm declarations: one is a declaration of a 'simple' realm and the other is a declaration of a realm based on another realm (by means of inheritance). The first realm (DS) defines the interface to a

data structure. As we can see, in this example, two classes define the data structure: a container class and a cursor class. In a certain way, the realm declaration is similar to a C++ template declaration. In the case of the DS realm the data type of the elements of the data structure is defined by the e template parameter. The second realm (DS_size) is based on the first. The realm DS_size is a subrealm of DS, so it inherits all the classes and methods of DS. As we can see it also adds the method read_size() to the container class.

```
template <class e>
realm DS
{
   class container
   {
      container();
      bool is_full();
      ... // other operations
   };

   class cursor
   {
      cursor (container *c);
      void advance();
      void insert(e *obj);
      void remove();
      ... // other operations
   };
};

template <class e>
realm DS_size : DS<e>
{
   class container { int read_size(); };
};
```

Figure 16. Example of realm declaration [Czarnecki 1998].

If we go back to the GenVoca expression presented in the previous section we can say that the DS realm could be exported by the unbounded layer and imported (parameter of) by the size_of layer. The size_of layer would export the DS_size realm. In turn, the concurrent layer could import and export the DS_size realm. This makes the concurrent a symmetric layer because it imports and exports the same realm.

Figure 17 presents the possible implementation of the layers size_of and concurrent. To be noted that because a layer is in fact a software component it is named that way in P++. The definition of a component starts by specifying what are its parameters. In the case of the size_of component it has two parameters: one type parameter (class e) and one realm parameter (DS<e> x). After the name of the parameter it is specified the exported realm(s). In the case of the size_of component the exported realm is DS_size<e>.

```
template <class e, DS<e> x>
component size_of: DS_size<e>
{
 class container
 {
  friend class cursor;
  x::container lower;
  int count;

  container() { count = 0; };

  int read_size() { return count; };

  bypass_type bypass(bypass_args)
  {
   return lower.bypass(bypass_args);
  };
 };

 class cursor
 {
  x::cursor *lower;
  container *c;

  cursor (container *k)
  {
   c = k;
   lower =
     new x::cursor(&(c->lower));
  };

  e* insert (e *element)
  {
   c->count++;
   return lower->insert(element);
  };

  void remove()
  {
   c->count--;
   lower->remove();
  };

  bypass_type bypass(bypass_args)
  {
   return
     lower->bypass(bypass_args);
  };
 };
};
```

```
template <class e, DS_size <e> x>
component concurrent: DS_size <e>
{
 class container
 {
  friend class cursor;
  x::container lower;
  semaphore sem;

  container() { };

  bypass_type bypass(bypass_args)
  {
   bypass_type tmp;
   sem.wait();
   tmp = lower.bypass(bypass_args);
   sem.signal();
   return tmp;
  };
 };

 class cursor
 {
  x::cursor *lower;
  container *c;

  cursor (container *k)
  {
   c = k;
   lower =
     new x::cursor(&(c->lower));
  };

  bypass_type bypass(bypass_args)
  {
   bypass_type tmp;
   sem.wait ();
   tmp = lower->bypass(bypass_args);
   sem.signal();
   return tmp;
  };
 };
};
```

Figure 17. Implementation of layers sizeof and concurrent [Czarnecki 1998].

The size_of component implementation is coherent with the fact that it exports the DS_size realm. Because it has to add the size feature to a data structure (the DS realm) it implements the

read_size() method of the container class and also defines the constructor (in order to initialize to zero the counter of the data structure elements). Another aspect of the implementation to be noted is the access to the parameters of the component. In the case of the sizeof component the parameter DS<e> is named x. This name of the parameter is used, for instance, in the declaration x::container lower. In this way the class container of the sizeof component can access the container implementation of the imported realm. The lower variable is used in the method bypass() of the container class. This method is a special method of the P++ language.

All the methods declared in the exported realm and not explicitly implemented in a component are implicitly defined by the bypass construct. In the case of the size_of container class what the bypass() does is redirect all other method invocations to the lower container class (bypass_args and bypass_type match, respectively, the method arguments and return type). The bypass() method is used in a different way by the concurrent component. It uses the bypass() method to wrap all the methods with invocations to a semaphore wait() and signal() statements in order to implement its behavior.

```
template <class e, int size>
component bounded : DS<e>
{
 class container
 {
  e objs[size];
  ... // other methods and attributes
 };

 class cursor
 {
  int index;
  ... // other methods and attributes
 };
};
```

Figure 18. Implementation of the layer bounded [Czarnecki 1998].

Components in GenVoca can also be specified without realms as parameters. This is the case of the bounded component presented in Figure 18. The bounded component has two parameters: a data type and an integer. The integer is used to initialize the number of elements of the bounded container. The type parameter specifies the type of the elements of the container. The bounded component exports the realm DS<e> so it could be a parameter of the component size_of.

Figure 19 presents an example of the successive implementations of read_size() after each composition layer. The marked text represents the successive changes in each component composition.

The GenVoca method presents a real possibility of composing software components through layers of types, i.e., other software components. This mechanism is similar to inheritance but more generic, since the components can be composed of multiple types. If we think of GenVoca *realms* as features we can say that a component implements features (the exported *realms*) and needs implemented features from other components. We can say that the GenVoca method brings closer

the problem space and the solution space. The mapping between features and implementing components is then more easily done.

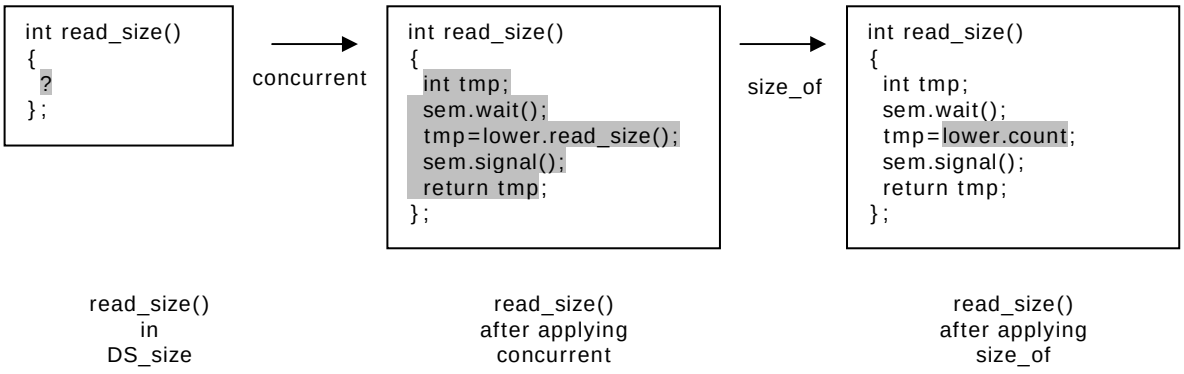

Figure 19. Successive implementations of read_size() [Czarnecki 1998].

GenVoca is also a transformation system. As depicted in Figure 19, functionalities of the components are transformed during their composition. This is similar to what happens also in the Draco system.

In Draco, a program specified in some domain is parsed by the domain parser into an internal form. The program in this internal form is then transformed and refined. Draco uses also source-to-source transformations techniques in order, for instance, to remove inefficiencies in the domain. The refinement will result in the internal form been specified in fragments of many other domains using software components. This process should continue until the resulting refinements lead into an executable domain.

A problem in the domain A is then specified in the external form of the syntax of the domain A, i.e., ext[A]. The parser for the domain A will transform that representation into the syntax of the internal form, i.e., int[A]. All the transformations in Draco are made in the internal form. The language prettyprinter of a domain should be able to transform an internal representation of the domain into an external representation. The source-to-source transformations mentioned transform internal representations of one domain into internal representations of the same domain (int[A] into int[A]).

In Draco, the software components specify the semantics of the objects and operations in a domain. The meaning (semantics) of software components is specified in terms of other domains in Draco. The specification of a domain component is based on relating the internal form of the domain to the internal form of other domain domains (or the same domain), i.e., int[A] to int [A,B,…,Z]. There is a component for each object and operation in a domain. Each component contains many refinements each of which is a possible refinement for the object or operation in the domain (which the component represents).

The Draco approach of using several domain languages, witch one aimed at specifying some particular aspect of a program and using a mechanism of transforming program specifications from one domain to another until the specification is expressed in an executable domain its simple but powerful. This approach aims at closing the gap between the problem space and the solution space. There will be some high level domain-specific language in witch domain experts and end-users can specify their domain logic. That is also the aim of using features and feature diagrams in domain engineering.

The Draco method influenced many research projects. We find of particular interest the work of Simonyi on intentional programming [Simonyi 1995], also a transformation-based programming environment. Intentional programming was a research project from Microsoft that was leaded by Simonyi. The main idea is that programming tools and technologies should represent the intentions of the programmer's abstractions without being confined to one language at a time.

This view of programming is based on the fact that real problems require abstractions from different domains. Therefore, only one domain-specific language normally is not enough. In order to achieve this goal in intentional programming, new abstractions are added as needed. This new abstractions are expressed in terms of abstractions already in the system. Another particularity of intentional programming is that programs are entered directly using commands. The traditional approach to programming based on source code does not apply to intentional programming. In reality programs are entered directly as abstract syntax trees. The only need for traditional parsers is for importing source code already written in legacy languages (C++, Java, RPG, etc.) into the system.

Because programs are represented using abstract syntax trees all the aspects of development are also based on this representation. As such, nodes of the abstract syntax tree have methods (default or user defined) for all aspects of development regarding the abstraction that is represented in the node. The methods of a node specify its behavior in development aspects such as editing, displaying, optimizing, generating code and debugging.

Intentional programming as presented is a very advanced programming system. To our knowledge the system did not pass a prototype stage of development even if documented results were promising. Nevertheless the project seems to have influenced recent commercial products from Microsoft and also some of its recent documented research. In fact, some characteristics of the .Net framework [.Net Framework] seem very close to intentional programming. For instance, the .Net framework support for multiple programming languages; extending metadata about programs using attribute programming; reflexive programming and the set of classes in the namespace CodeDOM that support the representation of the logical structure of source code at run-time, independent of language syntax. In the research area intentional programming seems to have influenced other projects like, for instance, the Phoenix project [Phoenix].

In this section we present some of the main methods, techniques and technologies used for implementing variability in software. We did not detail all the possibilities for implementing run-time variability. For instance, we did not describe the technique we think is the most used: pre-processing compiler directives for optional code compiling (in C/C++ languages the `#if` directive). Our focus was on variability implementation mechanisms with a high degree of manageability and productivity that are as close possibly to the domain language. This is the case of the GenVoca composition and transformation mechanism. The major limitation of this variability implementation mechanism is that they depend on selection and composition at build time. As mentioned in section 1 this is not sufficient for very high dynamic run-time situations. In theses cases we must deal with variability at run-time. In our work we propose that one way to achieve this is by using domain-specific languages at run-time.

For variability to be solved at run-time it is necessary the intervention of the application end-user. The end-user intervention can be as simple as a run-time feature selection, for instance, by means of an option selection in a configuration screen of the application. There can be also more complex scenarios, possible in very dynamic domains. In these cases it may be necessary for the end-user to complete the functional specification of application's features at run-time. This functional

completion ought to be made, if possible, by using a language as close to the domain language as possible. This requirement will make possible for an end-user with domain and system (application) knowledge to complete the functional specification of dynamic features. Clearly the end-user who is going to complete the application's functionality should also be a domain expert. As we will see in the next section, domain-specific languages are tools that potentially can support these requirements.

2.2 Domain-Specific Languages

Although several authors give their one definition of domain-specific language (DSL), there is a great degree of communality between them. One could agree that the term domain-specific language is almost self defined and does not need further explanation. Nevertheless is essential to add that the term DSL regards a programming language and that it syntax and semantics are specialized for a particular application domain or type of problem [Hudak 1998; Thibault 1998]. Given this definition several well known programming languages can be classified as domain specific. We could say that SQL is a DSL for the domain of querying and manipulating relational databases or that HTML is a DSL for the domain of constructing hyper linked digital documents. Following this line of rationing it is also acceptable to say that DSL are programming languages that sacrifice generality in order to achieve greater proximity with some specific problem domain [UTCAT].

Several times there is a natural tendency to relate the terms abstract and specific as if they represented opposite extremes of some dimension. The possibly range of values of such dimension would be from a value that represent something specific to one that would represent something at the highest abstraction level. Sometimes this view comes from some textbooks on object-oriented model that don't precisely clarify the terms. In fact these two terms are distinct and orthogonal. The term specific means something relating to one thing and not others; something particular. The term abstract means something existing as an idea, feeling or quality, not as a material object [Cambridge]. Thus it is possible for something to be specific (or particular) of some area and at the same time with a high degree of abstraction. In the case of programming languages one can classify them in these two orthogonal dimensions: specialization and abstraction.

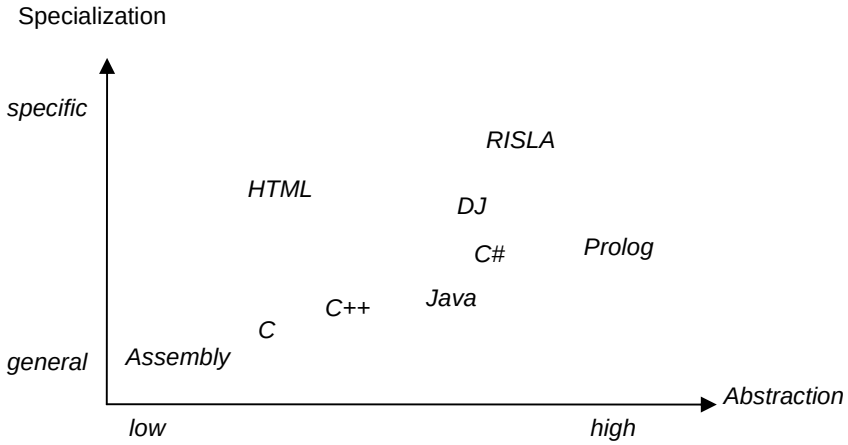

Figure 20. Dimensions of programming language classification.

As an example, the assembly language has a low degree of abstraction, since the programmer works with very low level constructs and can also be classified as having almost none specificity since the language constructs are also very generic, enabling the development of programs for very diverse areas. We can say that the assembly language is a generic programming language (GPL) (as opposed to specific).

The abstraction concept is different. As the former definition refers, an abstraction is something intangible, as an idea or feeling. In computer languages, abstractions are normally used to model or represent real objects. As such we can have very distinct levels for different abstractions of a given object. A very high level abstraction has fewer details. Lower level abstractions have more details.

As we raise the level of abstraction of some object we are underestimate elements, or features, that regard the real object but don't mater in that level of abstraction and in a given context of analysis. In the programming language area this usually (but not always) means that a programming language with a higher level of abstraction permit that the programmer write less code in order to do some operation than in a language with a lower level of abstraction. This also means that the programming language implicitly will do more work as it level of abstraction grows. Confusing regarding the terms specific and abstract rises from the fact that normally as the level of abstraction of one programming language rises, the language also narrows his scope, and so becomes more specific. But this doesn't happen always, as we can see in Figure 20.

The base constructs of programming languages are abstractions. Procedural languages, for example, have abstractions for procedures and basic data types. Some programming languages allow the definition of new abstractions by the programmer. In object oriented languages the class abstraction is used to enable the definition of new abstractions by the programmer [Thibault 1998].

Normally a GPL has some base generic abstractions that can be used in several domains of application of the programming language. The necessity of more specific abstractions can be solved if the GPL has constructs that enable the definition of new abstractions. A GPL with this characteristic can be used to define abstractions for specific domains. That’s why some domain-specific languages are based on GPLs with class libraries containing the abstractions specific to the domain. This is one of the ways one can build DSLs [Hudak 1996].

DSLs need to give the programmer an abstraction level appropriated to the given domain. The abstractions of the DSL should be very close to the abstractions the domain expert is used to work and think of. The constructs and syntax of the language should also be close to the technical/specific language of the domain expert. As DSLs should provide the domain abstractions it is possible that some DSLs do not supply means for creating new abstractions [Thibault 1998]. The reason is that the degree of specificity of some DSL is so great that all abstractions necessary to that very specific domain are pre-build in the language. In this case, if the DSL supported the means of defining new abstractions it would loose it specific character and become a more generic language (but it level of abstraction would be the same). DSL apart from being specific tend to have high levels of abstraction.

2.2.1 Classification of Domain-Specific Languages

As Czarnecki argues, DSLs can be further classified as general modeling languages or application-oriented languages [Czarnecki 1998]. A DSL can be specific in a system wide domain, for example, a synchronization language like COOL [Lopes 1997]. This language could be applied to several different application domains with synchronization needs. On the other end, a DSL can be specific to a given application oriented domain, such as a language for the specification of

financial products. That is the case of RISLA [Deursen *et al.* 1996]. So, a DSL, apart from having some degree of specialization and abstraction can also be further classified in a third dimension of application oriented features (vertical applicability vs. horizontal applicability), or scope of the language.

System wide, or horizontal languages, also normally encompass only one system part or aspect (like synchronization). Application specific or vertical languages encompass aspects of a given vertical domain and therefore cannot be used outside that vertical domain. This is the case of RISLA. This third dimension in language classification is presented in Figure 21.

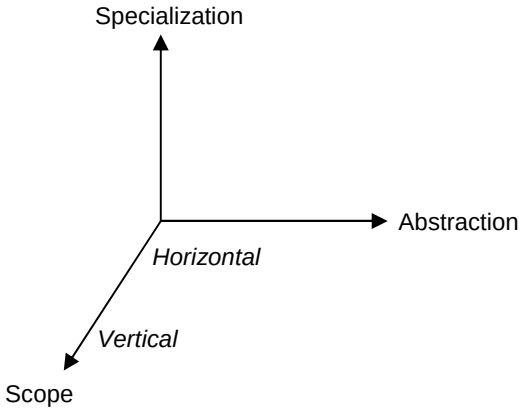

Figure 21. A third dimension of programming language classification.

Horizontal Scoped DSLs

A horizontal scoped DSL is a language that has a broader or system wide scope. The language, although specific, can be applied to diverse application domains. Examples of such horizontal scoped domains are synchronization, distribution and exception handling.

One such example is D, a horizontal domain-specific language aimed at programming coordination and distribution aspects of applications [Lopes 1997]. These are also called programming aspects of an application. In fact, D enables that the programmer specify horizontal aspects, or concerns of the application, in isolated modules that are then combined to form the resulting program. Aspect oriented program is the technique in witch D is based on [Kiczales *et al.* 1997].

In D, the programmer can separate the distribution and synchronization concerns from the program. Normally these concerns would be coded in a mixed way with the main functional aspects of the program. This will normally result in what is called tangled code. Tangled code has a mix of several aspects, and as such, is very difficult to identify and maintain the various aspects of the program. All aspects of the program are mixed in the generic programming language that is used to code the program. The aspect oriented programming technique is based on the use of a DSL to program horizontal or system wide aspects of the system, removing this code from the GPL program. On program compilation the DSL is combined with the GPL for the generation of the resulting program. Normally the syntax of the horizontal DSL is very close to the syntax of the GPL. On generation, the DSL is transformed into the native syntax of the GPL and the code is joined with the GPL code with a technique called code *weaving*. Section 2.3.2 presents an example of the D language and the techniques of aspect oriented programming.

Vertical Scoped DSLs

Vertical scoped DSLs are languages that are specific to a particular application or activity. This kind of DSL normally can only be applied to an application or family of applications. Examples of such vertical scoped domains are insurance, finance or medical domains.

One example of a DSL that has a vertical scope is RISLA. This DSL is used by a bank to describe is interest rate products. The domain expert of the bank can use RISLA to define interest rate products. The RISLA compiler will transform this product definition in COBOL programs that can be integrated in the bank system. This generated COBOL programs can then be used to permit the bank system users to work with the new specified product. The RISLA language and supporting tools is described in section 2.3.3.

2.2.2 DJ Language

D is a horizontal domain-specific language aimed at specifying coordination and distribution aspects of applications. The language itself was designed to work with a *host* language. In working with D, the host language is used to specify the functional aspects of the application. D was design to work with general-purpose languages as host languages for the functional aspects of applications. These general-purpose languages that D was design to work with are also named component languages because they are very well suited for implementing software components.

A software application build with D will have a separation of the coordination and distribution aspects of the program from the functional ones. D covers the coordination and distribution aspects and the host GPL covers the functional aspects. On compilation the three aspects are combined to generate the output program. One of the major advantages of using D is that the programmer can reason about the problem in three independent slices, or aspects: functional, distribution and coordination. D shares his goal with other aspect-oriented languages and with the aspect oriented programming technique. For the specification of the coordination and distribution aspects D has two domain-specific languages: *Cool* for the coordination aspects and *Ridl* for the distribution aspects.

For clarity reason we will present only the domain language used in D for specifying coordination aspects.

The Host Language

D was designed to work with component languages such as Java or C++. The major restriction of the host language is that it must be an object-oriented language. The D language expects the host language to have the *class* construct with the same semantic it has in the mainstream object-oriented languages. D also supports that classes be defined by means of inheritance.

DJ is a concretization of D for use with Java as a host language. The Java hosting language has only one restriction: it doesn't support the `synchronized` keyword. This is because the coordination aspects will use new keywords from the DSL and as such the `synchronized` Java keyword becomes obsolete.

The syntax of the DSL aimed at coordination and distribution is very close to that of the Java language. The objective is that the programmer sees D as an extension to Java, as opposed to a

completely new language that he has to use in order to specify coordination and distribution aspects of the application.

Example

Figure 22 represents an example of application of the DJ language. This example is based on [Lopes 1997] and presents one possibly solution for the bounded buffer problem using DJ for specifying the coordination aspects of the program.

The bounded buffer is basically an object that olds internally a buffer of objects. The bounded buffer object also manages access to the internal buffer. One can insert objects into the buffer (*producers*) and remove objects from the buffer (*consumers*). Producers will wait if the buffer is full and consumers wait if the buffer is empty. Access to the bounded buffer by the producers and consumers can be made concurrently.

The functional aspects of the program are specified in the class `BoundedBuffer`. This code is basically the Java implementation of the bounded buffer without regarding any questions or aspects of concurrency.

The coordination aspects of the problem are specified with the `coordinator` construct. This keyword is part of the DJ DSL for coordination and distribution. Coordinators are used to specify coordination aspects of one or more classes. The connection between the coordinators and the classes is made by name (bind by name).

The syntax of coordinators is very close to the syntax of Java classes. The coordinator has access to the members of the class that it controls. In the coordinator we can specify data members in a manner very similar to the way we do with regular classes. Methods can also be specified. The aim of these methods is to specify coordination aspects, namely suspension and notification of accessing threads, normally based on the state of the data members of the coordinator and possibly the coordinated objects.

The `selfex` keyword specifies that `put` and `take` methods of the coordinated class can only be executed by one thread at a given time. If the coordinator only had this statement it was possible to have two different threads executing the `put` and `take` methods at the same time. But this is not the behavior we want for the coordination aspect of this little program. We do not want `put` and `take` to be execute simultaneously. That's why the coordinator has the `mutex` statement. In DJ we can use the `mutex` keyword to specify mutual exclusion in a set of methods. In this case `put` and `take` cannot be executed simultaneously.

The coordinator of the bounded buffer class also specifies two condition variables: `full` and `empty`. Condition variables are different from ordinary variables since they can be used to control the suspended state of threads. Condition variables can only be of `Boolean` type.

For the control of accessing threads and change of the state of coordinators DJ uses *method managers*. Method managers are associated with methods of the controlled class and can use condition variables as entry conditions. A method manager can have code that is executed on entry of the associated class method and on exit. This code can access values of variables of the coordinator and of the coordinated classes but can only change variables of the coordinator.

```
public class BoundedBuffer
{
   private Object array[];
   private int putPtr = 0, takePtr = 0;
   private int usedSlots=0;

   public BoundedBuffer(int capacity)
   {
      array = new Object[capacity];
   }

   public void put(Object o)
   {
      array[putPtr] = o;
      putPtr = (putPtr + 1) % array.length;
      usedSlots++;
   }

   public Object take()
   {
      Object old = array[takePtr];
      array[takePtr] = null;
      takePtr = (takePtr + 1) % array.length;
      usedSlots--;
      return old;
   }
}

coordinator BoundedBuffer
{
   selfex put, take;
   mutex {put, take};
   cond full = false, empty = true;
   put: requires !full;
   on_exit
   {
      empty = false;
      if (usedSlots == array.length)
         full = true;
   }
   take: requires !empty;
   on_exit
   {
      full = false;
      if (usedSlots == 0) empty = true;
   }
}
```

Figure 22. Bounded Buffer implemented with horizontal DSL DJ [Lopes 1997].

In our example the coordinator class has two method managers relative to the put and take method of the controlled class. Basically these methods update the state of the condition variables full and empty. These variables represent the controlling state. They also guard the execution of the corresponding methods of the coordinated class. In this example the put method can only be

executed by one thread if the condition variable full is false, i.e., we can only add an element to the bounded buffer if it is not full. Similarly we can only remove an element form the bounded buffer if it is not empty.

```
public class BoundedBuffer
{
   private Object[] array;
   private int putPtr = 0, takePtr = 0;
   private int usedSlots = 0;

   public BoundedBuffer (int capacity)
   {
      array = new Object[capacity];
   }

   public synchronized void put(Object o)
   {
      while (usedSlots == array.length)
      {
         try
         {
            wait();
         }
         catch (InterruptedException e) {};
      }
      array[putPtr] = o;
      putPtr = (putPtr + 1) % array.length;

      if (usedSlots++ == 0)
      notifyAll();
   }

   public synchronized Object take()
   {
      while (usedSlots == 0)
      {
         try
         {
            wait();
         }
         catch (InterruptedException e) {};
      }
      Object old = array[takePtr];
      array[takePtr] = null;
      takePtr = (takePtr+1) % array.length;

      if (usedSlots-- == array.length)
      notifyAll();
      return old;
   }
}
```

Figure 23. Bounded Buffer implemented only with the Java GPL [Lopes 1997].

Figure 23 presents the implementation of the bounded buffer without DJ. This implementation is based only in the standard constructs of the general purpose language Java. The code that deals with coordination is marked. This code is intermixed with the functional code of the bounded buffer. This obviously makes the comprehension of the program more difficult. In the DJ version of the program the code is much more 'clean'. The coordination aspect of the program is mixed with the functional aspect. In the DJ version the two aspects are separated and can be developed and maintained with a high degree of independence.

Implementation Details

DJ is an implementation of D for working with the Java GPL. One of the design decisions of the authors of D was the independence between the aspect language and the GPL language [Lopes 1997].

The DJ language is a language based on the Java language with almost all the constructs of the Java language for specifying the functional aspects of a program. The only limitation is the lack of the `synchronized` keyword. For the coordination and distribution aspects of a program the DJ adds new constructs.

The implementation is totally based in the Java language. The compilation of a DJ program basically translates the source code of DJ into Java and then compiles the resulting source with the Java compiler.

The aspects that are independently specified in DJ are combined on compilation. The result is a 'standard' Java program with all the aspects 'merged'. This merging operation is known as *weaving*. The process of compilation of DJ (*weaving*) is presented in Figure 24.

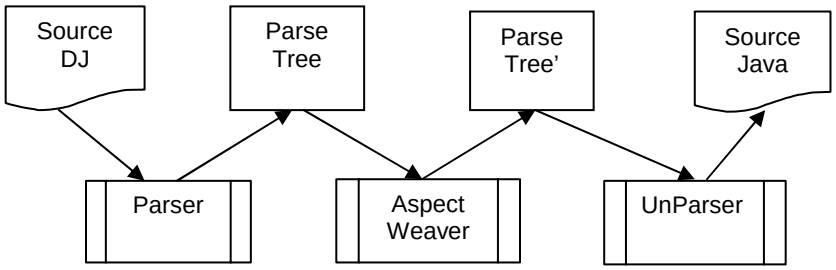

Figure 24. Compilation process of the coordinator aspects in DJ [Lopes 1997].

Each class (functional aspect of program) in DJ will result in a Java class and each coordinator in DJ will result in a Java class with the coordination aspects of the program. The weaving, or merge, of the functional and coordination aspects is done through changes in the functional class in order to call the coordination code. Basically this is achieved as depicted in Figure 25.

Each coordinated class will have a private variable of the correspondent coordinator class. This variable is initialized in the constructor of the coordinator class. If the coordinator functions on per instance 'mode', i.e., there is a coordinator object for each instance of the coordinated class this variable will be initialized with a new instance of the coordinator class. If the coordinator will

coordinate all the instances of the coordinated class then the variable is always initialized with the same instance of the coordinator.

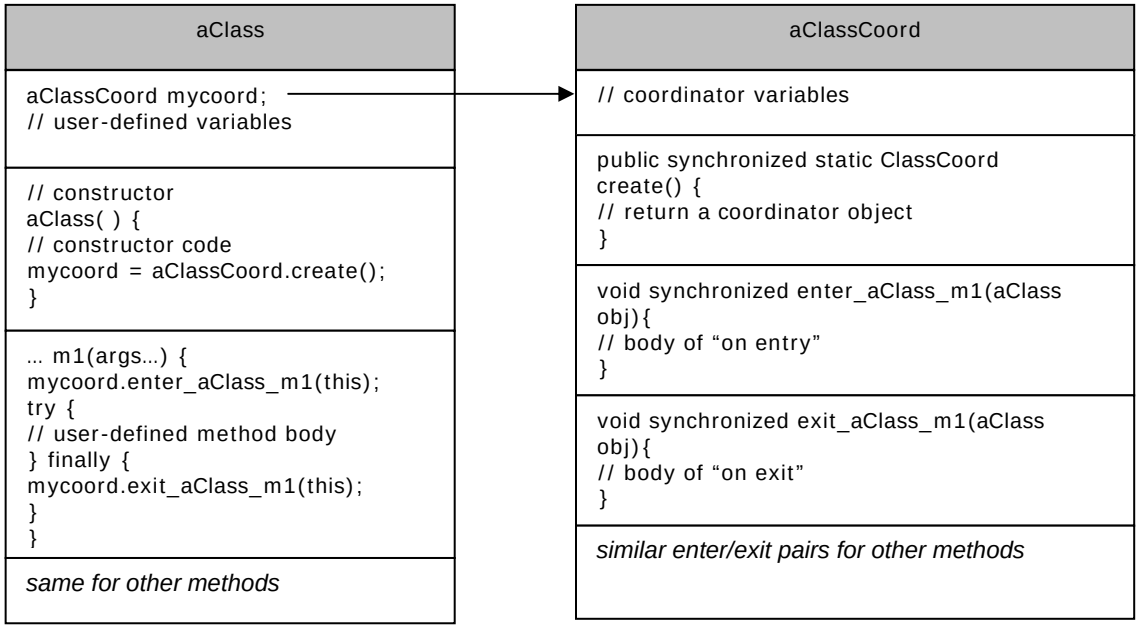

Figure 25. Java code resulting from weaving [Lopes 1997].

The coordinator variable in the coordinated object is used to invoke the coordinator at the coordination points. For each specification of a managed method in the coordinator the corresponding method of the coordinated class will be changed to possible include the on entry and on exit coordination logic. Basically the coordinated class will call an on entry method of the coordinator before executing the functional code and will call an on exit method after executing the functional code. In order to call these methods the coordinated class will use the coordinator variable. The called methods must have at least one parameter that is used to pass the reference of the coordinated class to the coordinator. This will enable the access to the coordinated object by the coordinator.

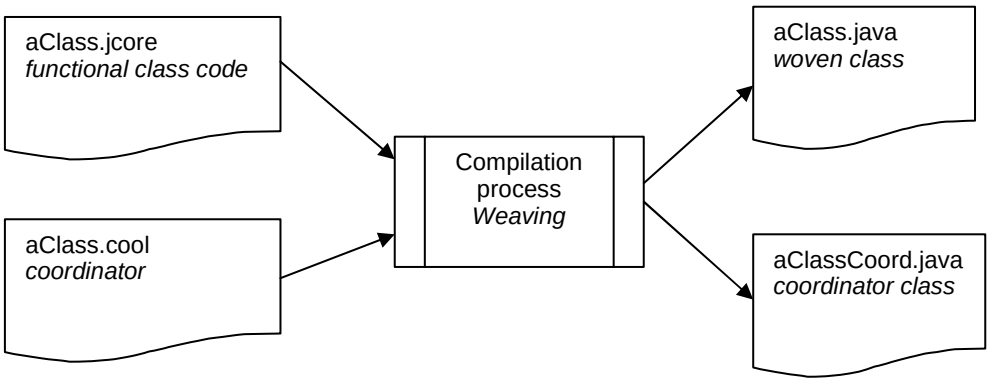

Figure 26. Relationship between DJ source and Java generated source [Lopes 1997]

Figure 25 presents a simplified representation of the relationship between coordinator and coordinated class after the described process of code weaving.

Figure 26 represents the relationship between the input and output source resulting from the weaving proc process.

There are much more details regarding DJ implementation. For legibility we leave out the distribution aspect of the language. We also leave out questions regarding how to deal with class heritance and how it may affect the coordinator aspect. The weaving process also becomes more complex when more than one aspect has to be weaved in one class, as is the case of DJ.

2.2.3 RISLA

RISLA is a language designed to enable the specification and manipulation of interest rate products [Deursen *et al.* 1996]. Mortgages, loans and futures are examples of financial products that can be specified in RISLA. RISLA was initially developed as a project involving a Dutch bank.

One of the major characteristics of finance products is their relative short lifetime as opposed to other industries like manufacturing. Financial enterprises have the need for rapidly create new products because of the also very rapid evolutions of the financial markets.

Many financial institutions have long time ago chosen to implement her software using proprietary platform and business programming languages like COBOL or RPG. This was the case of the referred Dutch bank. The relatively high time to market of new products, because of the need for new COBOL developments or changes in existing programs, lead the bank to start a project witch aimed the resolution of the problem. This was the context of the appearance of the RISLA project. This project also involved the software house CAP Volmac and the Dutch national center for mathematics and computer science CWI [Brand *et al.* 1996].

Implementation Details

The idea behind RISLA is to give the domain expert, i.e., the interest rate product engineer, the capability of describing new products in a language close to his technical language. In this way the creation of new interest rate products wouldn't require programming new software in COBOL.

Basically an interest rate product defines a scheme of agreements between two parts involved in a contract: the bank and the customer. As such, a product functions very much like a template or class for similar contracts. An instance of a product is therefore a contract made by the bank and a specific customer.

The product specification made by the interest rate product engineer using RISLA should be enough for the RISLA compiler to be able to generate all the COBOL programs needed. This includes not only COBOL programs to calculate cash flows for the new product but also programs to inspect and query the new contracts of the product and also programs to enter these new contracts. All this generated programs resulting from the compilation of a RISLA interest rate product specification will be linked and called by several points of the bank software system. The COBOL generation not only regards the easy interface of the programs with the bank system but also the reuse of COBOL routines already existent in the system. The RISLA definition of an interest rate product allows the declaration and reuse of such existing COBOL routines.

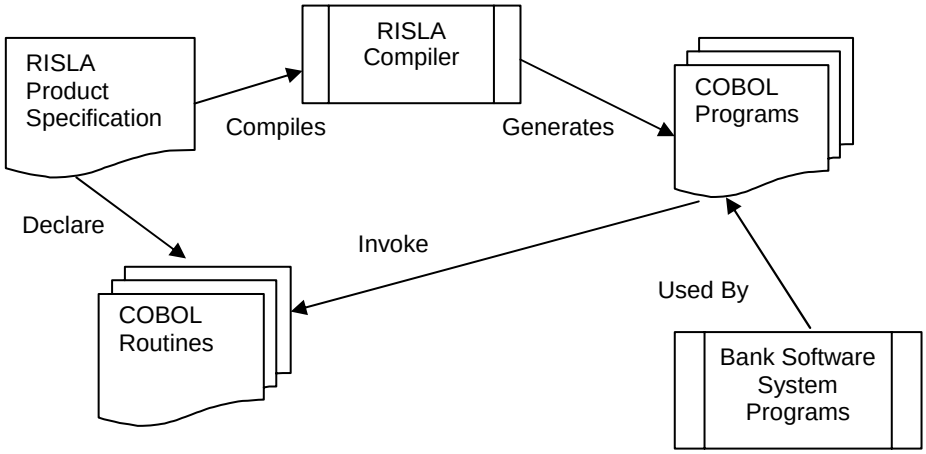

Figure 27. RISLA compilation and execution process.

Apart from the possibility of reusing existing COBOL routines the RISLA team also concluded that some reusing mechanism for RISLA programs was needed. In deed the adoption and generalization of RISLA for the specification of the interest rate products rapidly originated a significant number of RISLA programs with the same low-level computations specified in different products. This lead to the introduction of a reuse mechanism called *Modular RISLA*. Modular RISLA introduces the concept of component. Interest rate products can be specified using such components. A component is like a product; it has contract data and methods. As products can reference components, components can also reference other components. The actual implementation is based on a import/renaming mechanism that translates a modular RISLA program into a *flat* RISLA program. The new composition mechanism is called, in the generation process, before the *original* RISLA compiler, as can be observed in Figure 28.

The RISLA development tools also includes a language called *Risquest* witch aim is to facilitate the specification of new products by the domain experts. Basically this language is used to specify an interactive questionnaire to gather information regarding the new product. The answers to the questionnaire are used to select existing components in order to compose the new product. Eventually the specification of the product has to be completed by hand by the domain expert if the selected components and gathered information is not enough to complete the product definition.

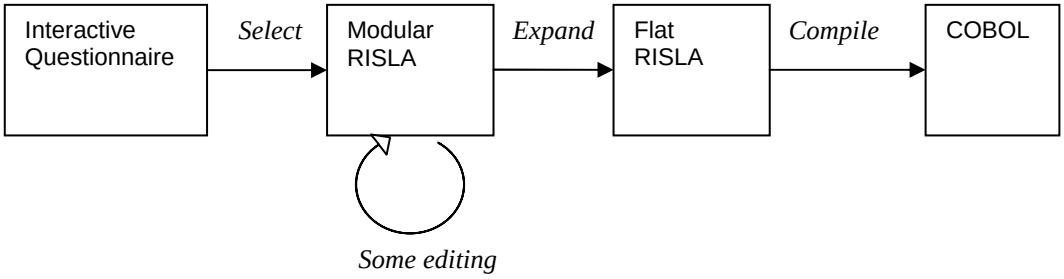

Figure 28. From questionnaire, via Modular and Flat Risla, to COBOL [Deursen *et al* 1996].

The initial implementation of the RISLA compiler was accomplished using C, lex and yacc [Arnold *et al.* 1995]. The RISLA language was formally defined using the algebric specification formalism ASF+SDF [Deursen *et al.* 1996]. The ASF+SDF Meta-Environment tool supports the development of language definitions and generates prototype tools from them. Although RISLA was specified in ASF+SDF for several reasons, explained in [Arnold *et al.* 1995], the initial implementation of the compiler was build with C and the compiler tools lex and yacc.

Example

The utilization of RISLA basically consists in describing interest rate financial products witch basically are series of cash flows [Deursen 1997].

```
product LOAN

declaration

  contract data
    PAMOUNT : amount                    %% Principal Amount
    STARTDATE : date                    %% Starting date
    MATURDATE : date                    %% Maturity data
    INTRATE : int-rate                  %% Interest rate
    RDMLIST := [] : cashflow-list       %% List of redemptions.

  information
    PAF : cashflow-list                 %% Resulting Principal Amount Flow
    IAF : cashflow-list                 %% Resulting Interest Amount Flow

  registration
    RDM(AMOUNT : amount, DATUM : date)      %% Register one redemption.

  local
    FPA(CHFLLIST : cashflow-list) : amount  %% Final Principal Amount
    FRDM : cashflow                         %% Final redemption

  error checks
    "Wrong term dates" in case of STARTDATE >= MATURDATE
    "Negative amount" in case of PAMOUNT < 0.0

implementation

  local
    define FPA as IBD(CHFLLIST, -/-PAMOUNT, MATURDATE)
    define FRDM as <-/-FPA(RDMLIST), MATURDATE>

  information
    define PAF as [<-/-PAMOUNT, STARTDATE>] >> RDMLIST >> [FRDM]
    define IAF as [< -/-CIA( BL(RDMLIST, <-/-PAMOUNT, <STARTDATE, MATURDATE>>),
                        INTRATE ), MATURDATE >]

  registration
    define RDM as
      error checks
        "Date not in interval" in case of (DATUM < STARTDATE) or (DATUM >= MATURDATE)
        "Negative amount" in case of AMOUNT <= 0.0
        "Amount too big" in case of FPA(RDMLIST >> [<AMOUNT, DATUM>]) > 0.0
      RDMLIST := RDMLIST >> [<AMOUNT, DATUM>]
```

Figure 29. Specification of a product loan in RISLA [Arnold *et al* 1995].

The specification of a product therefore contains rules defining how to compute cash flows from initial contract values. A contract is an instance of the product. The contract also represents the customer agreement on the rules of the product. The description of a RISLA product also includes what data is needed to create an instance of a product, i.e., a contract; routines needed to enter data into a contract and possible queries on contract data.

Figure 29 presents an example of a financial product described in RISLA. The product is a loan, one of the simplest interest rate products. Basically a product specification consists of a name, a declaration and an implementation section.

The declaration contains the data necessary to the contract in the form of fields of the contract. Fields must have appropriate data types. As seen in the example, there are several data types, some of them similar to the ones we can encounter in general-purpose languages others domain specific like for instance the `cashflow-list` data type. The information section of the declaration part defines data that can be inspected from the outside of the contract. The registration section can be used to specify routines used to enter data into a contract. In the example the product contains a routine used to register a redemption. The local section is used to specify routines that are used internally, for instance by financial analysts to evaluate the outcome of the contract. There is also a section in the declaration part of the product to specify validations on the contract data.

The implementation part of the product is used to detail how data, routines and queries are computed. There is one implementation section almost for each declaration section. The implementation is also used to link the RISLA product with COBOL routines regarding interest rate products that already exist at the bank and that can be used by the RISLA products.

2.2.4 Development of Domain-Specific Languages

Developing domain-specific languages has major differences from developing general programming languages [Spinellis 2001]. In order to develop a DSL it is necessary to have a very good understanding of the domain. This knowledge of the domain can be captured and represented using the methods and tools described in section 2.1. These methods and tools will enable the capture of the vocabulary and the semantics of the domain. This will provide great help in the design of a DSL.

The context in witch a DSL is developed is very different from the context of development of a GPL. Particularly the resources available for the development of a DSL are very low compared to a GPL. This is because the DSL will also have limited utilization within a domain or program family. Because of this several design patterns appeared aiming at reduce the implementation effort in the development of DSLs [Spinellis 2001]. These strategies in the build of DSLs solve several specific design problems and can be applied to similar situations.

[Deursen *et al.* 2000] describes the typical DSL development cycle as:

Analysis

(1) Identify the problem domain.
(2) Gather all the relevant knowledge regarding the domain.
(3) Aggregate the knowledge in a group of semantic notions and operations on them.
(4) Design a DSL that can describe with precision applications in the domain.

Implementation

(5) Build a library that implements the semantic notions.

(6) Design and implement a compiler that translates DSL programs into a sequence of calls to the library.

Utilization

(7) Write programs in the DSL for all the needed domain applications and compile them.

As we have already mentioned the analysis phase can be based in a domain engineering method, such as the ones presented in section 2.1. These methods provide good tools for gathering knowledge about a domain. This knowledge can then be used to define the necessary language constructs. This is not a critical phase because it should be done even if we are not going to build a DSL for the domain. The major problem regarding a DSL lies in the costs of the implementation phase.

The costs of the implementation phase of a DSL can be very high. In step 6 of the development cycle of a DSL we see that there is the need to implement a compiler to translate DSL programs to sequences of calls to the library of semantic notions developed at step 6. The development of the library should require a greater effort because it can follow a normal process of creating a library of functions using probably a well-known GPL. The problem lies on the effort needed to develop a compiler for the DSL.

A classical approximation to this problem is to build an interpreter or compiler in a similar way to classical general-purpose language compiler development. Standard tools for compiler development can be used like *lex* and *yacc* [Aho *et al.* 1986; Bentley 1986]. There is also the possibility of using tools specialized in the development of DSLs like Draco [Neighbors 1980] or the ASF+SDF that was used in the development of the RISLA language [Deursen *et al.* 1996].

This strategy into the implementations of DSLs has a high cost but also has major advantages. The compiler is designed for the specific needs of the language. Optimizations of generated code can be easily introduced and program validation and error detection is very accurate. The problem is that apart form the compiler other tools may be needed. For instance, a debugging tool is almost inevitable.

Another approach into the implementation of a DSL is to use some base language. In this approach the DSL programs are translated into some GPL. The implementation effort is reduced to the building of a translator that translates programs from the DSL into a GPL. The program resulting from the transformation is then compiled using the GPL compiler. The GPL helper tools, like the debugger, can be used and there is no need to build helper tools. We can view the DSL translator as a preprocessor of the GPL. This approach has limited costs but also has many disadvantages. For instance, errors are only normally detected at the GPL level. This means that it can very difficult to identify the origin of errors at the DSL level. Even if the GPL tools can be used it may be very difficult, for instance, to debug as DSL program, because the debugger will use the GPL concepts and notations and not the ones from the DSL.

There is another major approximation to the implementation of DSLs. This approximation is based on the concept of domain specific embedded language [Hudak 1996]. In this approximation the constructs of the base language are used to implement the constructs of the DSL. The base language has to have features that enable the definition of new functions and operators. That way, it is possible to embed in the base language the functions and operators of the DSL. What this approximation does is extend the base language with the constructs of the DSL. As such in this approach it is not necessary to translate the DSL into a GPL. The base language (GPL) is extended with the DSL. The GPL tools can be used without problems and errors are detected at the right

level. One characteristic of this approach is that the programmer can mix DSL constructs with GPL constructs. This can be a limitation or advantage, depending on the perspective. The major problem we see with this approach is that the characteristics of the base GPL used can limit the design of the DSL.

2.2.5 Adoption of Domain-Specific Languages

By reducing the conceptual distance between the problem space and the language used to express the problem when using a DSL, programming becomes more simple, easy and trustful [UTCAT]. The quantity of code that is necessary to write is reduced, increasing productivity and decreasing maintaining costs. The adoption of a DSL based approach to software engineering involves opportunities but also risks [Deursen *et al.* 2000]. The design of a DSL needs to have a good balance between the advantages and the possible risks or costs.

With DSLs, software solutions can be expressed using terminology and an abstraction level very close to the problem domain. As such, domain experts can understand, validate, change and, eventually, develop programs using the DSL. This is a major advantage of adopting a domain-specific language. As we saw in the RISLA example programs become more expressive from the viewpoint of end-users (i.e., domain experts). They will be able to more easily understand programs. This factor will increase the maintainability of programs because end-users will also be able to evolve programs [Kieburtz *et al.* 1996].

Being more close to the problem space programs become also a very good tool to document and preserve domain knowledge. Domain knowledge is conserved and reused more easily [Deursen *et al.* 2000].

The high-level of abstraction of a DSL can also improve the testability of programs. With DSLs it is possible for a program to be validated without transforming the high level specification into a low level executable language. For instance, it is possible for a RISLA program to be validated without transforming it to COBOL. This also means that program optimizations can be done at a domain level [Basu *et al.* 1997]. This can mean better results from optimization compared to optimization made at low-level where domain knowledge is probably lost. High-level domain-specific languages can also improve the portability of programs [Herndon and Berzins 1988].

Apart from the positive aspects in adopting a DSL, it is also possible to identify some points that need careful consideration. One of the most important problems with the adoption DSLs is that it normally means it has to be built. This is not a problem when programming with a GPL. The advantages of a DSL must be greater then the cost of development and maintenance of the DSL. In this evaluation one should not forget that there are characteristics of GPLs that we normally will also want to have in a DSL, for instance, debugging facilities. The advantages of a DSL can be greater than the cost if, for instance, the DSL is to be used in a product line or family of products. In this case the cost of developing a DSL can be ‘distributed’ by all the applications in the domain.

One problem that can also raise the adoption of a DSL has to do with its users. If the users of the language are the end-users or domain experts and the language was designed with careful decisions regarding the users suggestions then it is very provable that the users will fill comfortable with the adoption of the DSL. On the other hand, if the users are also programmers and are used to programming with GPLs then care must be taken into the design of the DSL. Users will have to be trained to use the new language. The users initial reaction could be negative. The designer of the DSL must take into account these factors. Nevertheless there must be a balance in the DSL

constructs. We should not forget that even if the SQL language is very different than a mainstream GPL, SQL have been adopted in general by the programmer's community as the database language. In the case of SQL, programmers frequently embed SQL statements in their GPL programs.

Another risk with the adoption of DSLs is the lake of performance. This risk is very high and can induce a rejection behavior in the end-users if they know that to implement the same functionality a GPL would perform better. The DSL should be optimized for a performance level near the one of the GPLs. This issue is not so important if the level of productivity gain derived from the high-level of abstraction of the DSL does compensate the lower performance. All these issues must be taken into account when developing a DSL.

In this section we saw that DSLs are great tools at several perspectives of software engineering. They enable high levels of abstraction in software engineering. For instance, we saw how a *horizontal* domain-specific language can be used to specify coordination aspects of an application. We also presented an example of a *vertical* domain-specific language called RISLA. This is a very interesting example of an approach to post-deployment functional completion of an application based on domain-specific languages.

3 Proposed Approach

The focus of this thesis is on how to cope with software applications with intensive feature variability. The Insurance domain was given as an example in section 1.

As we saw from section 2, feature modeling is a widely adopt tool to specify feature variability (and commonality) in domain engineering. Because features (at least capability features) are concepts normally very close to the end-user language, domain analysis based on feature modeling has many benefits. That's why Feature Oriented Domain Analysis and other methodologies use capability features as a means to do domain analysis and modeling. As features are specified using the end-user language they are a very good tool for domain knowledge interchange between end-users and domain analysts and in general developers. With the right mapping and dependency rules between capability features and other layers of features (technical and development features) it is possibly to close the gap between the problem space and the solution space [Kang *et al.* 1998].

In domain engineering the aim is to develop domain reusable artifacts. These artifacts can then be reused in the development of new applications in the domain (application engineering). Normally, artifacts to be reused are software architectures and software components. Apart from being used for domain engineering, features are also used in application engineering. User requirements can be 'translated' into feature selections and these selections used also to select an adequate architecture for the application (from reference domain architectures) and identify possible reusable components. To our knowledge, features and feature modeling are the major tools regarding domain engineering and, in particular, in documenting and specifying variability at the problem and design space.

One major issue is raised when realizing feature variability at the solution space. Domain methodologies recognize and normally apply three major binding times: compilation-time, load-time and run-time [Kang 1990]. Some authors argue that a more generic binding concept is needed because there can be specific times for a particular application or domain [Czarnecki 1998]. For this reason the concept of binding site was introduced to cover situations of both variations in time and location of features [Simos *et al.* 1996]. Given the fact that variations in applications can have diverse binding times there are, in fact, very limited proposed approaches to implement variability in run-time, or pos-deployment. As present in section 2, research has been done widely with pre-deployment techniques. For instance, GenVoca permit a very good mapping between features and implementation of variability but it's supported by compile-time composition of features. Other techniques, like generative programming, are based on static programming, i.e., they use a C++ template like programming to compose features [Czarnecki 1998]. Other authors present more pragmatic approaches to specify variability in software [Svahnberg and Bosch 2000]. Examples of such approaches are inheritance, parameterization, configuration and generation. Nevertheless they all have limitations in regard to run-time variability. Several authors recognize the limitations of these techniques [Czarnecki 1998]. They also seem to agree that it should be possible to specify components and their composition, manipulation and modification at different times. Our work so far also reveals that there are very little documented examples of run-time variability approaches.

In our approach to the run-time variability issue we see two (almost) orthogonal aspects:

(1) how to combine features at run-time, i.e., how do end-users select features at run-time;

(2) how to enable behavior specification holes in the applications for the end-users to fill at run-time.

Our research so far indicates that the first aspect is similar to feature composition techniques used at pre-run-time, like GenVoca. Nevertheless pos-compilation feature selection and composition raises some problems. One usual practice to achieve run-time variability is *table-driven* software. This technique has nonetheless one major limitation because it needs all the possible implementation of features to be present and compiled with the application. Another approach is to use some kind of dynamic code loading technique to load the selected feature implementation at run-time. To our knowledge this is a very used approach. It has the merit of having very little impact in the final application in terms of performance. It can also be used in an installation time selection of features via simple configuration files specifying with features will be used. The same technique of configuration files permits also a selection of features at instantiation time. So it is fair to say that this is a practice that has show good results so far. What is then the problem with this approach? We should not say that there is a problem but that there are limitations. Our point regard the fact that this kind of feature selection and composition is very limited compared to the possibilities that compile time techniques, like GenVoca, offer.

To reach the level of capabilities offered by compile time techniques in regard to feature selection and composition, run-time environments for applications need also to evolve. In section 2 it was visible that GenVoca offered the capability of layer composition by using a compile-time technique similar to inheritance. The interesting characteristic of this technique is that it was used to apply *inheritance* into components that could be composed from multiple classes. Each component (layer) implemented some feature and the interfaces needed to validate the composition where specified by means of a new construct called *realm*. To offer this kind of capabilities in a pos-compilation and pos-deployment time we propose an approach based on extending the run-time capabilities of the *operating* run-time environment. Our argument is that if the *operating* run-time environment exposes capabilities similar to ones existing at build time a feature selection and composition similar to the one made at compile-time should also be feasible. At this initial exploring phase of our work we purpose that the operating run-time environment should be extended with: (a) metadata; (b) reflection; and (c) run-time compilation. We assume that for the first aspect of our approach to run-time variability the end-user will only do feature selection and composition and so there isn't a crucial need for a high-level feature composition language. At the present it seems satisfactory to have a composition language similar to the one used with GenVoca.

Regarding the second aspect of our approach to the run-time variability issue the context in witch feature specification is done suffers a very significant change. In fact, what we are advocating in this aspect of run-time variability is that domain engineers and also application engineers will leave holes in their specification for the end-user to fill. This is a scenario where one or more features don't have possible implementations specified before run-time. This only occurs in very dynamic domains like the example of the insurance domain presented in section 1. In this scenario end-users will have to play a special role because they will have to fill the holes left in application engineering. This filling of holes basically corresponds to implementing features. In order to achieve this, the end-user will need to do some kind of program specification. This entails several implications: the end-user needs to assume a special role; development methodology has to be adapted to support such need; feature implementation/specification language has to be adapted to this special end-user.

Our work so far suggests that this role of run-time specification/implementation of features has to be played by a domain expert. In section 2 we presented an example of a vertical domain-specific language called RISLA. In this practical example the specification hole regards the behavior that different financial products can make to a bank application. In the example the special role of the

end-user was the one of an interest rate product engineer. The user playing this role uses RISLA as a high-level domain-specific interest rate product specification language.

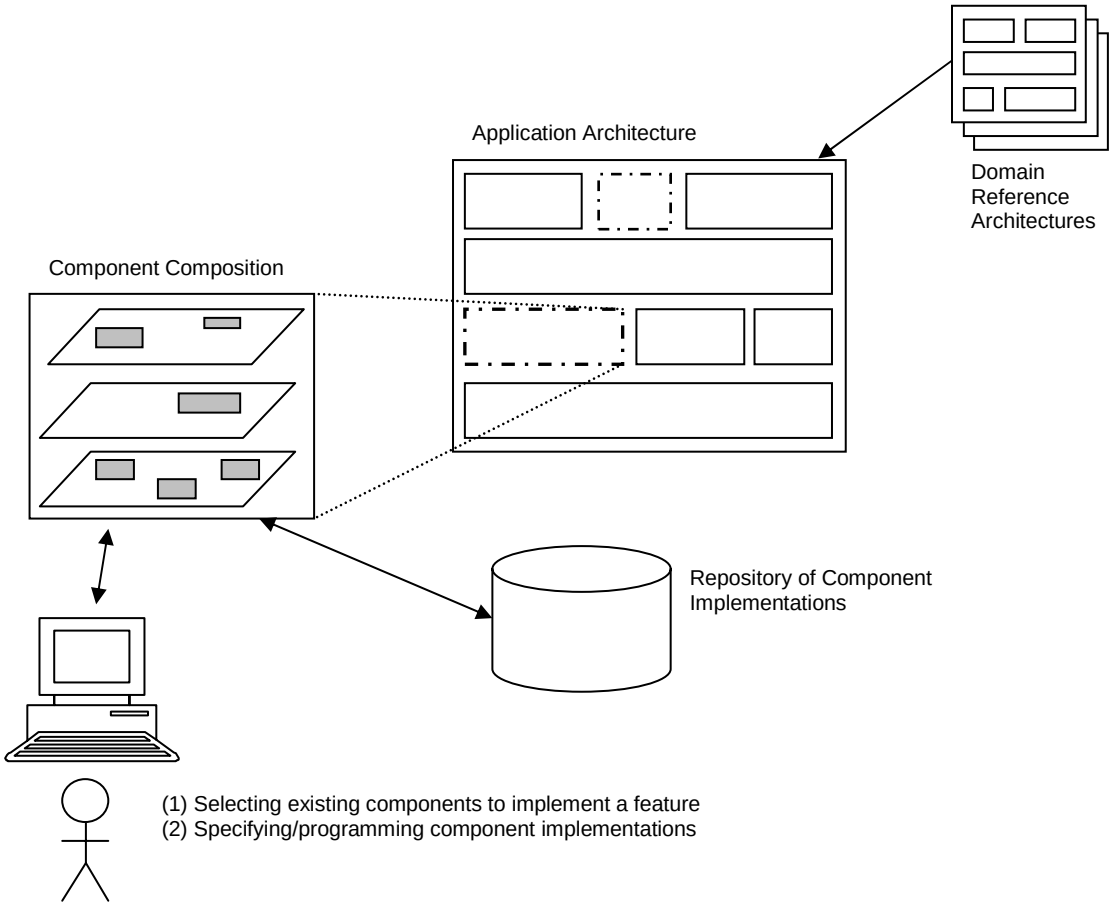

Figure 30. Combining features at run-time.

The development methodology may need also some adjustments. Domain engineering admits features that have run-time binding times (or binding sites). This means that the feature implementation will be selected at run-time. This does not normally mean that the concrete feature implementation will be done (i.e., programmed) at run-time. This will have implications in the domain engineering methodology, particularly, in the binding site specification for features and in the mapping from design space to solution space (artifacts). As noted in the next section we also plan to study the methodological implications (such as this one) of our work.

One other issue regarding dynamic run-time variability is the need for a special kind of run-time programming activity. We already mentioned the need for a new run-time role. We suggest the name of *application domain engineer* for this new role. This role is different from that of a domain engineer and also is different from a domain application engineer. An *application domain engineer* is a domain expert that is also an application expert, i.e., he can adapt/program an application in a pos-deployment time. A domain application engineer is someone who builds applications in a domain through the use of domain artifacts.

An *application domain engineer* is neither a programmer nor a software engineer. He is a domain expert that also needs to have a very high application expertise. Because dynamic run-time variability implies run-time feature programming this means the *application domain engineer* needs to have some kind of programming skill. This shouldn't be like a generic programming skill but more like a domain programming logic skill. An *application domain engineer* should use a domain-specific language with a grammar and semantic very close to the *natural* domain language and concepts. For instance, RISLA aims at achieving this goal. In our work we also propose the adoption of DSLs as means to specify dynamic run-time feature implementation. This means that a domain with dynamic run-time feature implementation needs a DSL. So, the domain engineering process needs to produce a DSL. In section 2 we present some approaches into the design and implementation of DSLs. Regarding this aspect we propose to study the integration of DSLs in domain engineering based on features and feature diagrams. Feature diagrams are used to model domains with diverse feature binding sites. To our knowledge supporting dynamic run-time features could enrich feature diagrams. Our study so far indicates that dynamic run-time features could be implemented (in a majority of cases) by very simple programming logic connecting existing high level domain concepts, witch in turn, are also features of the domain. If this is true, then adapted feature diagrams could be used to extract the design requirements of the domain DSL and also to design a library of resources to be used in the language programs (i.e., components).

Our approach to run-time variability and all the mentioned issues needs is based on an operating run-time infrastructure with the necessary services. We mentioned some of them: reflection, meta-data and run-time compilation. These are necessary not only from a functional perspective but also from a performance perspective. To support these requirements, and also to adopt a technology that could be more easily accessible for a vast audience, we intend to use a CLI compliant framework [CLI] for concept demonstration and prototyping. We will also use a proprietary technology in our case study that will be briefly presented in the next section.

4 Work Plan

We plan to develop our work in two parallel lines: a case study line and a generic line. In the case study line we will work in a specific domain with a domain-specific language and framework (*proprietary run-time*). These are proprietary tools developed and used by a Portuguese company specialized in software solutions for the insurance industry. In the second line of work we purpose to generalize and apply our work using general available technology. As we mentioned before we will be doing this using a CLI compliant framework (*generic run-time*).

Our plan clearly adopts an applied research methodology. In fact, we will be integrated in a company and in a working team. As such we will interact with and work within the context of this team. Such a research method has its risks but also as many advantages. The advantages start with the possibility to have a realistic case study and experiment and validation context for our research. One major risk is the eventually impossibility to full control the development of the research actions. In accepting these risks we are aware that we do not control all the outcomes of our work. We try to minimize this possibility with the second line of work mentioned above. This research method can clearly be classified as an *action research* method [Baskerville 2001].

After our first year work documented in this report we plan to carry out the future two years work with the following four major phases or activities: a) static run-time variability (generalization); b) static run-time variability (domain case study); c) dynamic run-time variability (domain case study); and d) dynamic run-time variability (generalization).

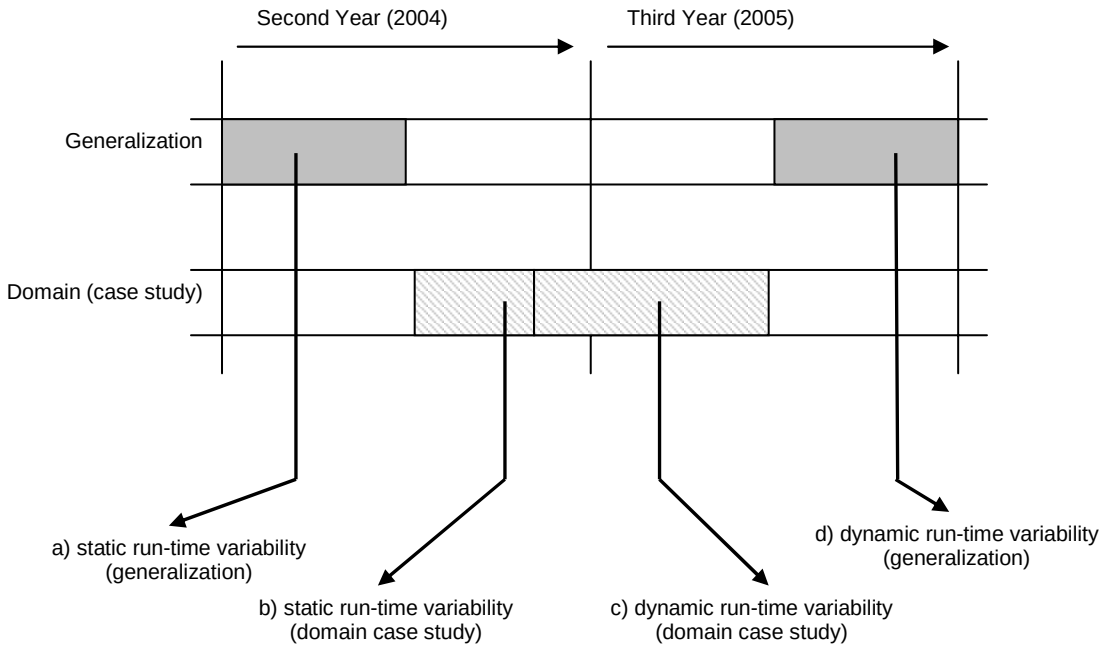

Figure 31. Future work plan.

The temporal sequencing of the four phases can be seen in Figure 31. The two first phases regard static run-time variability and the last two phases regard dynamic run-time variability. The details of each phase are presented next.

In the first phase of our work plan we will focus our research on static run-time variability. This phase will be developed on the first half of 2004. We will start by further increase the knowledge of methods and techniques used for specifying and implementing variation at pre-run-time. This study will focus on feature modeling and static composition of components, i.e., techniques similar to the ones applied in GenVoca and domain-specific languages like DJ. We propose to study adaptation of these techniques to a generic run-time operational environment. We will try to *translate* documented examples of pre-run-time component composition into a pos-run-time environment. We propose to validate this work mainly by comparing with similar functionalities of pre-run-time component selection and composition. We plan the results of this phase will be feature modeling adaptations for support the specification of static run-time variability and technology implementation requirements of run-time operational environment to support static run-time variability.

In the first four months of second half of 2004 we plan on using the results from the previous phase in the case study. We will apply these results to the case study. In this phase we plan on focus our research on validating the proprietary run-time operational environment for proper support of static run-time variability. We plan on doing so based on the outcome of the previous phase. This will probably result in possible suggestions for adapting the proprietary run-time. These proposed actions should be presented to the enterprise development team for validation and possible integration in the development/evolution planning for the proprietary run-time. Time permitting, experiments (prototype) with the proprietary run-time could be done and documented. Adaptations of the proprietary run-time also represent a concrete case of realization of the results from the previous phase (and can be viewed as a validation). The main output of this phase will be the possible suggestions for adapting the proprietary run-time and the documentation of a concrete realization of static run-time variability requirements for a proprietary run-time environment.

After validating static run-time variability on the case study we plan to continue our work on the case study, this time with the focus on dynamic run-time variability. This phase of our work will have a duration of eight months, from November 2004 until June 2005.

In this phase we will explore our propose for implementing dynamic run-time variability in applications. As already exposed, our propose is based on the adoption of domain-specific languages. We will start our work on dynamic run-time variability in the case study. The enterprise case study we will be working on already uses a domain-specific language and run-time environment in its product line. A practical application of this language and run-time has been presented in [Bragança 1998]. The language and run-time are used in the enterprise product line to implement dynamic run-time variability aspects. So, been this (to our knowledge) an open research area it makes sense to base our research in this practical case. This will be the core of our work.

We plan to validate our work on this phase using applications from the enterprise product line and (if possible) end-users and the enterprise software engineers. We also plan the outputs of this phase to be feature-modeling adaptations for support the specification of dynamic run-time variability and technology implementation requirements of run-time operational environment to support dynamic run-time variability.

During the last half of 2005, in the last phase of our work, we plan on generalizing the results of our research. Particularly we plan on generalize the knowledge resulting from the previous phase

using general available technology. The work done on the previous phase is based on proprietary technology and thus this phase will adapt it to a generic run-time framework. This will be done mainly at an implementation perspective since methodological aspects will be almost totally covered. Nevertheless we preview a possible small working effort in adapting methods. Validation in this phase will be done through experiments with requirements similar to the ones of the case study of the last phase. Results will be compared with the ones obtained in the previous phase. The major output from this last phase will be technology implementation requirements for a run-time operational environment to support dynamic run-time variability.

A Ph.D. work apart from the research activity should have at least two others parallel activities: the proper Ph.D. thesis writing and publication of research results (i.e. scientific papers that permit the sharing of experiences and knowledge with the scientific community in the field of study).

Figure 32presents the principal relations between our research activities, the writing of the thesis chapters and the publication of research results.

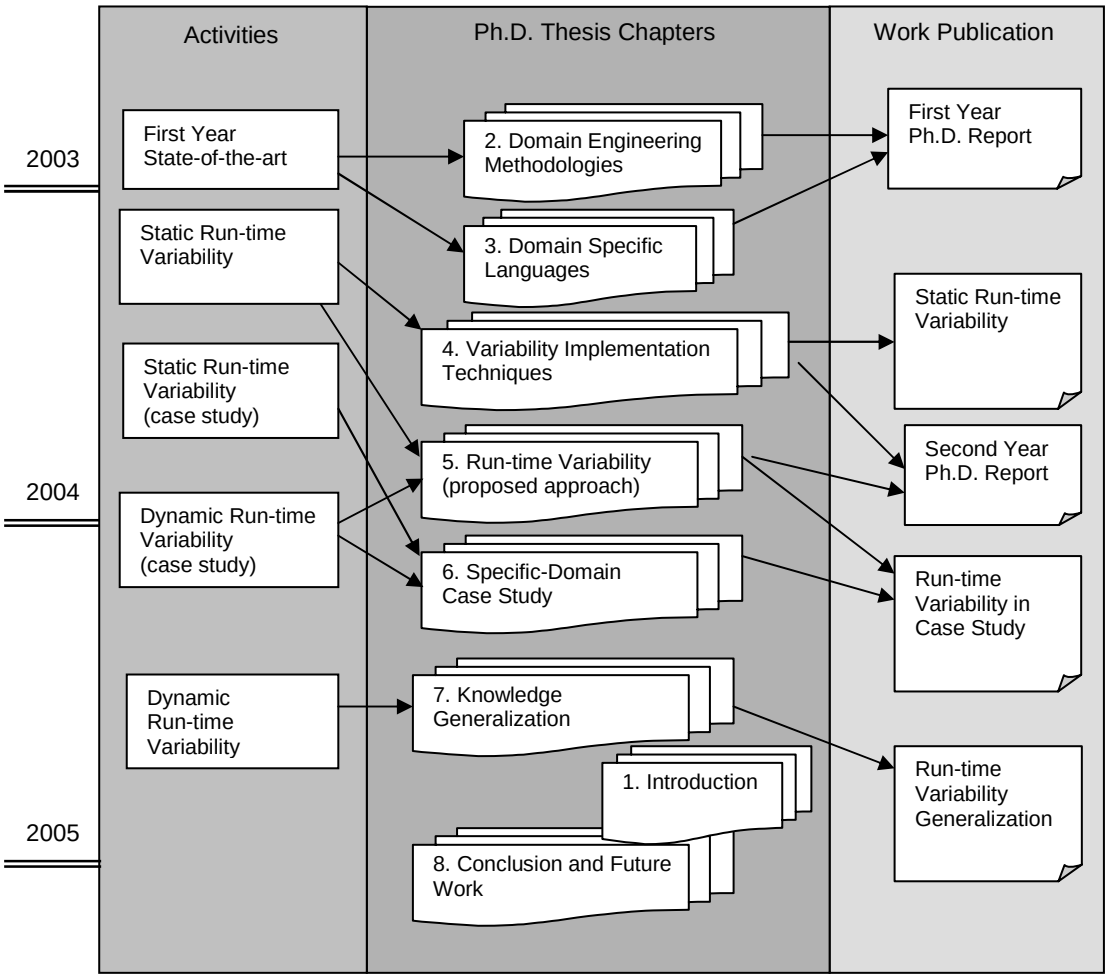

Figure 32. Ph.D. Activities vs. thesis chapters vs. work publication.

This technical report results from the first year work. The first year was mainly dedicated to study the research field and produce a state-of-the-art. This work also resulted in the writing of chapters 2 and 3 of the thesis. These chapters present the state-of-the-art in the research field.

The activities of the first half of 2004 (i.e., *phase a*) will result in material to complete the chapter 4 of the thesis. This chapter is dedicated to present techniques to implement variability. It will be based on study made during the first year and also from the results of *phase a*. We see feasible the publication of these results in a scientific paper.

The next two phases (*b* and *c*) will produce a significant part of the results of our work. These two phases regard the core of the thesis, i.e., supporting dynamic run-time variability based on domain-specific languages and operating run-time environments. As such, during phase's *b* and *c*, the chapter presenting and detailing the thesis will be written. Phase *c* will also be the source of a chapter reporting the case study. In the final of phase *c* (end of first half of 2005) we also intend to write a scientific paper to present the results from our work.

The last phase of our work (phase *d*) will be focused on generalizing our work into a public well-known run-time environment. The results of this phase will also lead to the writing of a scientific paper reporting our results.

The major parts of first (Introduction) and last (Conclusion and Future Work) chapters of the thesis will be written as the last step of the Ph.D. work. As a result these chapters will reflect a broader look of the field and approached issues.

5 Expected Contributions

The major goal of domain engineering is to enable a high productive mechanism for software reuse. As we saw, to achieve this goal adequate methodologies, tools and techniques must be used. The principal issue regarding domain engineering is how to identify, represent and implement commonalities and variability in the domain space and solution space. A common feature will enable reuse and a variable feature will provide application differentiation within a domain. Without variable features all applications are equal and there will be not productivity gain because there will be only one application.

Variability is the core issue of this thesis. We presented the state-of-the-art of domain engineering within variability identification, representation and implementation. We also identified actual limitations regarding run-time application feature variability implementation. This limitation also extends to the methodology level. We then presented our approach to this problem based on the study we made so far.

We laid out a work plan to develop and validate our approach and publish the results. We expect our proposed research to contribute to the field in the following ways:
- ***Case study contribution.*** We will present and document an industry real case that implements applications with run-time variability features.
- ***Methodological contribution***. Features diagrams and in general feature modeling is very limited regarding run-time variability specification. We expect to contribute with adaptations to feature modeling in order to better support run-time variability.
- ***Technological contribution***. Current examples and implementations of variability rely almost only on pre-deployment techniques. We will study run-time variability techniques and technologies based on a proprietary domain-specific language and framework. We expect to generalize and adapt our results so that it could be the base for general available run-time variability supporting tools.

References

[.Net Framework] .Net Framework web site at Microsoft, http://msdn.microsoft.com/netframework/.

[Arnold *et al*. 1995] B.R.T. Arnold, A. van Deursen and M. Res, *An Algebraic Specification of a Language for Describing Financial Products*, ICSE-17 Workshop on Formal Methods Application in Software Engineering, pp. 6-13, IEEE, April 1995.

[Baskerville 2001] Richard Baskerville, *Conducting Action Research: High Risk and High Rewardin Theory and Practice*, Qualitative Research in Information Systems, pp. 192-218, Idea Group Publishing, 2001.

[Basu *et al.* 1997] A. Basu, M. Hayden, G. Morrisett and T. von Eicken, *A language-based approach to protocol construction*, Proceedings of DSL '97 - First ACM SIGPLAN Workshop on Domain-Specific Languages, in Association with POPL '97, Paris, France, University of Illinois Computer Science Report, pp. 1-15, January 1997.

[Batory and O'Malley 1992] D. Batory and S. O'Malley, *The Design and Implementation of Hierarchical Software Systems with Reusable Components,* ACM Transactions on Software Engineering and Methodology, vol. 1, no. 4, pp. 355-398, October 1992.

[Batory *et al.* 1988] D. Batory, J.R. Barnett, J.F. Garza, K.P. Smith, K. Tuskuda, B.C. Twichell, and T.E. Wise, *GENESIS: An Extensible Database Management System*, IEEE Transactions on Software Engineering, vol. 14, no. 11, pp. 1711-1730, November 1988.

[Beck 1999] Kent Beck, *Extreme Programming Explained*, Addison-Wesley, 1999.

[Bentley 1986] J. L. Bentley, *Programming pearls: Little languages*, Communications of the ACM, vol. 29, no. 8, pp. 711-721, August, 1986.

[Borgida *et al.* 1984] Alexander Borgida, John Mylopoulos and Harry K. T. Wong, *Generalization/Specialization as a Basis for Software Specifications*, On Conceptual Modeling, Book resulting from the Interleave Workshop 1992, pp. 87-117, Springer-Verlag, New York, 1984.

[Bosch 2000] Jan Bosch, *Design and Use of Software Architectures Adopting and Evolving a Product-Line Approach*, Addison-Wesley, 2000.

[Bosch 2002] Jan Bosh, *Maturity and Evolution in Software Product Lines: Approaches, Artifacts and Organization*, Proceedings of the Second Software Product Line Conference (SPLC2), Lecture Notes in Computer Science, vol. 2379, pp. 257-271, Springer-Verlag, 2002.

[Bragança 1998] Alexandre Bragança, *Sistema para Gestão Operacional de Processos, Uma Visão Global na Indústria e nos Serviços no Contexto dos Sistemas de Fluxo de Trabalho*, Master Thesis, Faculdade de Engenharia da Universidade do Porto, April 1998.

[Brand *et al.* 1996] M. van den Brand, A. van Deursen, P. Klint, S. Klusener and E. van der Meulen, *Industrial applications of ASF+SDF*, Algebraic Methodology and Software Technology (AMAST '96), Lecture Notes in Computer Science, vol. 1101, pp. 9-18. Springer-Verlag, 1996.

[Cambridge] Cambridge Dictionaries Online, http://dictionary.cambridge.org/.

[Chen 1976] P. P. Chen, *The Entity-Relationship Model- Toward a Unified View of Data,* ACM Transactions on Database Systems, vol. 1, no. 1, pp. 9-36, March 1976.

[CLI] CLI ECMA (International-European association for standardizing information and communication systems) web site, http://www.ecma-international.org/memento/TC39-TG3.htm.

[Czarnecki 1998] Krzysztof Czarnecki, *Generative Programming Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models*, Ph.D. Thesis, Department of Computer Science and Automation, Technical University of Ilmenau, 1998.

[Deursen 1997] A. van Deursen, *Domain-specific languages versus object-oriented frameworks: A financial engineering case study*, Proceedings Smalltalk and Java in Industry and Academia, STJA'97, pp. 35-39, Ilmenau Technical University, 1997.

[Deursen *et al.* 1996] A. van Deursen, J. Heering and P. Klint, *Language Prototyping: An Algebraic Specification Approach,* AMAST Series in Computing, vol. 5, World Scientific Publishing Co., 1996.

[Deursen *et al.* 2000] A. van Deursen, P. Klint and J. Visser, *Domain-Specific Languages: An Annotated Bibliography*, ACM SIGPLAN Notices, vol. 35, no. 6, pp. 26-36, June 2000.

[Deursen *et al.* 2002] Arie van Deursen, Merijn de Jonge and Tobias Kuipers, *Feature-Based Product Line Instantiation using source-level packages*, Proceedings of the Second Software Product Line Conference (SPLC2), Lecture Notes in Computer Science, vol. 2379, pp. 217-234, Springer-Verlag, 2002.

[Fayad and Johnson 1999] Mohamed E. Fayad and Ralph E. Johnson, *Domain-Specific Application Frameworks: Framework Experience by Industry*, John Wiley & Sons, 1999.

[Foreman 1996] John Foreman, *Product Line Based Software Development- Significant Results, Future Challenges*, Proceedings of the Software Technology Conference, Salt Lake City, April 1996.

[Fowler 2002] Martin Fowler, *Patterns of Enterprise Application Architecture*, Addison-Wesley Pub Co., 2002.

[Garlan and Shaw 1990] David Garlan and Mary Shaw, *An Introduction to Software Architecture*, Carnegie Mellon University Technical Report CMU-CS-94-166, January 1994.

[Gomaa 1984] Hasan Gomaa, *A Software Design Method for Real-Time Systems*, Communications of the ACM, vol. 27, no. 9, pp. 938-949, September 1984.

[Griss *et al.* 1998] M. L. Griss, J. Favaro, and M. d'Alessandro, *Integrating Feature Modeling with the RSEB*, Proceedings of the Fifth International Conference on Software Reuse (Victoria, Canada, June 1998), IEEE Computer Society Press, pp. 76-85, 1998.

[Gurp 2003] Jilles van Gurp, *On the Design & Preservation of Software Systems*, Ph.D. Thesis, Computer Science Department, University of Groningen, 2003.

[Hayes 1994] F. Hayes-Roth, *Architecture-Based Acquisition and Development of Software: Guidelines and Recommendations from the ARPA Domain-Specific Software Architecture (DSSA) Program*, Version 1.01, Informal Technical Report, Teknowledge Federal Systems, February 4, 1994.

[Herndon and Berzins 1988] R. M. Herndon and V. A. Berzins, *The realizable benefits of a language prototyping language*, IEEE Transactions on Software Engineering, vol. 14, no. 6, pp. 803-809, 1988.

[Hudak 1996] P. Hudak, *Building domain-specific embedded languages*, ACM Computing Surveys, vol. 28, no. 4, December 1996.

[Hudak 1998] P. Hudak, *Modular Domain Specific Languages and Tools*, Proceedings of the Fifth International Conference on Software Reuse, IEEE Computer Society Press, pp. 134-142, Victoria, Canada, June 1998.

[Jacobson *et al.* 1992] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard, *Object-Oriented Software Engineering*, Addison-Wesley, Workingham, England, 1992.

[Jacobson *et al.* 1997] I. Jacobson, M. Griss, and P. Jonsson, *Software Reuse: Architecture, Process and Organization for Business Success*, Addison Wesley Longman, May 1997.

[Jaring *et al.* 2002] Michel Jaring, Jan Bosch, *Representing Variability in Software Product Lines: A case study*, Proceedings of the Second Software Product Line Conference (SPLC2), pp. 15-36, Lecture Notes in Computer Science, vol. 2379, Springer-Verlag, 2002.

[Java] Java language web site at Sun, http://java.sun.com/.

[Kang *et al.* 1990] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William, E. Novak and A. Spencer Peterson, *Feature-Oriented Domain Analysis (FODA) Feasibility Study Technical Report*, CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, November 1990.

[Kang *et al.* 1998] Kyo C. Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Gerard Jounghyun Kim and Euiseob Shin, *FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures*, Annals of Software Engineering, vol. 5, pp. 143-168, Kluwer Academic Publishers, 1998.

[Kiczales *et al.* 1997] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier and John Irwin, *Aspect oriented programming*, Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland, Lecture Notes in Computer Science, vol. 1241, Springer-Verlag, June 1997.

[Kieburtz *et al.* 1996] R. B. Kieburtz, L. McKinney, J. M. Bell, J. Hook, A. Kotov, J. Lewis, D. P. Oliva, T. Sheard, I. Smith and L. Walton, *A software engineering experiment in software component generation*, Proceedings of the 18th International Conference on Software Engineering ICSE-18, pp. 542-553, IEEE, 1996.

[Lopes 1997] Cristina Lopes, *D: A language Framework For Distributed Programming*, Ph.D. Thesis, College of Computer Science, Northeastern University, 1997.

[McLeod 1978] Dennis McLeod, *A Semantic Data Base Model and its Associated Structured User Interface*, Ph.D. Thesis, Massachusetts Institute of Technology, 1978.

[Medvidovic 1997] Medvidovic, N., *A Classification and Comparison Framework for Software Architecture Description Languages*, Report, UCI-ICS-97-02, University of California, Irvine, January, 1997.

[Myers 1988] Brad A. Myers, *A Taxonomy of Window Manager User Interfaces*, IEEE Transactions on Computer Graphics & Applications, vol. 8, no. 5, pp. 65-84, September, 1988.

[Naur *et al.* 1969] Naur, P. and B. Randell (eds.), *Software Engineering*, Conference sponsored by the Nato Science Commite, NATO, 1969.

[Neighbors 1980] J. Neighbors, *Software Construction Using Components*, Ph.D. Thesis, Department Information and Computer Science, University of California, Irvine, 1980.

[Neighbors 1984] J. M. Neighbors, *The Draco approach to constructing software from reusable components*, IEEE Transactions on Software Engineering, vol. 10, no. 5, pp. 64-74, September 1984.

[O'Malley and Peterson 1992] S. W. O'Malley and L. L. Peterson, *A dynamic network architecture*, ACM Transactions on Computer Systems, vol. 10, no. 2, pp. 110-143, May 1992.

[Ossher *et al.* 1994] Harold Ossher, William Harrison, Frank Budinsky, Ian Simmonds, *Subject-Oriented Programming: Supporting Decentralized Development of Objects*, Proceedings of the 7th IBM Conference on Object-Oriented Technology, July 1994.

[Parnas 1976] D. Parnas, *On the Design and Development of Program Families*, IEEE Transactions on Software Engineering, vol. 2, no. 1, pp. 1-9, 1976.

[Peterson and Stanley 1994] A. Spencer Peterson and Jay L. Stanley, Jr., *Mapping a Domain Model and Architecture to a Generic Design*, Carnegie Mellon University/Software Engineering Institute, Technical Report CMU/SEI-94-TR-8, May 1994.

[Phoenix] Phoenix web site at Microsoft Research, http://research.microsoft.com/phoenix/.

[Pressman 1994] Roger S. Pressman, *Software Engineering A Practitioner's Approach*, McGraw Hill, 1994.

[Prieto-Diaz 1990] Ruben Prieto-Diaz, *Domain Analysis: An Introduction*, ACM SIGSOFT Software Engineering Notes, vol. 15, no. 2, pp. 47-54, April, 1990.

[UML] Unified Modeling Language (UML v1.5) web site at Rational Software Corporation, http://www.rational.com/uml/.

[Reenskaug *et al.* 1996] T. Reenskaug, P. Wold and O.A. Lehne, *Working with Objects: The OOram Software Engineering Method*, Manning, 1996.

[RUP] Rational Unified Process web site at IBM, http://www-3.ibm.com/software/awdtools/rup/.

[SEI] Software Engineering Institute web site, http://www.sei.cmu.edu/.

[Simonyi 1995] Charles Simonyi, *The death of Computer languages, the birth of intentional programming*, Microsoft Research Technical Report, MSR-TR-95-52, September 1995.

[Simos *et al.* 1996] M. Simos, D. Creps, C. Klinger, L. Levine, and D. Allemang, *Organization Domain Modeling (ODM) Guidebook*, Version 2.0, Informal Technical Report for STARS, STARS-VC-A025/001/00, June 14, 1996.

[Singhal and Batory 1993] V. Singhal and D. Batory, *P++: A Language for Large-Scale Reusable Software Components*, Proceedings of the 6th Annual Workshop on Software Reuse, Owego, New York, 1993.

[Spinellis 2001] Diomidis Spinellis, *Notable design patterns for domain specific languages*, ACM Journal of Systems and Software, vol. 56, no. 1, pp. 91-99, February 2001.

[Svahnberg and Bosch 2000] Mikael Svahnberg and Jan Bosch, *Issues Concerning Variability in Software Product Line*, Proceedings of the Third International Workshop on Software Architectures for Product Families, Springer Verlag, 2000.

[Thibault 1998] S. Thibault, *Domain Specific Languages: Conception, Implementation and Application*, Ph.D. Thesis, Université de Rennes 1, 1998.

[UTCAT] University of Texas Center for Agile Technology web site, http://www.cat.utexas.edu/dsl.html.

[wxWindows] wxWindows project web site, http://www.wxwindows.org/.

[Yourdon *et al*. 1978] Edward Yourdon and L. Constantine, *Structured Design*, Yourdon Press, 1978.